Programmazione Assembly

SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Pellegrini

Calcolatori Elettronici
Sapienza, Università di Roma

A.A. 2012/2013

Scheletro di un programma assembly

```
1  ORG [INDIRIZZO CARICAMENTO]
2
3  ; Dichiarazione costanti e variabili globali
4
5  CODE
6
7  ; Corpo del programma
8
9  label:
10
11     halt ; Per arrestare l'esecuzione
12 END
```

Scheletro di un programma assembly (2)

