Corso di Calcolatori Elettronici I

Sottoprogrammi

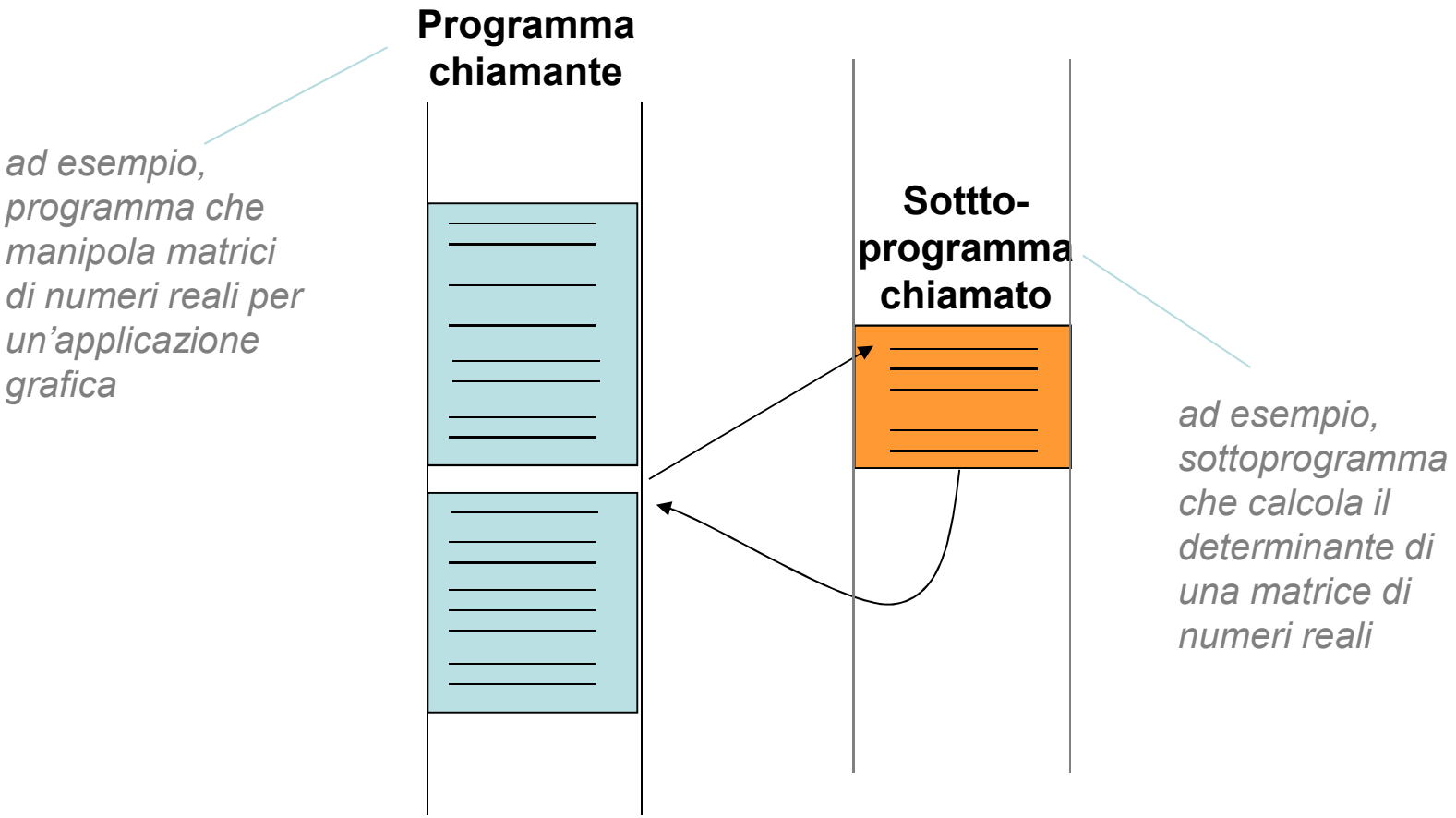
ing. Alessandro Cilardo

Corso di Laurea in Ingegneria Biomedica

Sottoprogrammi

- E' utile e spesso necessario scomporre i programmi in **sottoprogrammi**
- Sottoprogramma o *subroutine*
  - un segmento di codice che svolge un'elaborazione su dati forniti in input e restituisce eventualmente risultati in output, indipendentemente da altri segmenti
- Realizza un'*astrazione* procedurale, fornendo un servizio attraverso un'interfaccia
- Consente un'organizzazione modulare dei programmi

Istruzioni di collegamento a sottoprogramma



Sottoprogrammi: parametri

- Un sottoprogramma scambia con il programma chiamante dei dati sia in input che in output
- Le informazioni scambiate sono dette ***parametri***
- Nel testo del sottoprogramma, ci si riferisce ai parametri mediante dei nomi simbolici: i *parametri formali*
- L'attivazione (o invocazione) di un sottoprogramma richiede che ai parametri formali siano assegnate degli opportuni valori: *parametri effettivi*

Sottoprogrammi: problematiche

- **Collegamento** (o ***linkage***)
  - modo in cui un calcolatore rende possibili le operazioni di *chiamata* e di *ritorno* delle procedure, gestendo opportunamente gli indirizzi delle istruzioni
- **Passaggio dei parametri**
  - insieme di convenzioni sulle modalità di *scambio dei parametri*, che definiscono:
    - quali informazioni si scambiano il programma chiamante ed il sottoprogramma ("cosa"), sia in *ingresso* che in *uscita* ovvero al *ritorno* del sottoprogramma
    - attraverso quali meccanismi avviene lo scambio (memoria, registri, …) ("come")

Scambio dei parametri

Nei linguaggi procedurali di alto livello:

- "cosa" va passato, ovvero l'elenco ed il tipo dei parametri di scambio viene definito nella dichiarazione della procedura:
  - Esempio: `void prod (int a, int b, int *p)`
- "come" avviene lo scambio è convenuto dal linguaggio: la convenzione è implementata dal compilatore

Se si programma in assembler, è il programmatore che stabilisce ed implementa le convenzioni

- a meno che non si debba scrivere moduli assembler che interagiscono con moduli C o di altro linguaggio di alto livello
- in tal caso occorre che il programmatore rispetti le convenzioni del compilatore

Sottoprogrammi: procedure e funzioni

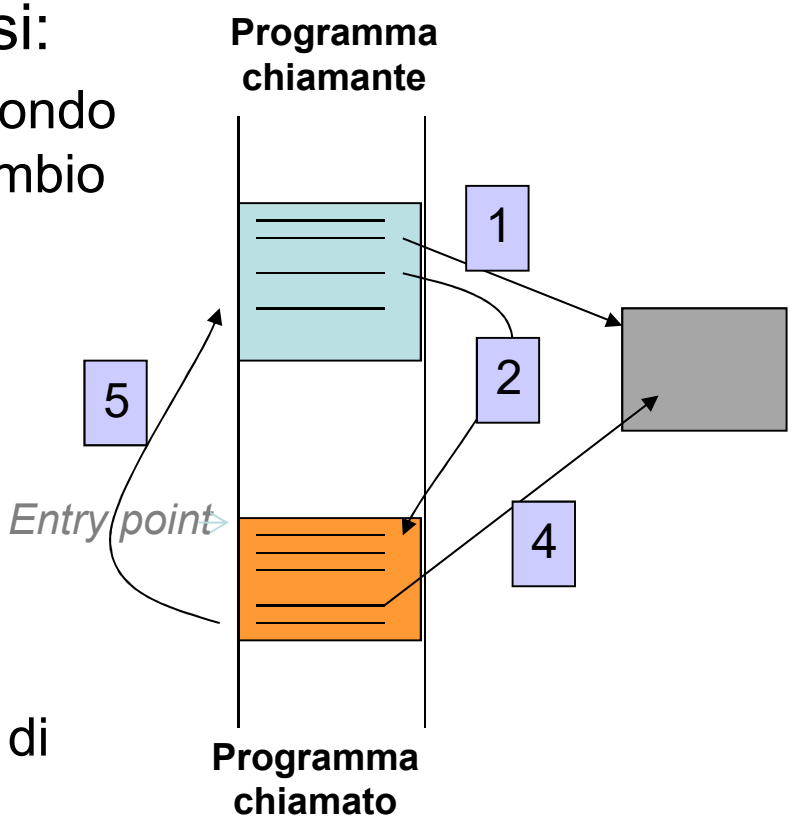
- In alcuni linguaggi di alto livello (Pascal) si distingue esplicitamente tra
  - *funzioni*: sottoprogrammi al cui nome è associato un valore "di ritorno" assegnabile ad una variabile

    esempio: `a := max(b, c);`
  - *procedure*, che invece compiono solo una elaborazione sui parametri di scambio ma non hanno un valore di ritorno

    esempio: `stampa(vettore, n);`
- In C tutti i sottoprogrammi sono detti *funzioni*. Le procedure hanno un dato di ritorno di tipo `void`

Sottoprogrammi: esecuzione

- L'esecuzione di un sottoprogramma avviene mediante i seguenti passi:
  1. Scrittura dei parametri effettivi secondo la convenzione stabilita per lo scambio
  2. Trasferimento del controllo al sottoprogramma ("chiamata")
  3. Esecuzione del sottoprogramma
  4. Scrittura dei parametri di output
  5. Trasferimento del controllo al programma principale ("ritorno dal sottoprogramma") ed esecuzione dell'istruzione successiva a quella di chiamata



Esempio (in C)

- Programma chiamante:

*Chiamata al sottoprogramma con passaggio di parametri (**parametri effettivi**)*

```
. . .
det = calcolaDeterminante(myMat, dim);
. . .
```

- Sottoprogramma:

*Dichiarazione del sottoprogramma, con **parametri formali** e valore di **ritorno***

```
float calcolaDeterminante(float *f, int n){
    . . .
}
```

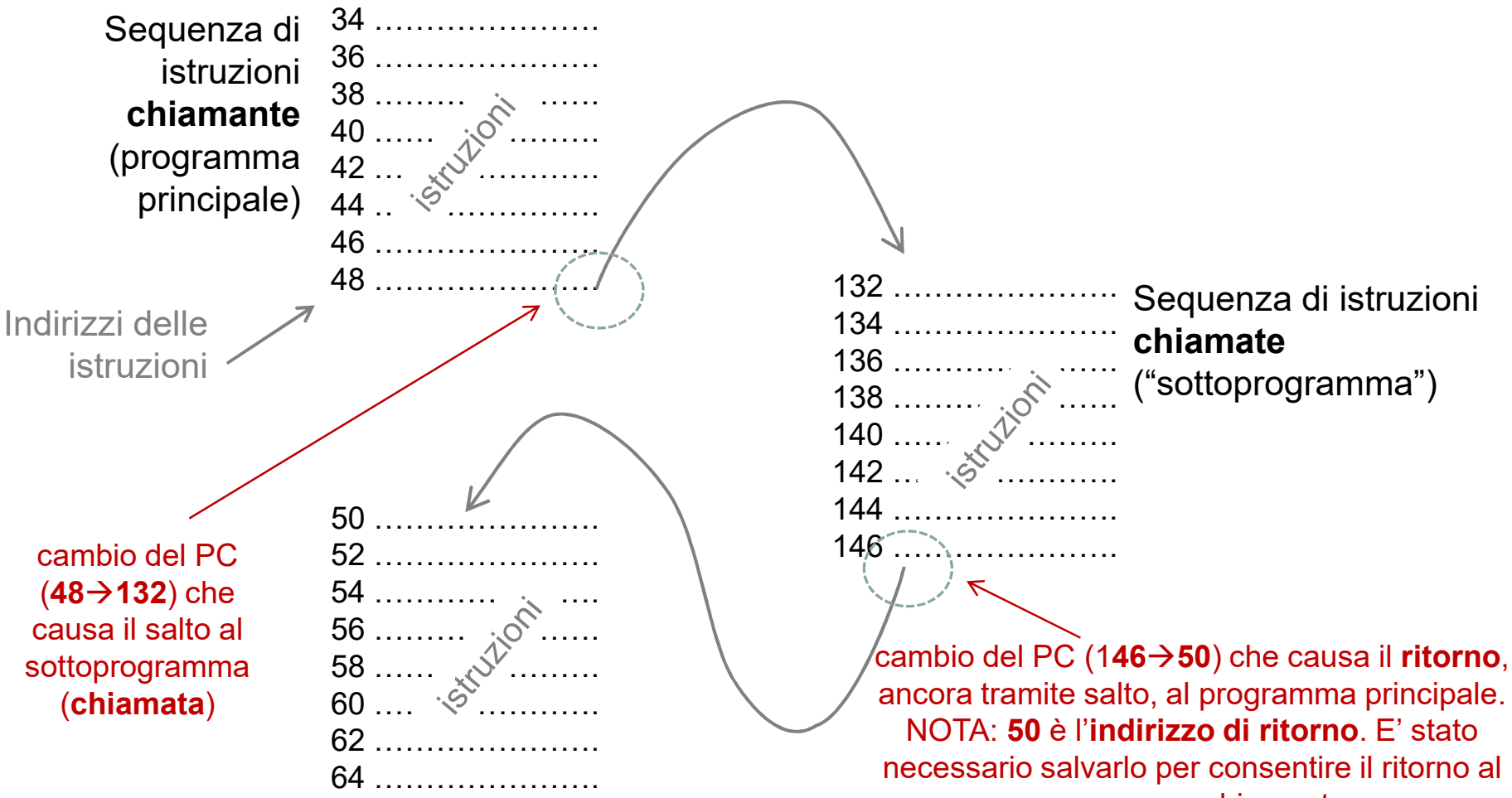
Sottoprogrammi in assembly

- La chiamata di un sottoprogramma interrompe l'esecuzione sequenziale delle istruzioni del programma chiamante e pertanto si presenta come un caso particolare di salto
- L'indirizzo della prima istruzione della subroutine (*entry point*) deve essere caricato nel *Program Counter*
- A differenza di un normale salto, occorre prevedere un meccanismo per il ritorno al chiamante

Istruzioni di collegamento a sottoprogramma

- Le istruzioni di salto a sottoprogramma (*Jump To Subroutine* o *Call*) salvano il valore del PC per consentire il ritorno al programma chiamante
- Le istruzioni di ritorno da sottoprogramma (*Return From Subroutine*) ripristinano il valore del PC salvato per realizzare il ritorno al programma chiamante
- Il valore del PC può essere salvato in un apposito registro (*Link Register*, LR) e/o in una opportuna area di memoria chiamata stack
- Esempi di istruzioni per la chiamata di subroutine:
  - Motorola 68000: **`jsr label`**, **`bsr label`**
  - Intel 8086: **`call label`**
  - PowerPC e MIPS: **`jal label`**

Istruzioni di collegamento a sottoprogramma



Collegamento (Linkage)

- Il problema di consentire il ritorno dal sottoprogramma al programma chiamante, o meglio all'istruzione del programma chiamante successiva alla chiamata, è detto problema di *collegamento del sottoprogramma* (*subroutine linkage*)
- Tre soluzioni sono possibili:
  - L'indirizzo di ritorno è salvato in un registro di macchina
  - L'indirizzo di ritorno è salvato in una particolare locazione di memoria (ad esempio, nelle locazioni immediatamente precedenti l'entry point della subroutine)
  - L'indirizzo di ritorno è salvato in una particolare locazione di memoria associata al programma chiamante
  - L'indirizzo di ritorno è salvato in un'area di memoria separata detta stack

Link register

- In alcune architetture (PowerPC, MIPS) un'istruzione di *jump&link* carica l'indirizzo di ritorno in un registro del processore (*Link Register*) e successivamente effettua il salto. Nell'esempio, il *Link Register* è chiamato **`$ra`** :

  `jal label` Effetto: **`[PC]→$ra, label→PC`**

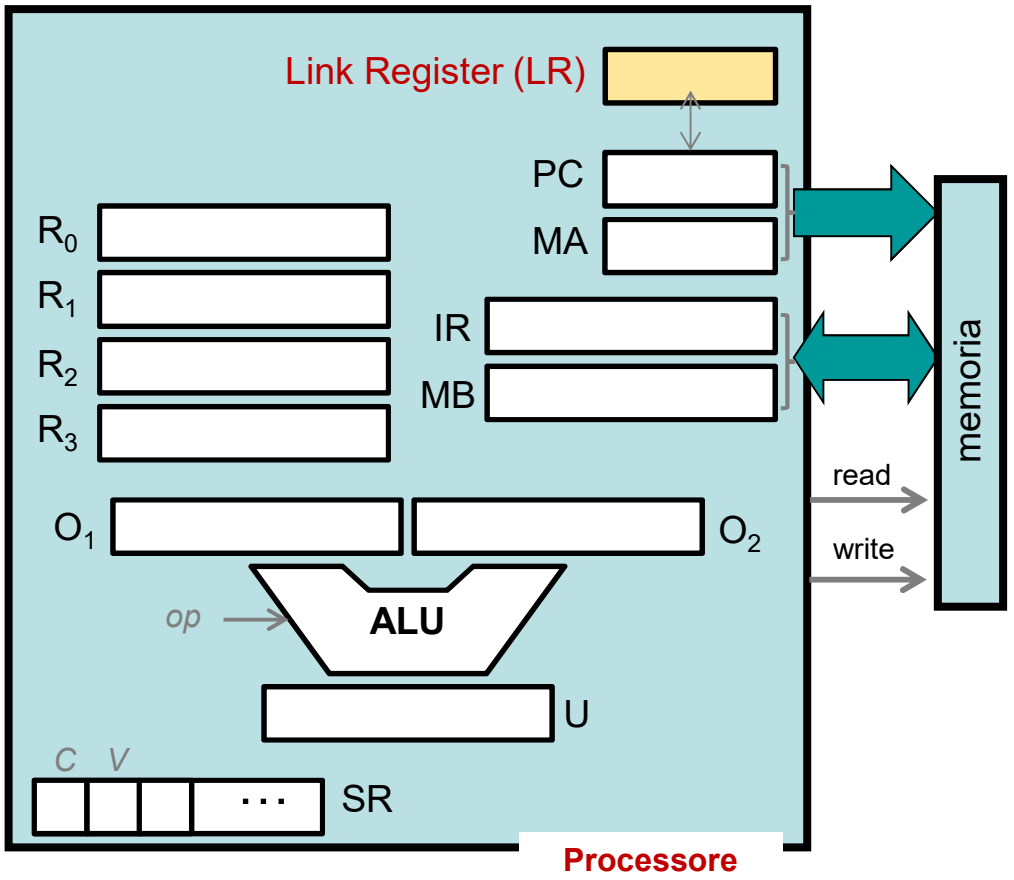
- Un'altra istruzione effettua l'operazione reciproca, il caricamento nel PC del valore salvato nel registro di link:

  `jr $ra` Effetto: **`[$ra] → PC`**

- Tuttavia, il nesting delle subroutine e la ricorsione richiedono il salvataggio del link register all'interno di una memoria esterna

Link register

- Al momento della chiamata, prima di cambiarne il valore, il contenuto del **PC** è copiato nel Link Register (**LR**)



Linkage tramite memoria

- Sono possibili due varianti:
  - Il sottoprogramma definisce un'area di memoria usata per lo scambio dei parametri
  - Il programma chiamante alloca un'area di memoria per lo scambio dei parametri e ne passa l'indirizzo iniziale al sottoprogramma

    Quest'ultimo accede ai parametri effettivi mediante un opportuno spiazzamento rispetto all'indirizzo base (ad es., usando un modo di indirizzamento indiretto con *displacement*)

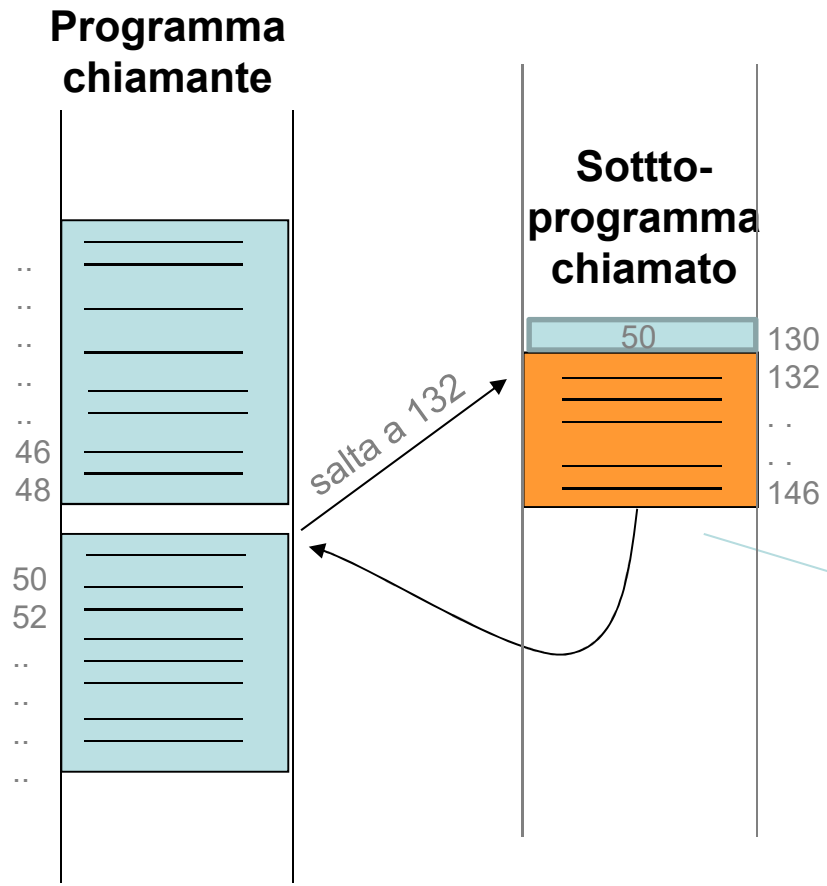
Linkage tramite memoria

- Si può ad es. stabilire una convenzione per la quale l'indirizzo di ritorno è salvato *nelle locazioni precedenti l'entry point* della subroutine
- Inconvenienti:
  - non consente chiamate multiple
  - non può essere usata se il codice della subroutine è in ROM
  - non consente il nesting, la ricorsione, …

```
MAIN   MOVE.L #RET1,SUBR
       JMP SUBR+4
RET1   ..istr. successiva
         del chiamante. . .
         . . .

SUBR   DS.L  1
      ..prima istruzione
      ..altre istr. della
            subroutine...
         . . .
       MOVEA.L SUBR,A0
       JMP (A0)
```

In questo esempio, **RET1** è l'indirizzo "*di ritorno*", quello a cui dovrà saltare la subroutine al termine. Tale indirizzo è salvato all'indirizzo **SUBR**. L'*entry point* della subroutine è invece all'indirizzo **SUBR+4**

Linkage tramite memoria



Immediatamente prima della chiamata, il programma chiamante **copia** *il valore del* **PC** *all'interno della locazione di memoria riservata in testa al sottoprogramma (ad esempio, nella locazione* **130**, *quella che precede l'entry point del programma)*

Entry-point (prima istruzione) del sottoprogramma (indirizzo **132**)

*Al termine dell'esecuzione, il sottoprogramma legge l'indirizzo di ritorno (***50**) *dall'apposita locazione e lo ripristina nel registro* **PC** *del processore. Ciò determina il* **ritorno** *al programma chiamante*

Linkage tramite memoria: problema

- Se sono in esecuzione **diversi programmi** contemporaneamente, ed entrambi chiamano lo stesso sottoprogramma, *oppure*
- se il sottoprogramma chiama se stesso, realizzando una **chiamata ricorsiva**

  (*ovvero, per entrambi i casi, se accade che un sottoprogramma sia chiamato mentre ne è già in esecuzione un'altra "**istanza**"*)
- … allora la locazione per l'indirizzo di ritorno verrà sovrascritta, e si perderà l'indirizzo di ritorno per la prima chiamata

Passaggio di parametri tramite Stack

- Il modo più efficace per gestire aree di memoria per lo scambio di parametri è il cosidetto ***stack***
  - *stack* (“pila”) fa riferimento al fatto che ciascun dati non è memorizzato sempre nello stessa locazione, ma piuttosto alla prima libera
  - precisamente ogni elemento è memorizzato nella locazione successiva a quella in cui è stato posto il dato oggetto della precedente memorizzazione
  - man mano che i dati vengono memorizzati, essi si “accatastano” o “impilano” l’uno sull’altro nella memoria

Stack

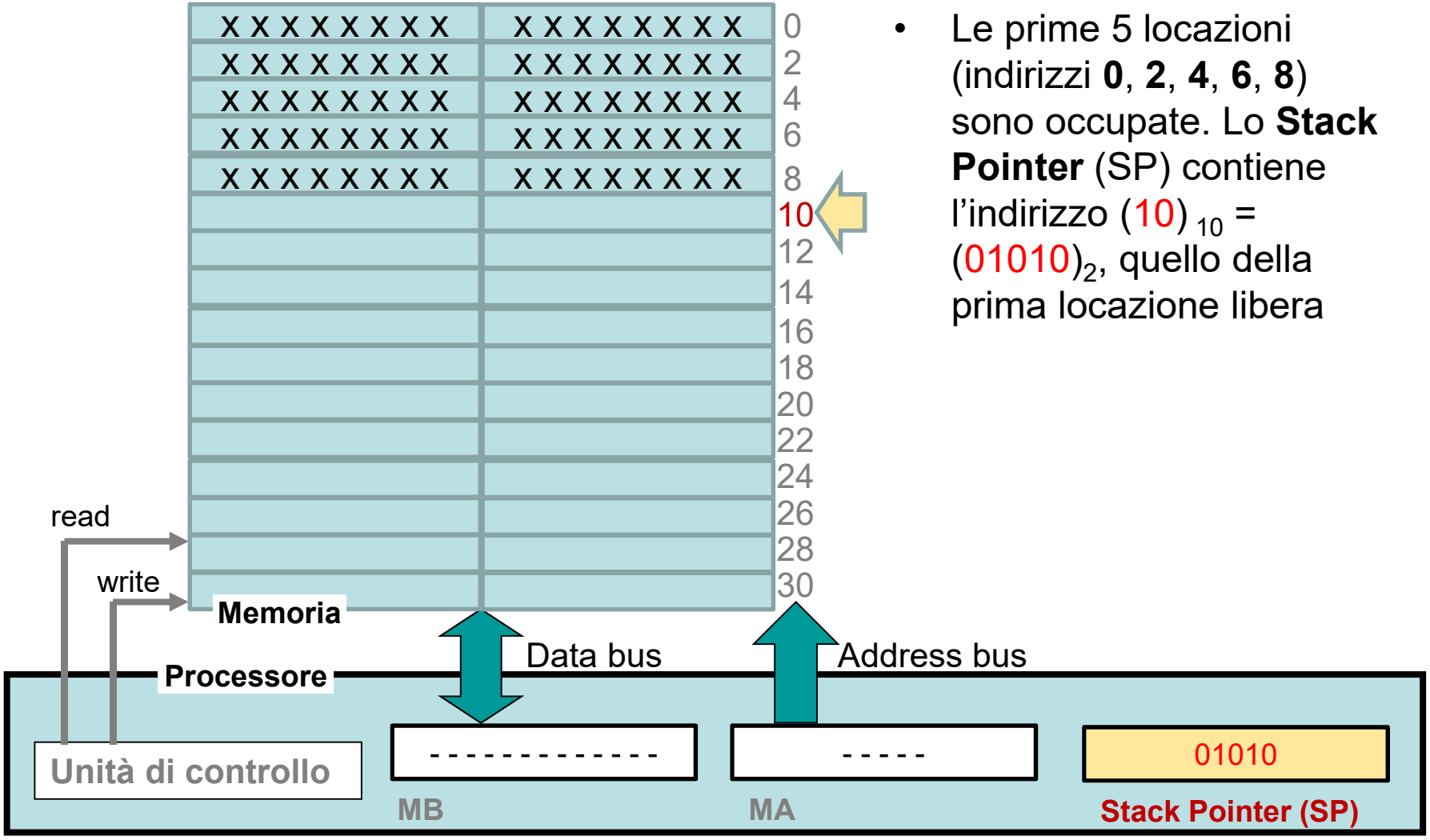
- Lo stack è gestito tramite un puntatore, tipicamente un registro, che memorizza l'indirizzo della prima locazione libera in memoria
  - quasi sempre chiamato ***Stack Pointer*** (SP)

- Il processo di memorizzazione di un dato (***push***) consiste nel porre il dato stesso in memoria all'indirizzo **SP**, incrementando poi **SP** in modo che *continui a contenere l'indirizzo della prima locazione libera*

Stack

- Il processo di prelievo di un dato dalla memoria (***pop***) consiste nell'effettuare in maniera speculare le operazioni del ***push***:
  - viene prima decrementato **SP** in modo da contenere l'indirizzo dell'ultimo dato che era stato inserito, dopodiché viene fatta la lettura
  - la locazione è quel punto considerata libera (in quanto puntata dall'indirizzo contenuto in **SP**)

Esempio di funzionamento dello Stack

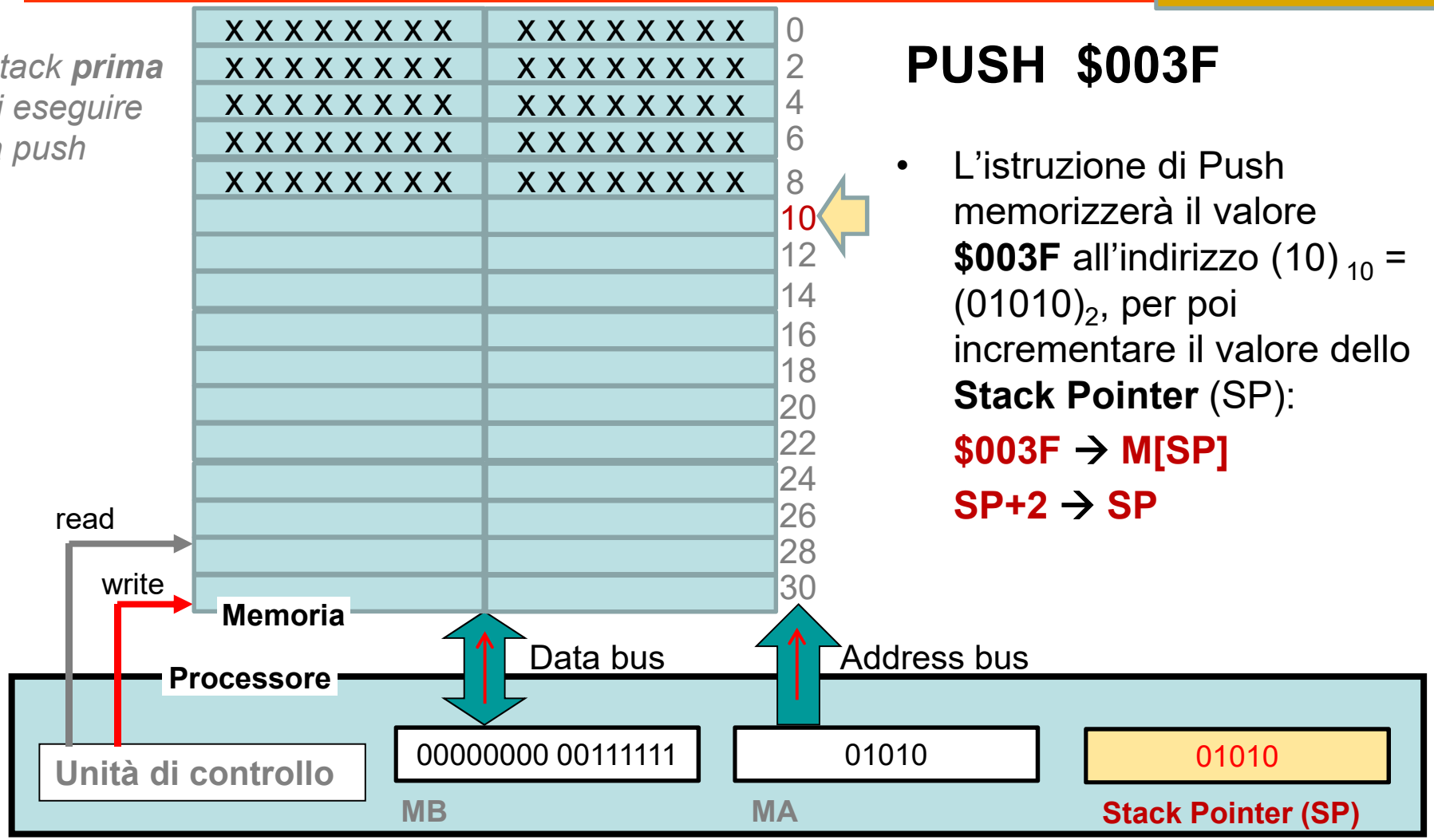


- Le prime 5 locazioni (indirizzi **0**, **2**, **4**, **6**, **8**) sono occupate. Lo **Stack Pointer** (SP) contiene l'indirizzo $(10)_{10}$ = $(01010)_2$, quello della prima locazione libera

Esempio di funzionamento dello Stack

Prima istruzione

*Stack **prima** di eseguire la push*



PUSH $003F

- L'istruzione di Push memorizzerà il valore **$003F** all'indirizzo $(10)_{10} = (01010)_2$, per poi incrementare il valore dello **Stack Pointer** (SP):

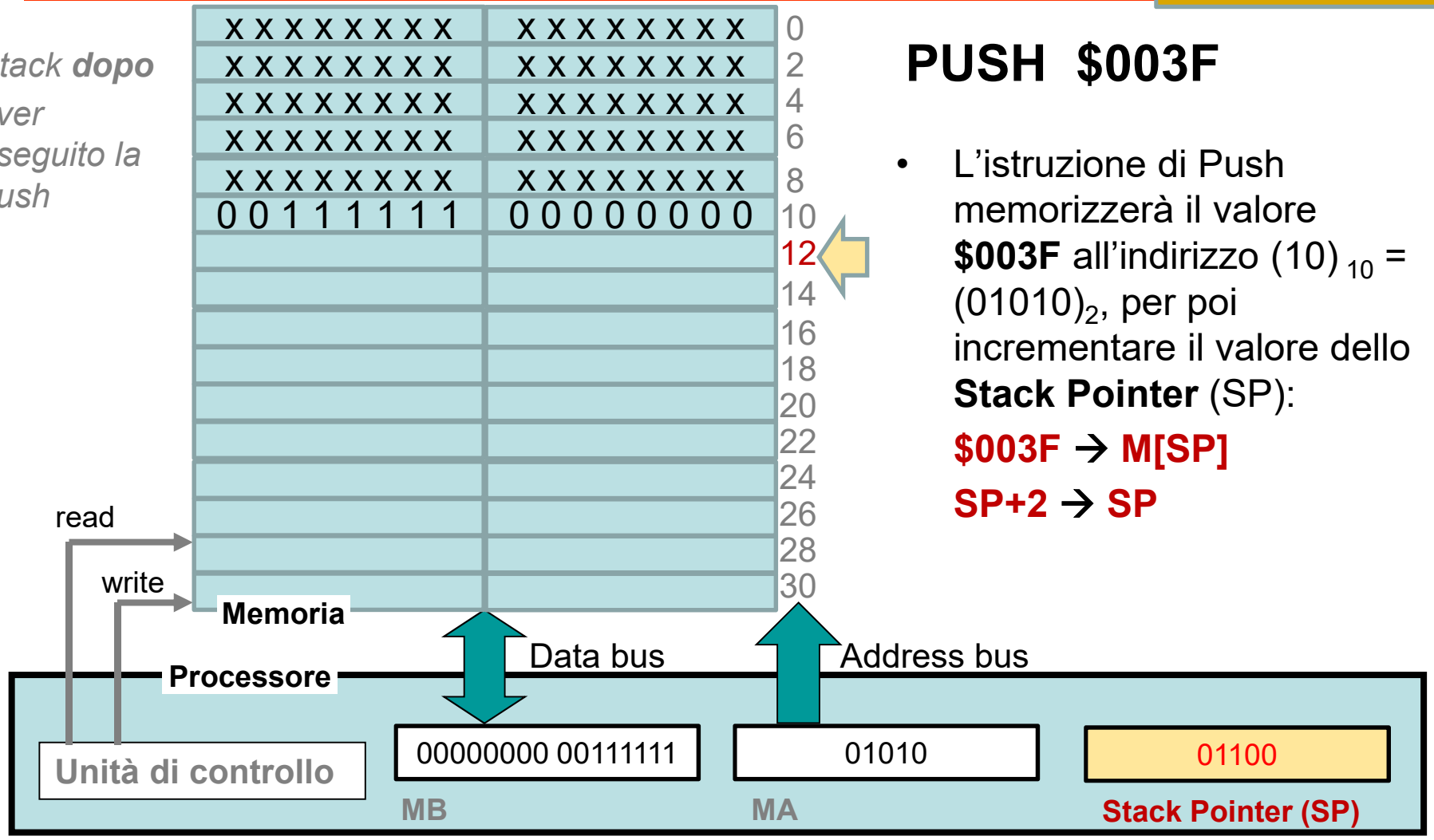
$003F → M[SP]

SP+2 → SP

Esempio di funzionamento dello Stack

Prima istruzione

*Stack **dopo** aver eseguito la push*



PUSH $003F

- L'istruzione di Push memorizzerà il valore **$003F** all'indirizzo $(10)_{10}$ = $(01010)_2$, per poi incrementare il valore dello **Stack Pointer** (SP):

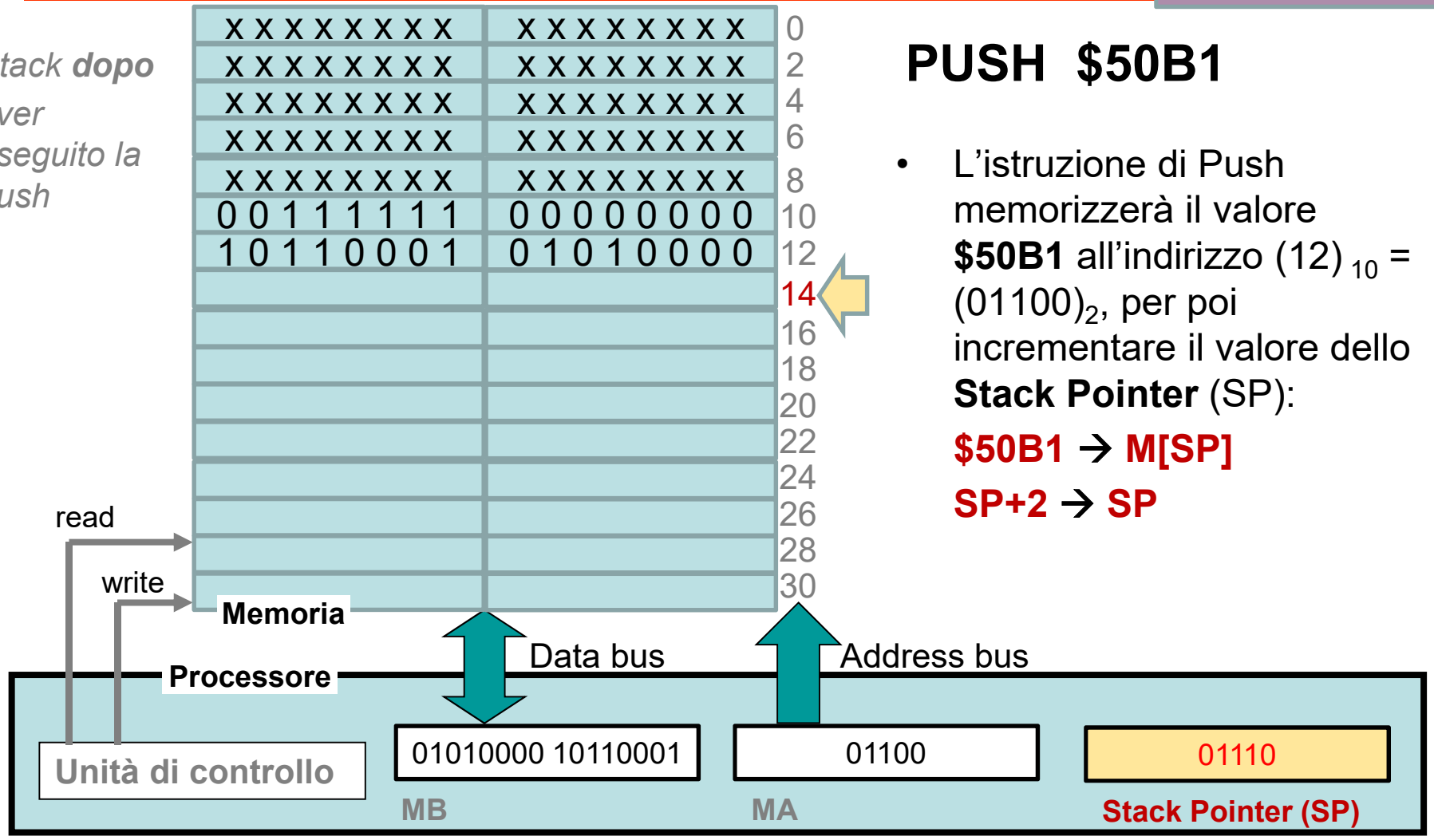
$003F → M[SP]

SP+2 → SP

Esempio di funzionamento dello Stack

Seconda istruzione

*Stack **dopo** aver eseguito la push*



PUSH $50B1

- L'istruzione di Push memorizzerà il valore **$50B1** all'indirizzo $(12)_{10}$ = $(01100)_2$, per poi incrementare il valore dello **Stack Pointer** (SP):

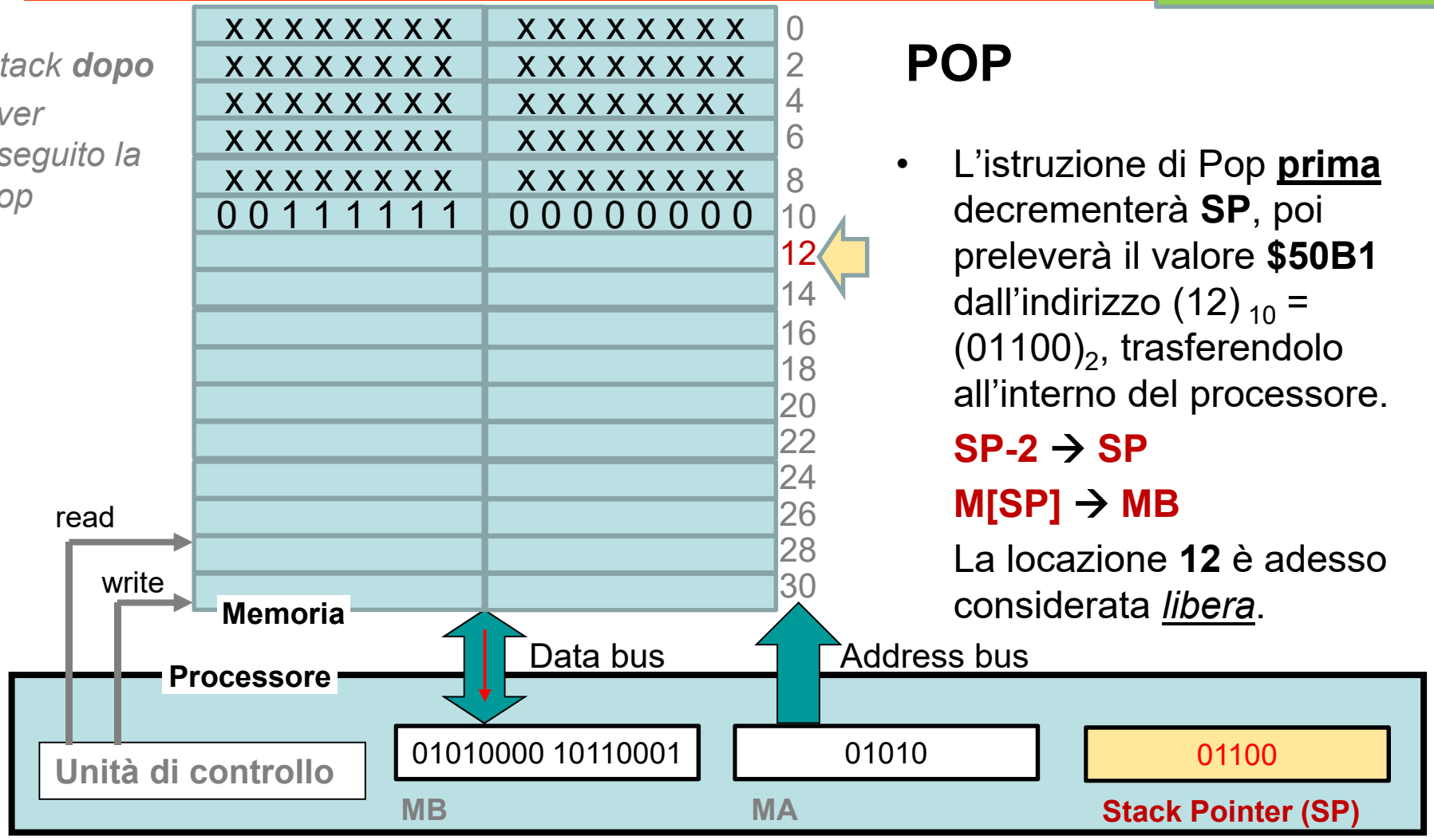
$50B1 → M[SP]

SP+2 → SP

Esempio di funzionamento dello Stack

Terza istruzione

*Stack **dopo** aver eseguito la pop*



POP

- L'istruzione di Pop **prima** decrementerà **SP**, poi preleverà il valore **$50B1** dall'indirizzo $(12)_{10}$ = $(01100)_2$, trasferendolo all'interno del processore.

  SP-2 → SP

  M[SP] → MB

  La locazione **12** è adesso considerata *libera*.

Accesso allo Stack

- **Importante**: Le uniche operazioni di accesso allo stack sono quelle di ***push*** per l'inserimento e ***pop*** per il prelievo
- Non è possibile accedere a posizione arbitrarie, ma solo all'ultima, la *cima* dello stack
  - la locazione è quel punto considerata libera (in quanto puntata dall'indirizzo contenuto in **SP**)
- Inserimenti e prelievi vanno sempre fatti in maniera speculare (*Last in First Out* , LIFO)
  - il primo dato letto sarà l'ultimo inserito (la cima)
  - non è possibile "saltare" ad elementi interni

Organizzazione dello stack

- Processori differenti adottano scelte differenti per l'organizzazione dello stack:
  - *cima dello stack*: **SP** è l'indirizzo della prima locazione libera (come nell'esempio) o dell'ultima occupata
  - *direzione*: **SP** parte da un valore di indirizzo basso e viene incrementato con i push (come nell'esempio), o parte da un valore alto e viene decrementato con i push (stack a crescere/decrescere)
- Le due scelte sono indipendenti e possono essere combinate in qualsiasi modo

Supporto per lo Stack

- Di base, non è un meccanismo fisico e non richiede un cambiamento all'architettura del processore
- E' un modo di organizzare e gestire la memoria
- Spesso i processori forniscono anche un supporto architetturale, tipicamente limitato alla presenza di un registro **stack pointer** dedicato, ad di istruzioni di ***push*** e ***pop***, ed al salvataggio automatico dell'indirizzo di ritorno in occasione della chiamata a funzione.

Stack nel processore M68000

- Il M68000 adotta un'organizzazione dello stack a *decrescere* con **SP** che punta all'*ultima locazione occupata*.
- Il registro **A7** è un registro speciale usato esplicitamente o implicitamente da alcune istruzioni come Stack Poiter
  - **A7** è anche indicato come **SP** in assembler
- **A7'** è uno Stack Pointer fisicamente diverso da **A7**, usato in modalità supervisore (ad esempio, dal codice del sistema operativo)
- Non sono presenti istruzioni esplicite di ***push*** e ***pop***.
- ***push*** ottenuto come predecremento: `MOVE.L D1,-(SP)`
- ***pop*** ottenuto come postincremento: `MOVE (SP)+,D3`

Stack nel processore M68000

- In alcuni processori (es. il Motorola 68000 e nella famiglia Intel x86) l'istruzione di salto a subroutine salva *automaticamente* l'indirizzo di ritorno sulla cima dello stack:

  `jsr subr` Effetto: **`push(SP, PC); subr → PC`**

- Reciprocamente, un'istruzione di ritorno da subroutine consente il ripristino dell'indirizzo di ritorno dalla cima dello stack

  `rts` Effetto: **`pop(SP, PC)`**

Esempio con JSR

File: `programma020.a68`

- Scrivere una subroutine che calcola il valore assoluto del registro **D0**:

  |**D0**| → **D0**

- Chiamare la subrouting da un programma principale, che la applica a tutti gli elementi di un vettore di interi con segno di tipo Long
- Osservare il comportamento dello stack (puntato da **A7'**) durante l'invocazione della subroutine

Esempio con JSR

File: `programma020.a68`

```
        ORG             $8000
START   MOVE    N,D1
        SUBQ            #1,D1
        MOVEA.L         #VET,A2
LOOP    MOVE.L          (A2),D0
        JSR                     ABS
        MOVE.L          D0,(A2)+
        DBRA            D1,LOOP
        ORG             $8100
VET     DC.L            -3,4,-4,3,5,3,-1,2
N       DC.W            8

        ORG $8200
ABS     CMP.L   #0,D0
        BGE     RET
        NOT.L   D0
        ADDQ.L  #1,D0
RET     RTS
        END             START
```

Passaggio di parametri

- Problematiche per la gestione dei sottoprogrammi:
  - Linkage (*visto prima*)
  - Passaggio dei parametri

Passaggio di parametri

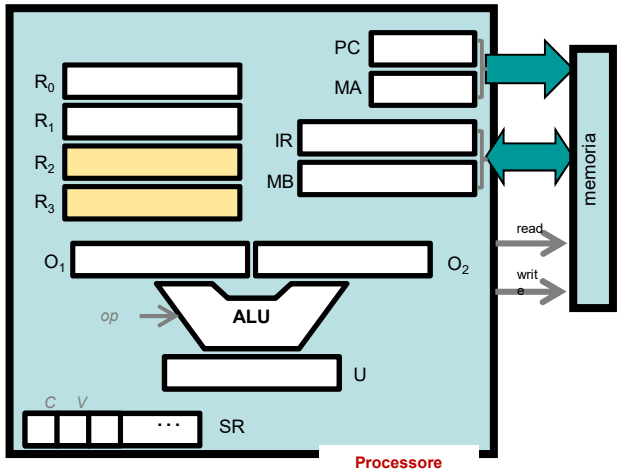
- Il problema in questo caso non è salvare un singolo, specifico valore (quello del **PC** prima del salto alla subroutine), come accade per il linkage
- Bisogna invece rendere accessibili al sottoprogramma i valori dei *parametri* su cui il sottoprogramma lavorerà
  - in generale, possono essere in numero e di dimensione qualsiasi
- Bisogna inoltre che il sottoprogramma possa rendere accessibili i *risultati* al programma chiamante al ritorno

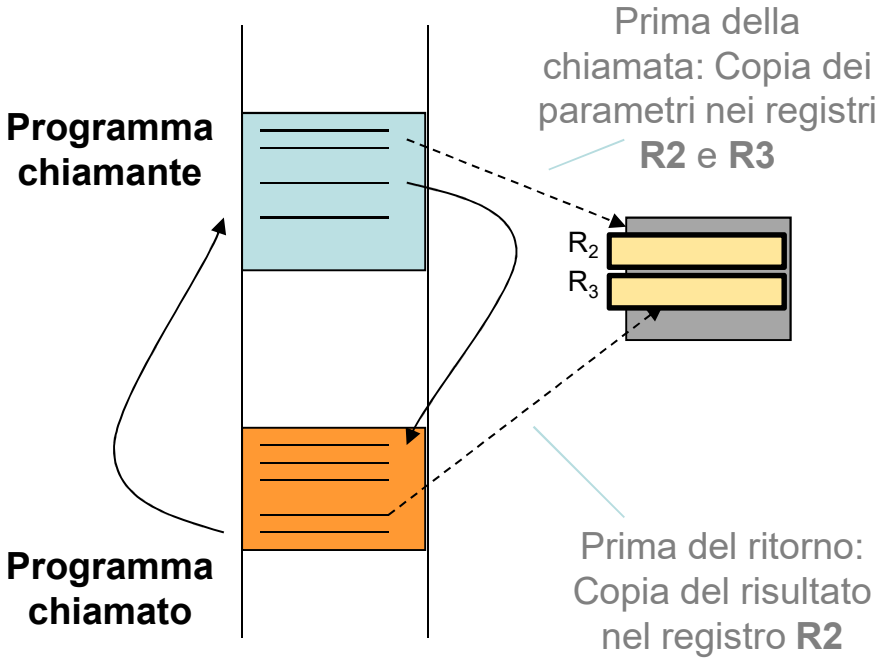
Passaggio dei parametri con registri

- E' la tecnica più veloce
  - non richiede accessi in memoria
- Programma chiamante e sottoprogramma si accordano sull'uso dei registri
  - i parametri di ingresso sono messi sempre in alcuni specifici registri, dove il programma chiamato si aspetterà di trovarli
  - viceversa per i risultati, che saranno scritti dal sottoprogramma in alcuni registri prefissati, dove il programma chiamante si aspetterà di trovarli
- Il numero ridotto di registri generali del processore rappresenta **un limite**

Passaggio dei parametri con registri

- Ad esempio: il sottoprogramma riceve due interi, che si aspetta di trovare nei registri **R2** e **R3**. Produce inoltre un risultato, che il programma chiamante si aspetterà di trovare nel registro **R2**



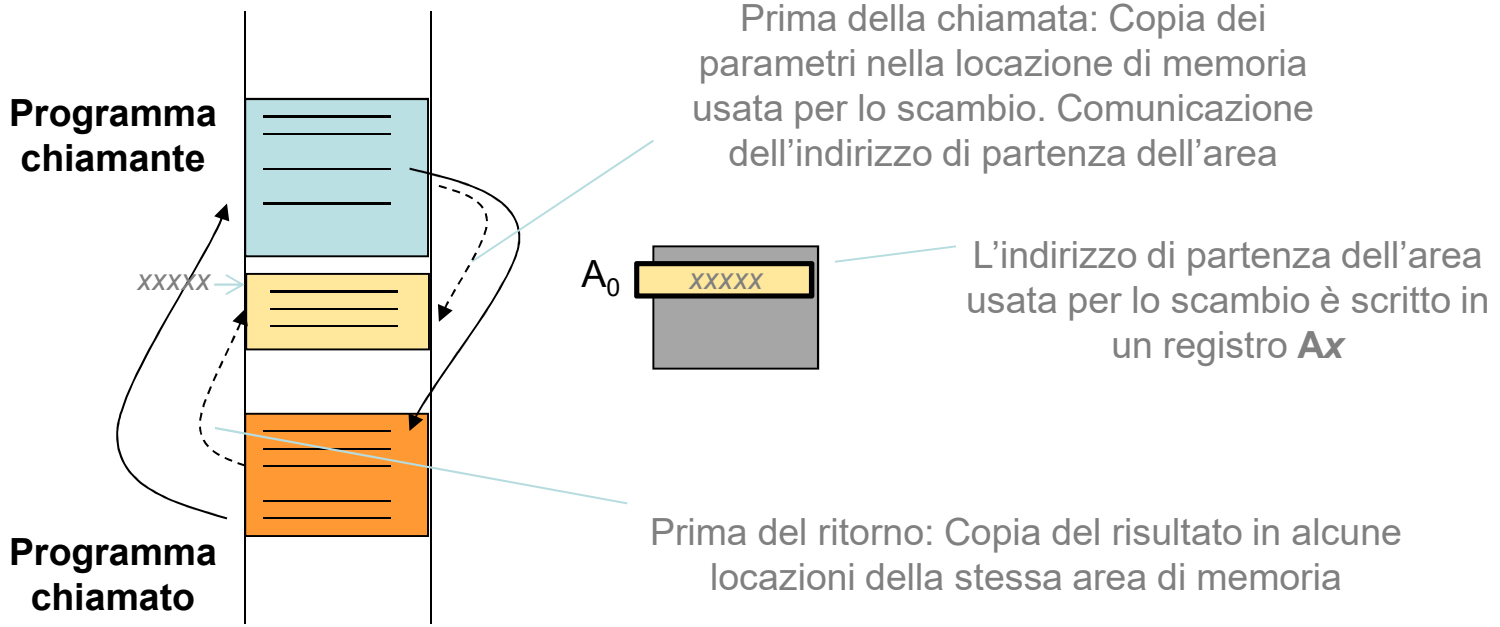


Passaggio dei parametri su memoria

- Invece che basarsi sul ridotto numero di registri interni al processore per lo scambio di parametri, programma chiamante e sottoprogramma possono condividere un'**area di memoria**
  - Lo scambio di parametri consiste quindi nella scrittura dei dati all'interno dell'area di memoria e nel passaggio del solo indirizzo di partenza dell'area di memoria
  - deve essere comunque stabilita una convenzione sull'ordine con cui i dati sono disposti all'interno dell'area di memoria
- Si deve assumere che l'area di memoria sia sempre disponibile, poiché la chiamata può avvenire in qualsiasi momento

Passaggio dei parametri su memoria

- L'indirizzo di partenza dell'area di memoria può essere comunicato tramite un registro indirizzo (ad esempio un registro **A*x*** del M68000)
- Ciò permette di usare di volta in volta aree fisiche diverse



Passaggio dei parametri su memoria

Esempio:

- L'area di scambio (**AREA**) dati è posta all'indirizzo $6000
- Il programma chiamante (**MAIN**) ne pone l'indirizzo in **A0**
- Il sottoprogramma (**SUBR**) legge l'indirizzo e lo usa per accedere ai dati

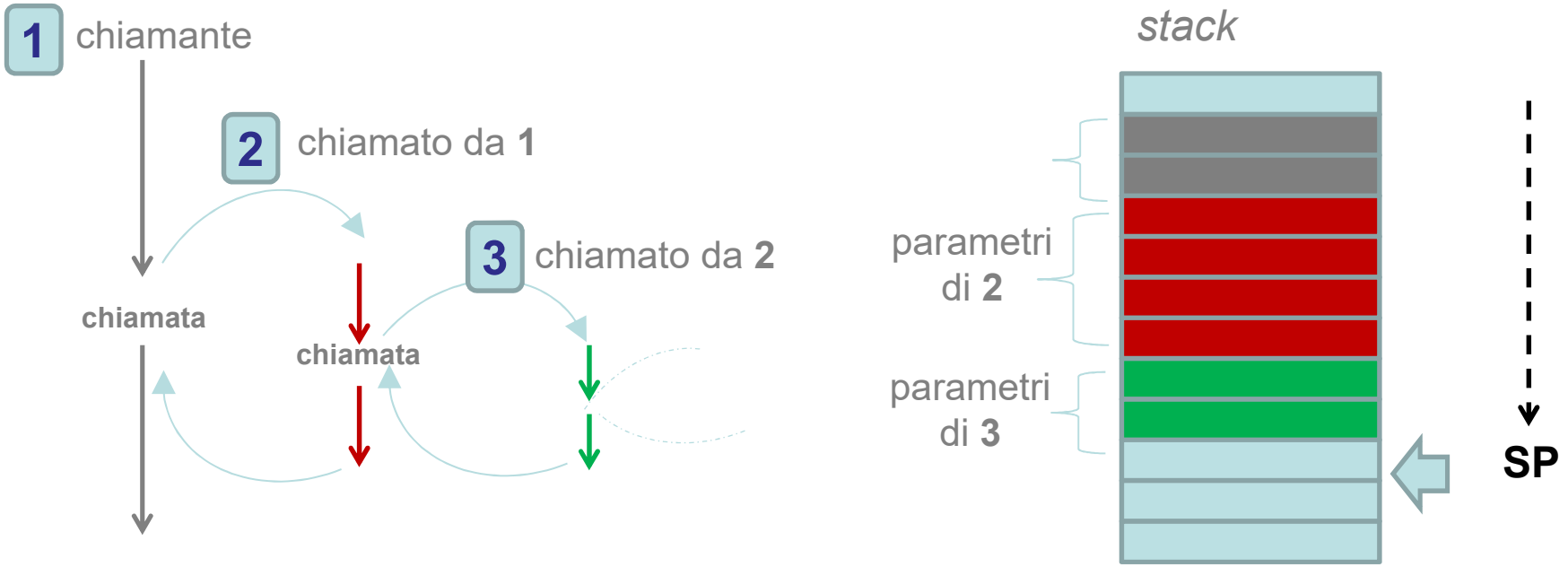
```
A           EQU         0
B           EQU         4
C           EQU         6

            ORG         $6000
AREA        DS.L        1
            DS.W        1
            DS.W        1

            ORG         $8000
MAIN    . . .
            MOVEA.L #AREA,A0
            . . . scrive i parametri
            . . . effettivi nell'area
            JSR             SUBR
        . . .
SUBR        MOVE.L  A(A0),D0
            MOVE.W  B(A0),D1
            MOVE.W  C(A0),D2
```

Passaggio di parametri tramite Stack

- Lo ***stack*** fornisce un meccanismo ideale per gestire aree di memoria durante lo scambio di parametri
- La disciplina **Last in First Out**, LIFO (l'ultimo elemento inserito sarà il primo letto) è ideale per memorizzare parametri e indirizzo di ritorno, in particolare per consentire le chiamate *annidate* a sottoprogrammi:
  - situazione in cui un sottoprogramma a sua volta chiama altri sottoprogrammi (eventualmente sé stesso, in quelle che si chiamano *chiamate ricorsive*)

Stack per le chiamate annidate

- La funzione che termina per prima (ad esempio **3**) è l'ultima che aveva usato lo stack tramite delle operazioni di ***push*** per memorizzare l'indirizzo di ritorno e i parametri (oltre che, eventualmente, variabili locali)

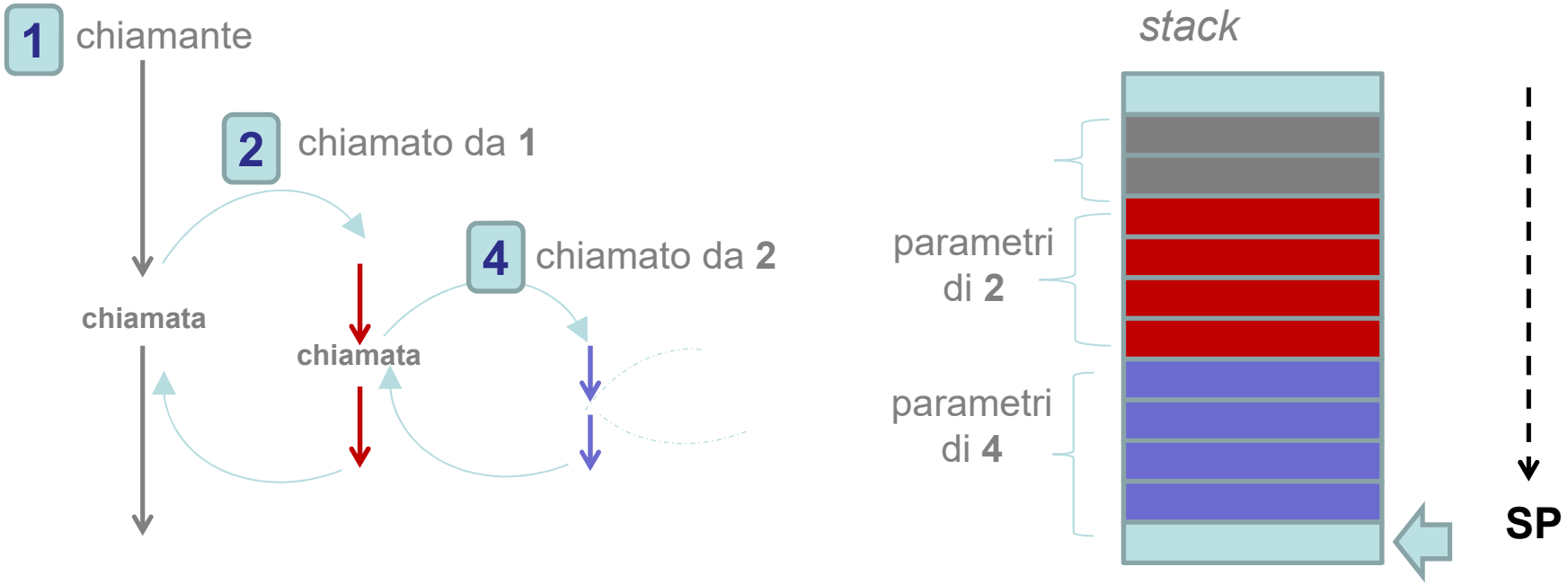


Stack per le chiamate annidate

- Appena il sottoprogramma **3** termina, *ripulirà* lo stack, tramite delle operazioni di ***pop*** riportandolo alla condizione creata dal sottoprogramma **2**.

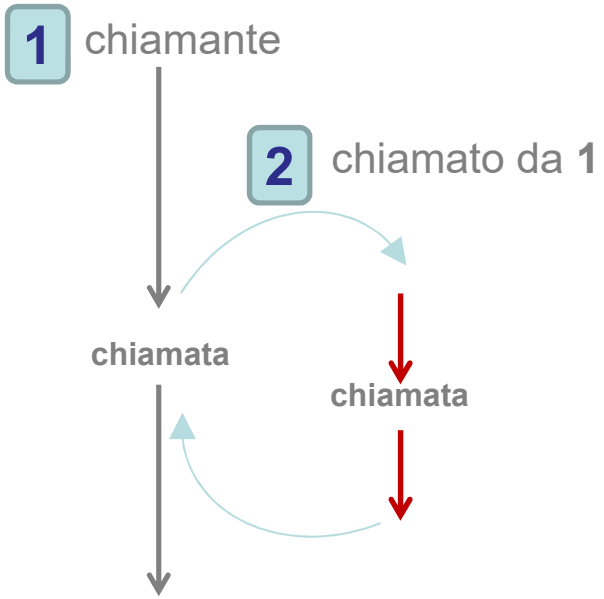
Stack per le chiamate annidate

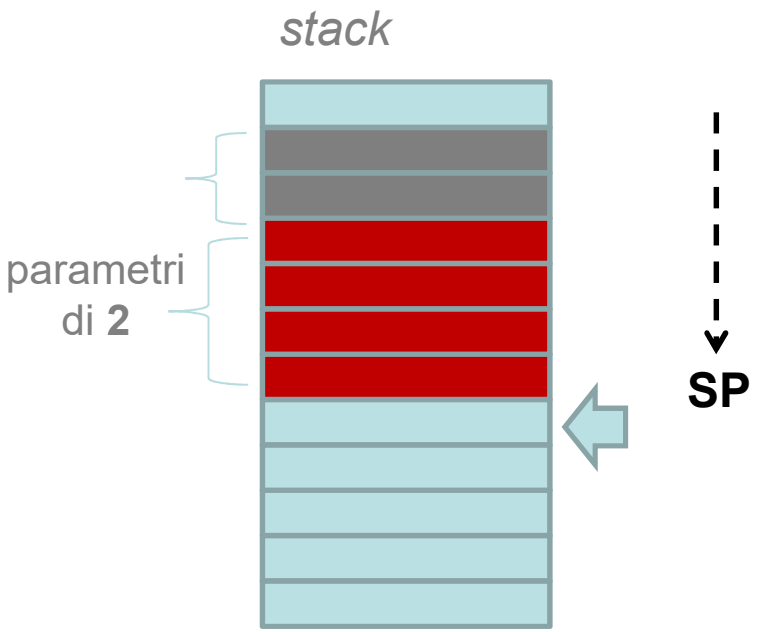
- Successivamente, può essere chiamato un differente sottoprogramma (ad esempio, **4**). *Linkage* e *scambio di parametri* avvengono indipendentemente dallo stato degli altri sottoprogrammi e dalle specifiche locazioni in cui sono memorizzati



Stack per le chiamate annidate

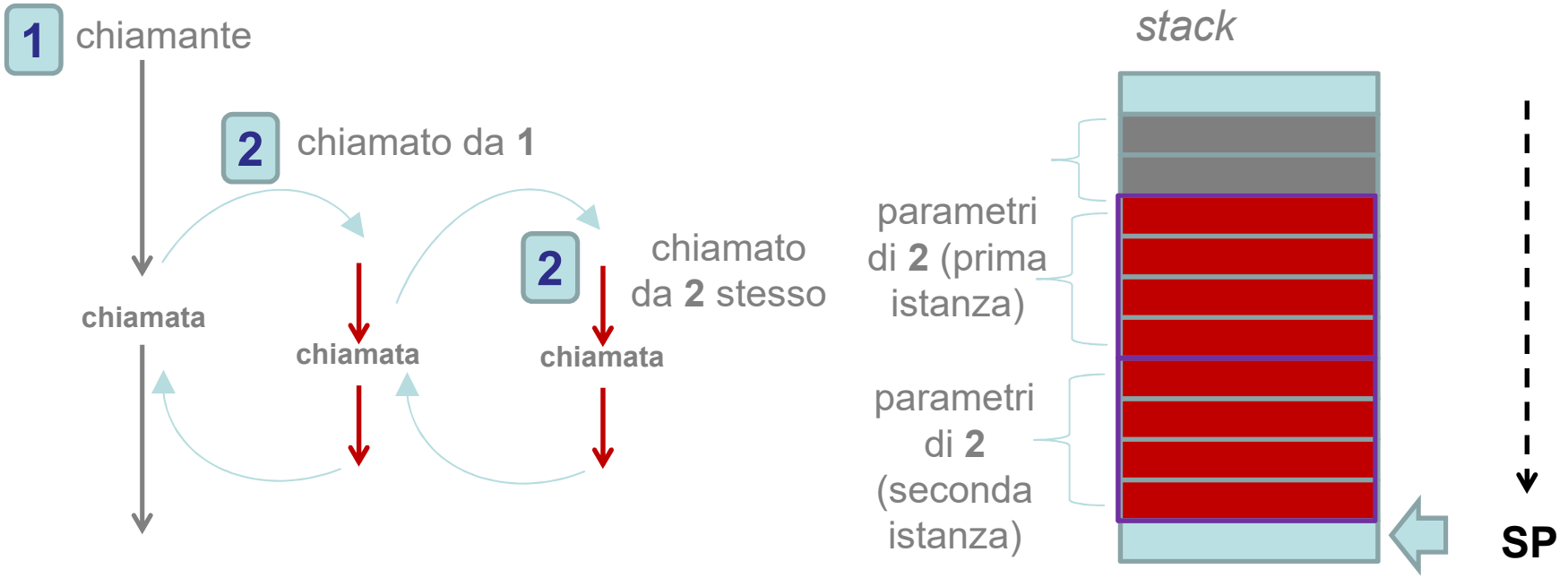
- Ciascun “collegamento” tra programma chiamante e sottoprogramma avviene esclusivamente tramite operazione di operazioni di ***push*** di indirizzi di ritorno e parametri nello stack e prelievi tramite operazioni di ***pop***, indipendentemente dalla posizione nello stack





Stack per le chiamate ricorsive

- E' possibile che una funzione chiami *sé stessa*: Le istruzioni della funzione sono le stesse, ma le locazioni occupate nello stack da indirizzo di ritorno e parametri sono diverse per ciascuna chiamata (ovvero, per ciascuna ***istanza*** della funzione)



Esempio: passaggio di parametri tramite stack

File: `programma021.a68`

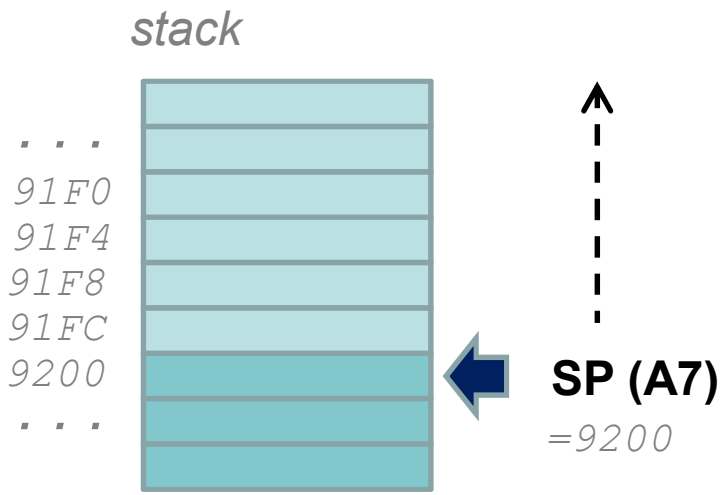
- Scrivere una subroutine che prenda due parametri Long, ***b*** ed ***e***, e calcoli b^e
- Parametri e risultato devono essere passati tramite *stack*
- Scrivere un esempio di programma che chiama la subroutine scritta

Esempio: passaggio di parametri tramite stack

File: `programma021.a68`

```
        ORG     $8000
MAIN    MOVE.L  #2,-(A7)
        MOVE.L  #4,-(A7)
        JSR     SUBR
        MOVE.L  (A7)+,RES
        ADD     #4,A7
        ORG $8100
RES     DS.L    1

        ORG $8140
SUBR    MOVE.L  4(A7),D0
        MOVE.L  8(A7),D1
        MOVE.L  #1,D2
        SUBQ.L  #1,D0
LOOP    MULU    D1,D2
        DBRA    D0,LOOP
        MOVE.L  D2,4(A7)
        RTS

        END     MAIN
```

Il programma chiamante passa i valori immediati **2** e **4** al sottoprogramma in modo da calcolare 2^4.
Supponiamo che lo stack sia inizialmente alla locazione **9200**. Lo stack pointer **SP** (ovvero **A7**) punta all'*ultima locazione occupata*. Lo stack è a *decrescere*.

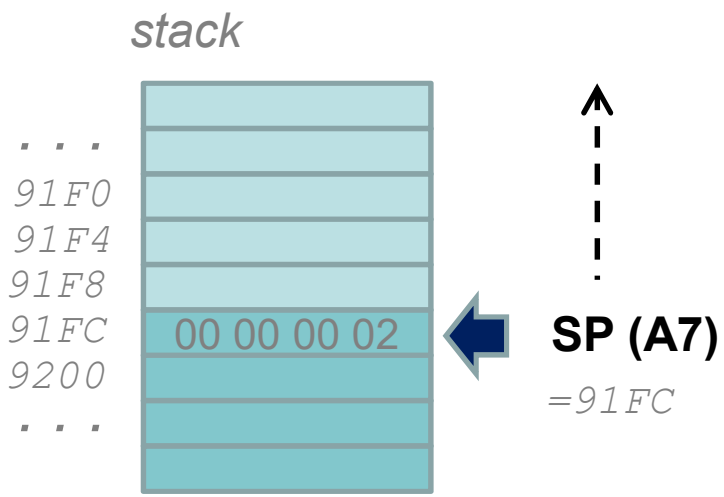


Esempio: passaggio di parametri tramite stack

File: `programma021.a68`

```
        ORG     $8000
MAIN    MOVE.L  #2,-(A7)
        MOVE.L  #4,-(A7)
        JSR     SUBR
        MOVE.L  (A7)+,RES
        ADD     #4,A7
        ORG $8100
RES     DS.L    1

        ORG $8140
SUBR    MOVE.L  4(A7),D0
        MOVE.L  8(A7),D1
        MOVE.L  #1,D2
        SUBQ.L  #1,D0
LOOP    MULU    D1,D2
        DBRA    D0,LOOP
        MOVE.L  D2,4(A7)
        RTS

        END     MAIN
```

La prima **MOVE** rappresenta un **push** dell'immediato **#2** nello stack (la base). Lo Stack Pointer viene prima decrementato, poi si scrive in memoria il valore immediato **#2** (visto come Long, quindi su 4 byte)

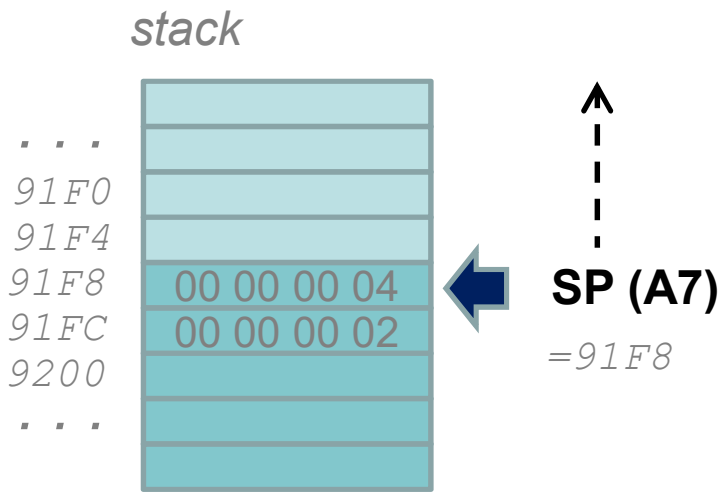


Esempio: passaggio di parametri tramite stack

File: `programma021.a68`

```
        ORG     $8000
MAIN    MOVE.L  #2,-(A7)
        MOVE.L  #4,-(A7)
        JSR     SUBR
        MOVE.L  (A7)+,RES
        ADD     #4,A7
        ORG $8100
RES     DS.L    1

        ORG $8140
SUBR    MOVE.L  4(A7),D0
        MOVE.L  8(A7),D1
        MOVE.L  #1,D2
        SUBQ.L  #1,D0
LOOP    MULU    D1,D2
        DBRA    D0,LOOP
        MOVE.L  D2,4(A7)
        RTS

        END     MAIN
```

La seconda **MOVE** fa il push dell'immediato **#4** (l'esponente) sullo stack



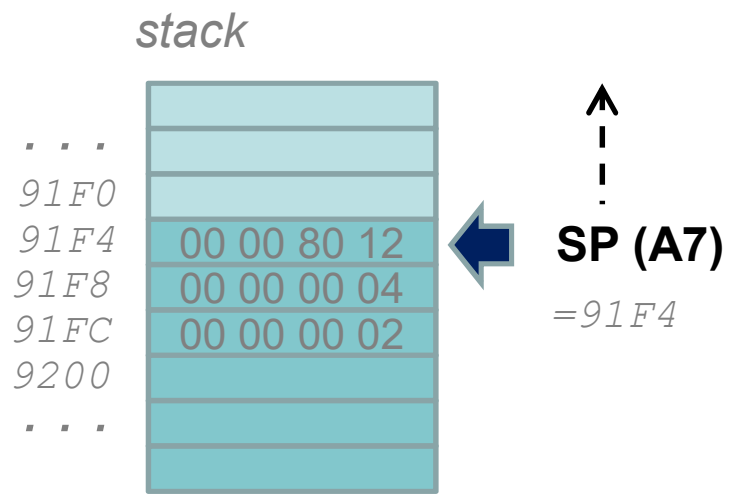
Esempio: passaggio di parametri tramite stack

File: `programma021.a68`

```
        ORG     $8000
MAIN    MOVE.L  #2,-(A7)
        MOVE.L  #4,-(A7)
        JSR     SUBR
        MOVE.L  (A7)+,RES
        ADD     #4,A7
        ORG $8100
RES     DS.L    1

        ORG $8140
SUBR    MOVE.L  4(A7),D0
        MOVE.L  8(A7),D1
        MOVE.L  #1,D2
        SUBQ.L  #1,D0
LOOP    MULU    D1,D2
        DBRA    D0,LOOP
        MOVE.L  D2,4(A7)
        RTS

        END     MAIN
```

La **JSR** compie due azioni:

- push del **PC** nello stack. Viene cioè salvato nello stack (per poi essere usato dopo alla fine del sottoprogramma) l'indirizzo di ritorno, ovvero l'indirizzo della prima istruzione che segue la **JSR** (la terza **MOVE** nell'esempio)
- Aggiornamento del **PC** in modo da spostare l'esecuzione a **SUBR**

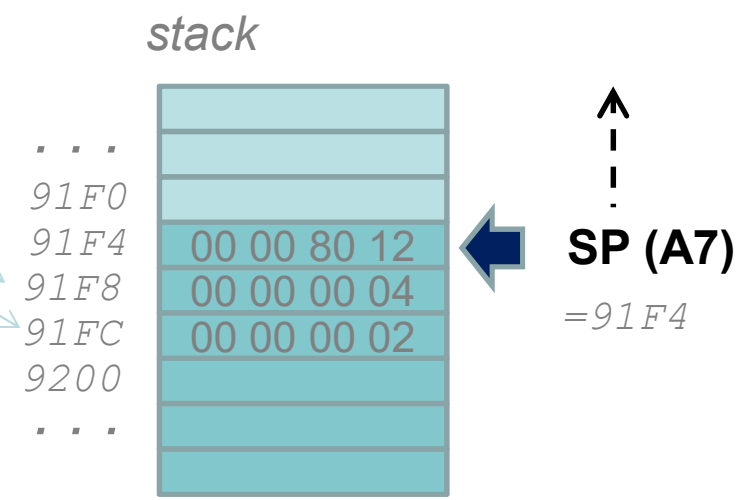


Esempio: passaggio di parametri tramite stack

File: `programma021.a68`

```
        ORG     $8000
MAIN    MOVE.L  #2,-(A7)
        MOVE.L  #4,-(A7)
        JSR     SUBR
        MOVE.L  (A7)+,RES
        ADD     #4,A7
        ORG $8100
RES     DS.L    1

        ORG $8140
SUBR    MOVE.L  4(A7),D0
        MOVE.L  8(A7),D1
        MOVE.L  #1,D2
        SUBQ.L  #1,D0
LOOP    MULU    D1,D2
        DBRA    D0,LOOP
        MOVE.L  D2,4(A7)
        RTS

        END     MAIN
```

La subroutine effettua due letture usando opportuni spiazzamenti a partire dal valore corrente dello Stack Pointer (**A7**) in modo da copiare i due parametri. Queste due letture non cambiano lo Stack Pointer e *non costituiscono quindi delle pop*

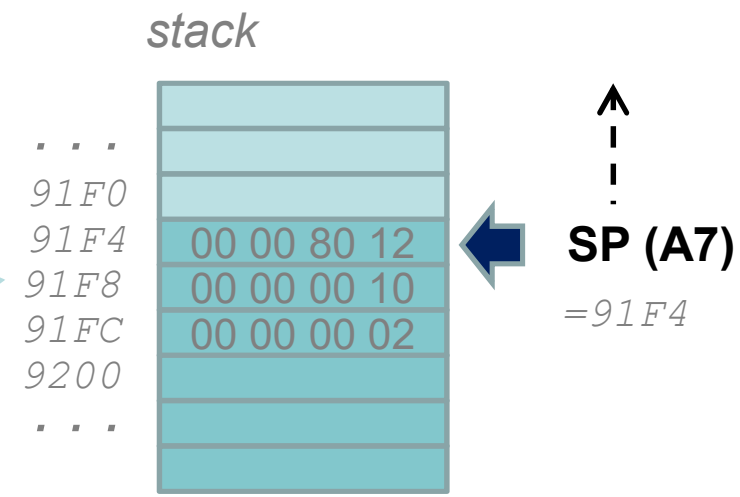


Esempio: passaggio di parametri tramite stack

File: `programma021.a68`

```
        ORG     $8000
MAIN    MOVE.L  #2,-(A7)
        MOVE.L  #4,-(A7)
        JSR     SUBR
        MOVE.L  (A7)+,RES
        ADD     #4,A7
        ORG $8100
RES     DS.L    1

        ORG $8140
SUBR    MOVE.L  4(A7),D0
        MOVE.L  8(A7),D1
        MOVE.L  #1,D2
        SUBQ.L  #1,D0
LOOP    MULU    D1,D2
        DBRA    D0,LOOP
        MOVE.L  D2,4(A7)
        RTS

        END     MAIN
```

Al termine dell'esecuzione, la subroutine copia il risultato ($2^4=16=10_{16}$ presente nel registro **D2**) nello stack, precisamente nella seconda posizione, subito sotto l'indirizzo di ritorno. Questa scrittura non cambia lo Stack Pointer e pertanto *non è una push*.

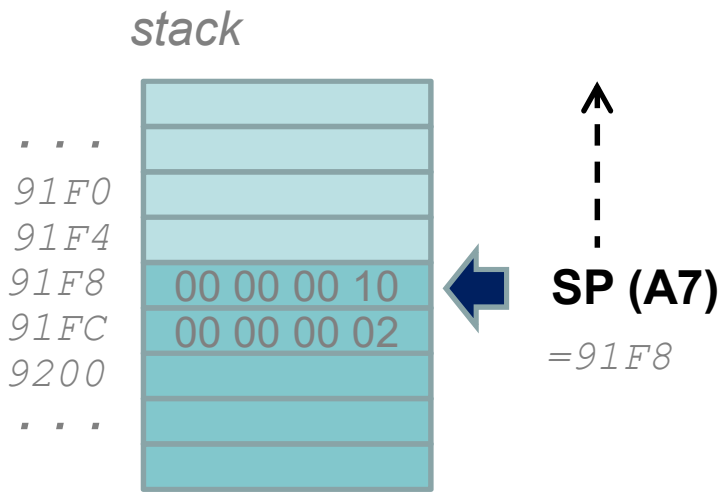


Esempio: passaggio di parametri tramite stack

File: `programma021.a68`

```
        ORG     $8000
MAIN    MOVE.L  #2,-(A7)
        MOVE.L  #4,-(A7)
        JSR     SUBR
        MOVE.L  (A7)+,RES
        ADD     #4,A7
        ORG $8100
RES     DS.L    1

        ORG $8140
SUBR    MOVE.L  4(A7),D0
        MOVE.L  8(A7),D1
        MOVE.L  #1,D2
        SUBQ.L  #1,D0
LOOP    MULU    D1,D2
        DBRA    D0,LOOP
        MOVE.L  D2,4(A7)
        RTS

        END     MAIN
```

La **RTS** effettua un pop dallo stack, ripristinando il vecchio valore del **PC**, aggiornando lo Stack Pointer **A7**, e determinando un salto indietro al programma chiamante

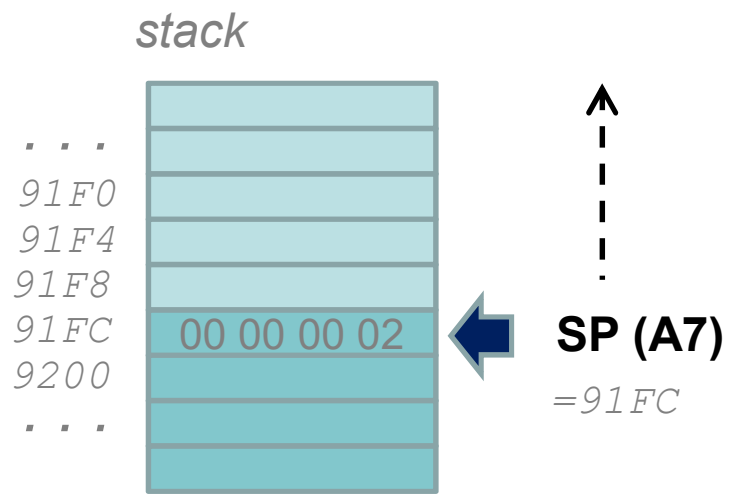


Esempio: passaggio di parametri tramite stack

File: `programma021.a68`

```
        ORG     $8000
MAIN    MOVE.L  #2,-(A7)
        MOVE.L  #4,-(A7)
        JSR     SUBR
        MOVE.L  (A7)+,RES
        ADD     #4,A7
        ORG $8100
RES     DS.L    1

        ORG $8140
SUBR    MOVE.L  4(A7),D0
        MOVE.L  8(A7),D1
        MOVE.L  #1,D2
        SUBQ.L  #1,D0
LOOP    MULU    D1,D2
        DBRA    D0,LOOP
        MOVE.L  D2,4(A7)
        RTS

        END     MAIN
```

Il programma chiamante fa un **pop** per recuperare il risultato e copiarlo nella locazione **RES**

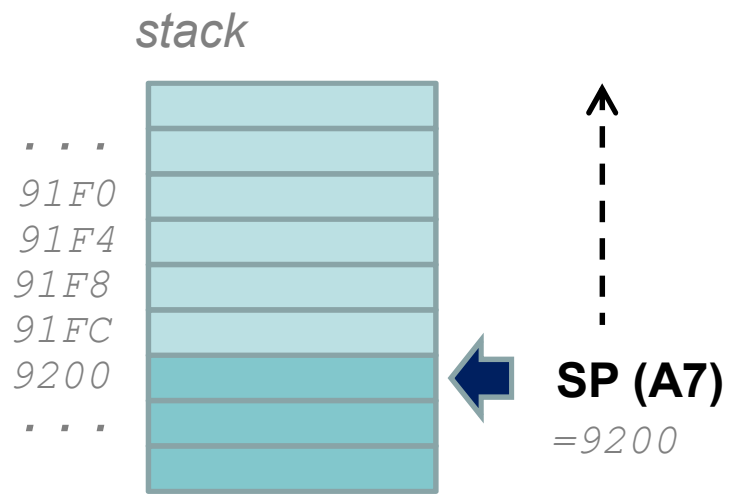


Esempio: passaggio di parametri tramite stack

File: `programma021.a68`

```
        ORG     $8000
MAIN    MOVE.L  #2,-(A7)
        MOVE.L  #4,-(A7)
        JSR     SUBR
        MOVE.L  (A7)+,RES
        ADD     #4,A7
        ORG $8100
RES     DS.L    1

        ORG $8140
SUBR    MOVE.L  4(A7),D0
        MOVE.L  8(A7),D1
        MOVE.L  #1,D2
        SUBQ.L  #1,D0
LOOP    MULU    D1,D2
        DBRA    D0,LOOP
        MOVE.L  D2,4(A7)
        RTS

        END     MAIN
```

Infine, con un opportuno aggiornamento del registro **A7** (incrementato di **4**), lo stack viene riportato esattamente nella stessa condizione in cui si trovava prima della chiamata.



Esempio: ricorsione

File: `programma022.a68`

- Scrivere un sottprogramma programma che calcoli il fattoriale di un intero tramite chiamate ricorsive
- Invocare il sottoprogramma in un programma principale di esempio.

Esempio: ricorsione

File: `programma022.a68`

- In un linguaggio di alto livello, il programma potrebbe essere sviluppato come segue:

```
int fact(int n){
        if (n==1){
                return 1;              // caso base
        }else{
                return n*fact(n-1); // chiamata ricorsiva
        }
}
```

- *sulla base della proprietà:* $n! = n*(n-1)!$

Esempio: ricorsione

File: `programma022.a68`

```
        ORG     $8000
MAIN    MOVE.L  #5,-(A7)
        JSR     FACT
        MOVE.L  (A7)+,RES
FINE    JMP     FINE

        ORG $8100
RES     DS.L    1
```

```
        ORG $8140
FACT    MOVEM.L D0-D1,-(A7)
        MOVE.L  12(A7),D0
        CMP.L   #1,D0
        BNE     SKIP
        MOVEM.L (A7)+,D0-D1
        RTS
SKIP    MOVE.L  D0,D1
        SUB.L   #1,D1
        MOVE.L  D1,-(A7)
        JSR     FACT
        MOVE.L  (A7)+,D1
        MULU    D1,D0
        MOVE.L  D0,12(A7)
        MOVEM.L (A7)+,D0-D1
        RTS
        END     MAIN
```

Sottoprogramma ricorsivo

Esempio: ricorsione

File: `programma022.a68`

```
        ORG     $8000
MAIN    MOVE.L  #5,-(A7)
        JSR     FACT
        MOVE.L  (A7)+,RES
FINE    JMP     FINE

        ORG $8100
RES     DS.L    1
```

- ***Push*** di un parametro (il valore immediato **#5**) su cui la subroutine calcolerà il fattoriale
- Chiamata alla subroutine **FACT**
- ***Pop*** del risultato che il chiamante si aspetta di trovare nello stack alla stessa posizione in cui ha posto il parametro di ingresso

Esempio: ricorsione

File: `programma022.a68`

Salva nello stack con un **push** i registri che saranno "sporcati"

Legge il parametro di ingresso **n** in **D0**

Verifica se il parametro di ingresso è **1**. Nel caso, il risultato coincide con l'ingresso e semplicemente ritorna.

Altrimenti, decrementa in **D1** il parametro di ingresso **n**, e chiama la funzione **FACT** stessa, passandole quindi come parametro il valore **n-1**

Preleva il risultato, ovvero il fattoriale di **n-1**, lo moltiplica per **n**, ancora contenuto in **D0**, e pone il risultato nello stack, dove si aspetterà di trovarlo il chiamante

Ripristina i vecchi valori dei registri

```
        ORG   $8140
FACT    MOVEM.L D0-D1,-(A7)
        MOVE.L  12(A7),D0
        CMP.L   #1,D0
        BNE     SKIP
        MOVEM.L (A7)+,D0-D1
        RTS
SKIP    MOVE.L  D0,D1
        SUB.L   #1,D1
        MOVE.L  D1,-(A7)
        JSR     FACT
        MOVE.L  (A7)+,D1
        MULU    D1,D0
        MOVE.L  D0,12(A7)
        MOVEM.L (A7)+,D0-D1
        RTS
        END     MAIN
```

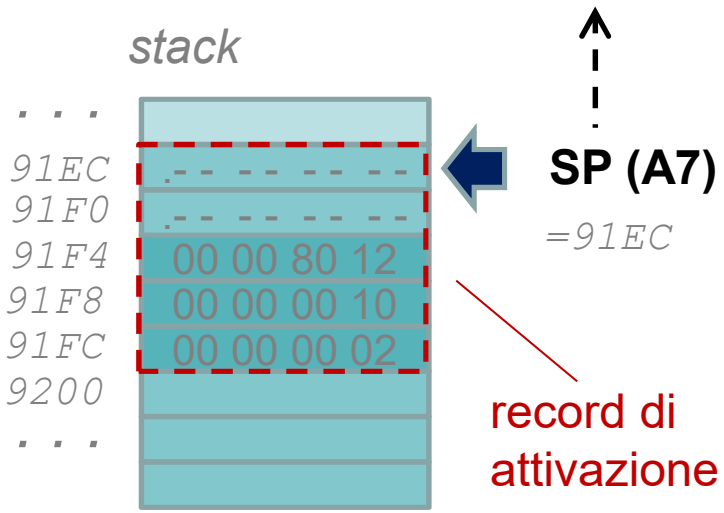
Record di attivazione nello Stack

- Oltre all'indirizzo di ritorno ed ai parametri scambiati tra programma chiamante e sottoprogramma, tipicamente nello stack si salvano anche le **variabili locali** del sottoprogramma
  - ad esempio, in un'area dello stack immediatamente successiva ai parametri
  - L'intera struttura composta da indirizzo di ritorno, parametri, locazioni per ospitare il valore di ritorno ed area delle variabili locali viene spesso indicata con il nome di ***Stack Frame***, o ***Record di Attivazione***

Record di attivazione nello Stack

- Se la subroutine ha bisogno di gestire **variabili locali**, le può *allocare* nello stack
- Basta decrementare lo Stack Pointer della quantità di byte necessaria
- L'effetto è quello della direttiva **DS**, ma l'allocazione è dinamica:
  - effettuata al momento della chiamata
  - *deallocazione* al ritorno

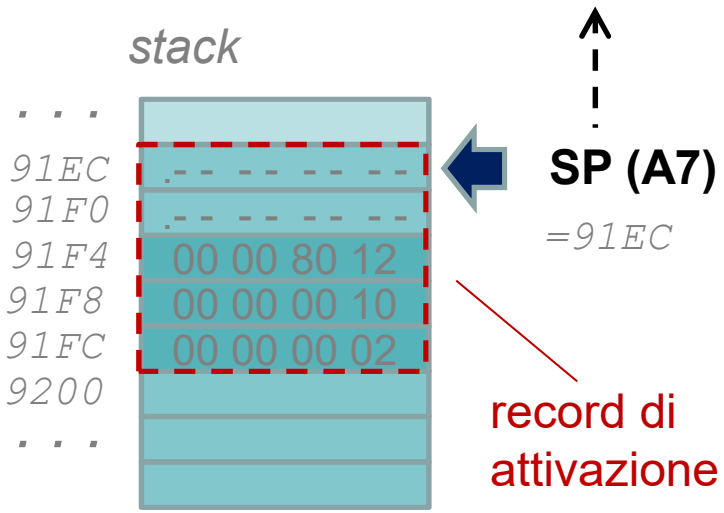
Esempio di record di attivazione:
In questo esempio, la subroutine ha necessità di due variabili locali Long (**8** byte in tutto). Sottrae quindi **8** allo Stack Pointer riservando spazio sullo stack. Le variabili saranno accessibili agli indirizzi **91F0** e **91EC**



Record di attivazione nello Stack

- Nel momento della chiamata, la subroutine può accedere allo Stack Frame per differenza rispetto allo Stack Pointer mediante opportuni spiazzamenti
  - (vedi esempio a lato)
- Tuttavia, durante il suo funzionamento, lo Stack Pointer può variare…

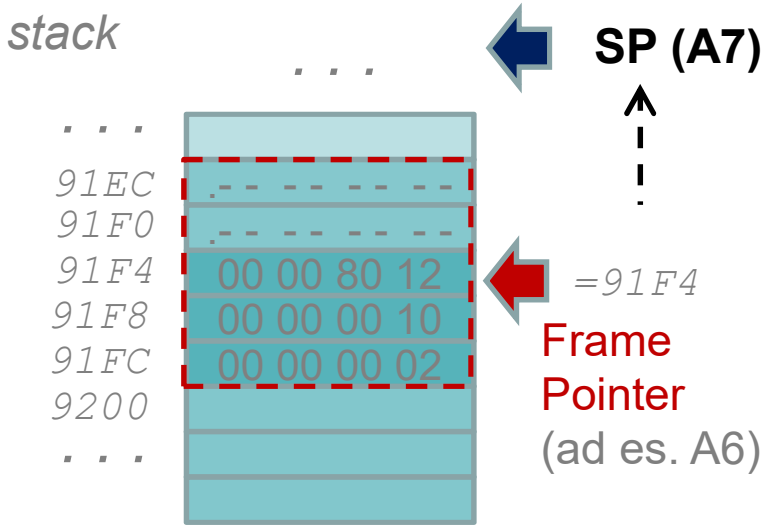
- La prima variabile locale è all'indirizzo **`4(A7)`**
- La seconda variabile locale è all'indirizzo **`0(A7)`**
- Il primo parametro è all'indirizzo **`16(A7)`**
- Il secondo paramentroè all'indirizzo **`12(A7)`**



Record di attivazione nello Stack

- La subroutine potrà fare ulteriori push nello stack, ad esempio per chiamare altre subroutine
- E' quindi spesso opportuno, al momento della chiamata, salvare l'indirizzo di partenza dello Stack Frame, ad esempio in un registro
  - L'indirizzo salvato **rimarrà fisso** durante il funzionamento della subroutine
  - L'indirizzo salvato è detto **Frame Pointer**

Anche variando lo Stack Pointer durante il suo funzionamento, la subroutine potrà far riferimento in qualsiasi momento al contenuto dello stack frame mediante opportuni spiazzamenti rispetto al **Frame Pointer**, tipicamente salvato in un registro indirizzo (ad es. **A6**)



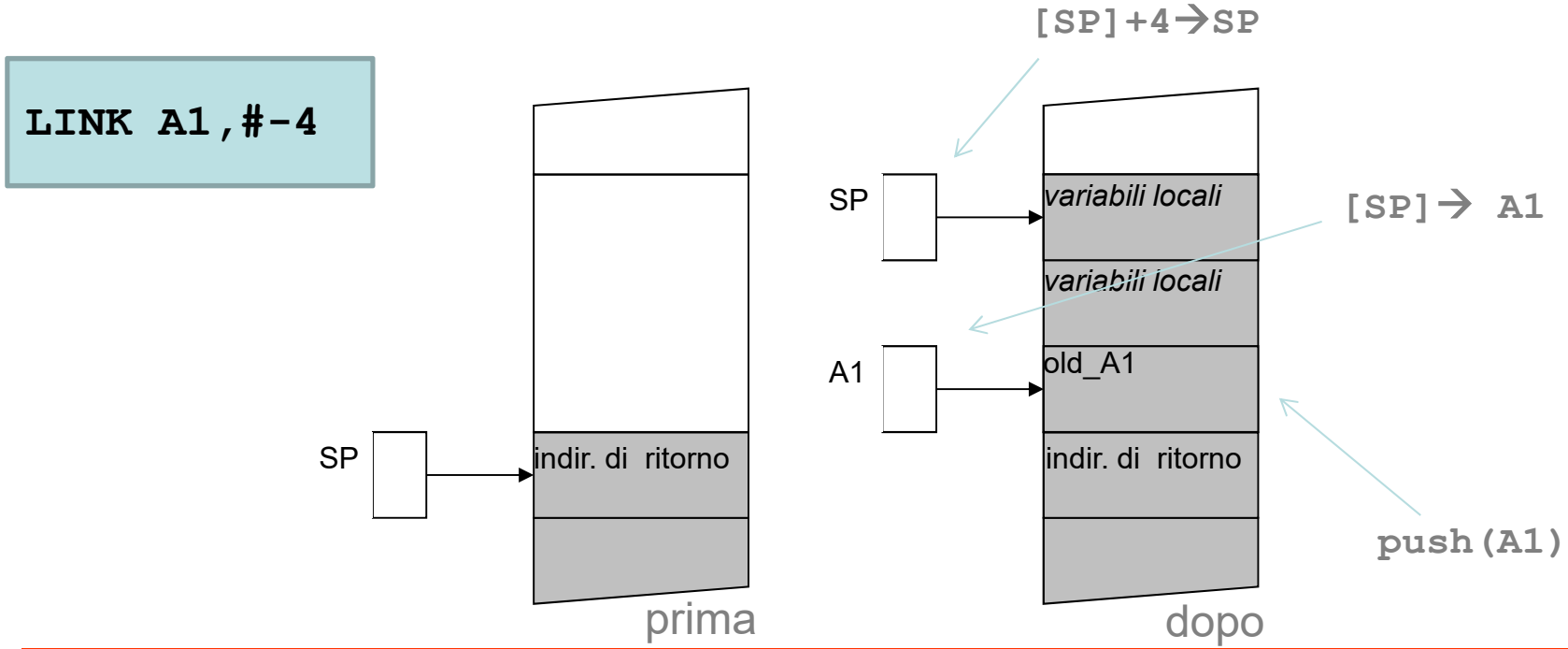
Istruzione LINK del 68000

- Per facilitare la gestione del record di attivazione sullo stack, il processore Motorola 68000 dispone di un'istruzione specifica: **`LINK Ax, #im`**

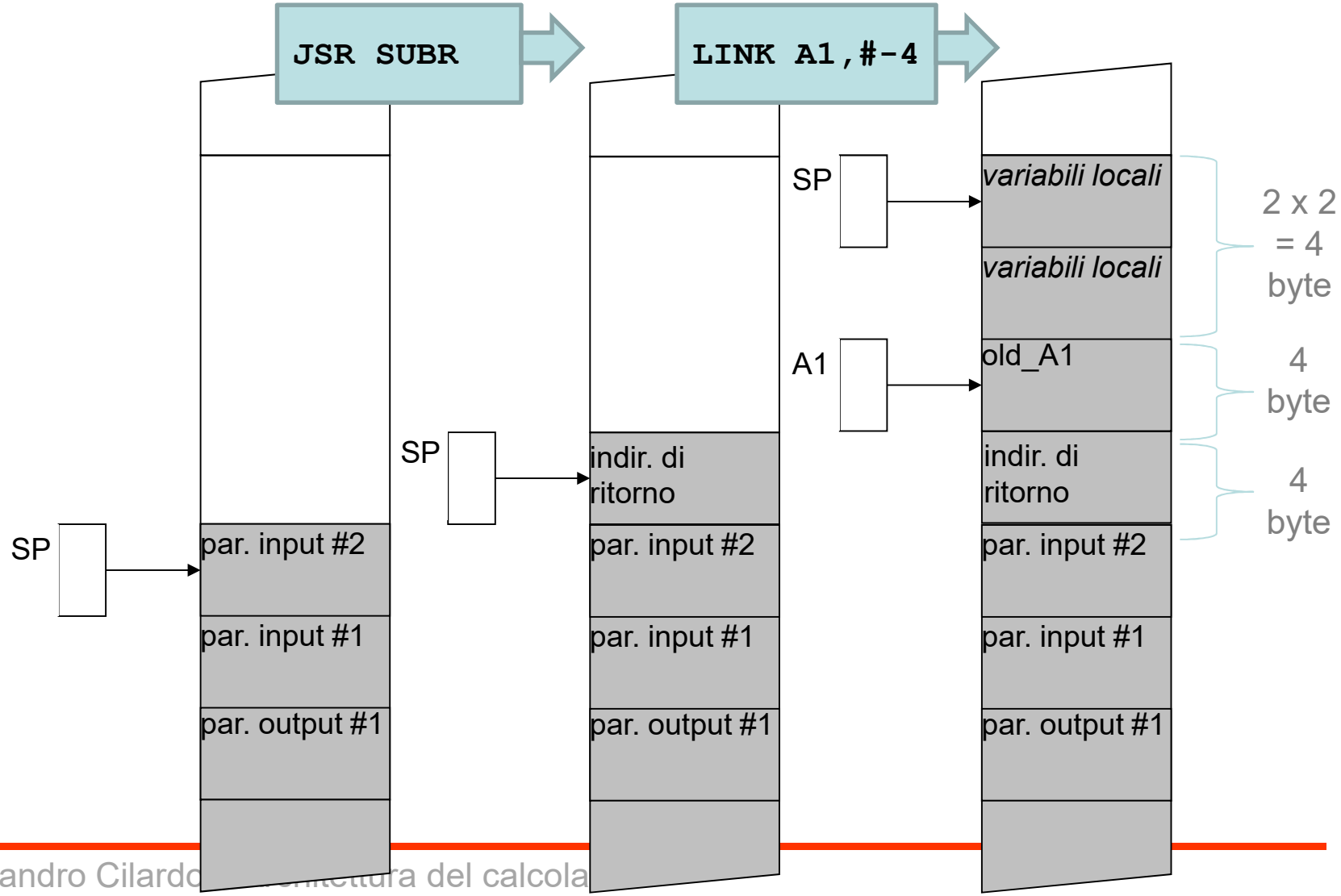
  Effetto: **`push(Ax)`**; **`[SP]→Ax`** ; **`[SP]+im→SP`**;
- salva il contenuto del registro **`Ax`** sullo stack, carica in **`Ax`** il valore aggiornato dello Stack Pointer, ed infine incrementa **`SP`** (**`A7`**) dell'offset **`im`**
- Sommando il valore negativo **`im`** al contenuto di **`SP`**, l'istruzione **`LINK`** riserva un'area di memoria di **`im`** byte sulla cima dello stack. Quest'area è utilizzata per l'allocazione delle variabili locali del sottoprogramma.
- Dualmente: **`UNLK Ax`** effettua: **`[Ax]→SP`**; **`pop(Ax)`**;

LINK e UNLK

- Stato dello stack prima e dopo **`LINK`**
- Supponiamo che la routine richieda due variabili locali, entrambe *Word* (**4** byte in tutto)

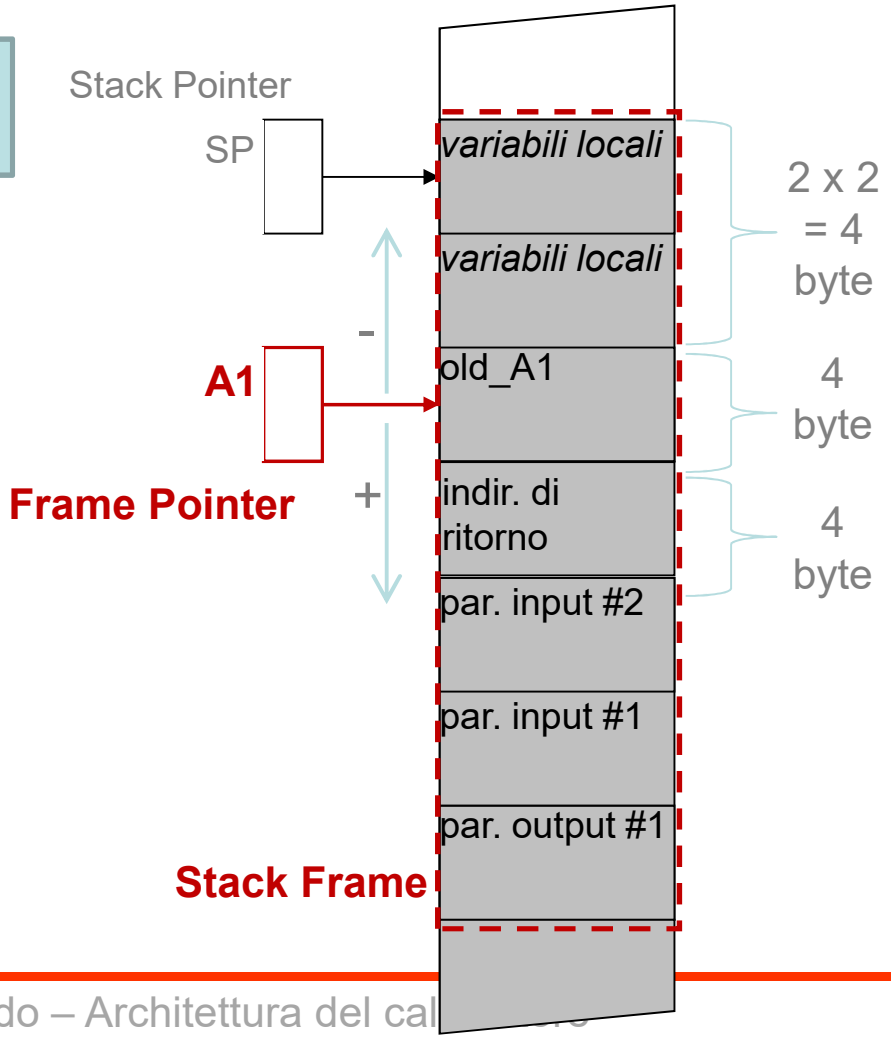


LINK e UNLK


JSR SUBR
LINK A1,#-4
SP
par. input #2
par. input #1
par. output #1
SP
indir. di ritorno
par. input #2
par. input #1
par. output #1
SP
A1
variabili locali
variabili locali
old_A1
indir. di ritorno
par. input #2
par. input #1
par. output #1
2 x 2 = 4 byte
4 byte
4 byte

LINK e UNLK

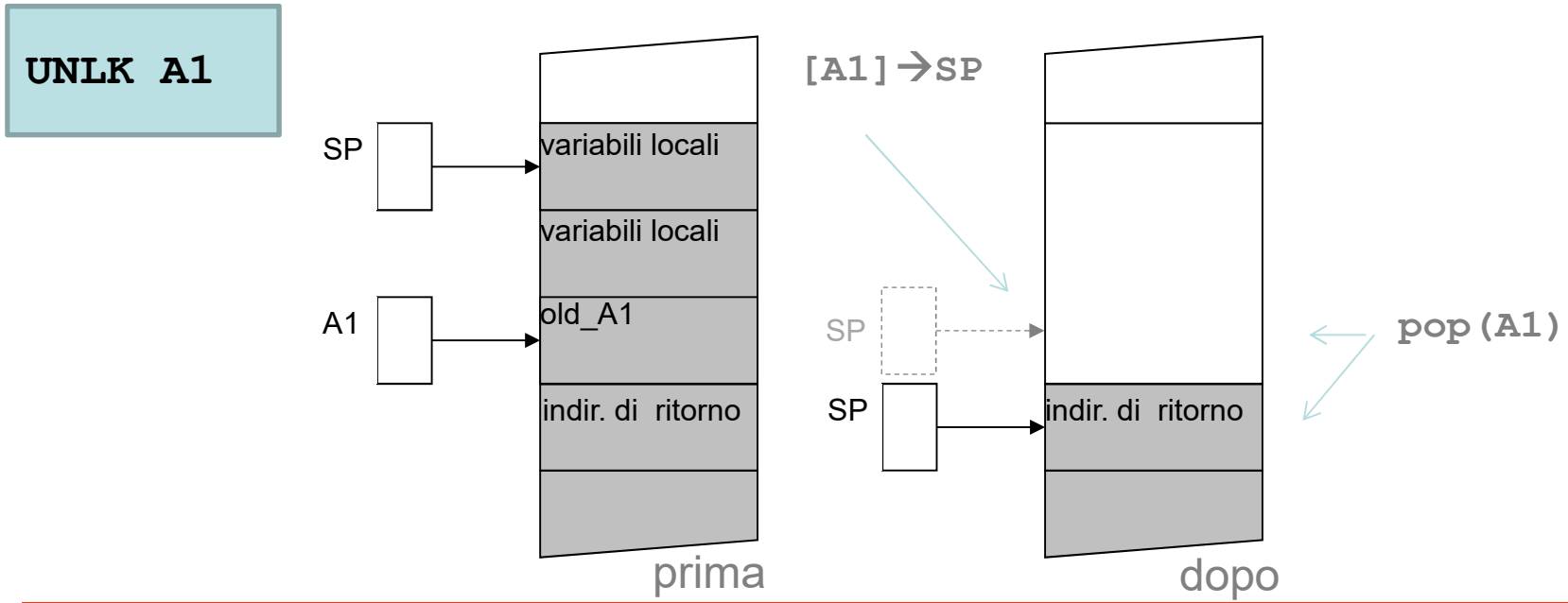
```
LINK A1,#-4
```



Parametri e variabili locali sono accessibili tramite *spiazzamenti rispeto al* Frame Pointer, **positivi** per i parametri e **negativi** per le variabili locali

LINK e UNLK

- Stato dello stack prima e dopo **`UNLK`**
- L'istruzione **`UNLK Ax`** ripristina in **`Ax`** il valore puntato da **`Ax`** (old_**`Ax`**) e



Esempio con LINK/UNLK

File: programma023.a68

```
* Esempio di chiamata a sottoprogramma con LINK/UNLK
* Calcola RIS = BASE^ESP mediante sottoprogramma POWR
* Programma principale
        ORG     $8000
MAIN    ADDA.L  #-2,SP          riserva 2 byte per RIS
        MOVE    ESP,-(SP)       push dell'esponente
        MOVE    BASE,-(SP)      push della base
        JSR     POWR            chiama la subroutine POWR
        ADDQ    #4,SP           rimuovi BASE e ESP dallo stack
        MOVE    (SP)+,RIS       copia parametro di output in RIS

FINE    JMP     FINE            ferma qui l'esecuzione del programma

* Allocazione variabili X, Y e Z
BASE    DC.W      3             base
ESP     DC.W      4             esponente
RIS     DS.W      1             risultato
```

Esempio con LINK/UNLK

File: programma023.a68

```
BASE_OFF        EQU     8
ESP_OFF         EQU     10
RIS_OFF         EQU     12
* Subroutine POWR
POWR    LINK    A6,#0                   usa A6 come frame pointer
        MOVEM.L D0-D2,-(SP)             salva i registri sullo stack
        MOVE    BASE_OFF(A6),D0         base in D0
        MOVE    ESP_OFF(A6),D1          esponente in D1
        MOVE.L  #1,D2                   usa D2 come accumulatore
LOOP    SUBQ    #1,D1           decrementa l'esp. (fa da contatore)
        BMI.S   EXIT            se D1 è diventato negativo allora
                                salta ad EXIT (con offset Short)
        MULS    D0,D2           moltiplica D2 per A
        BRA     LOOP            e ripeti se necessario
EXIT    MOVE    D2,RIS_OFF(A6)  risultato in D2
        MOVEM.L (SP)+,D0-D2     ripristina i registri salvati prima
        UNLK    A6
        RTS
        END     MAIN
```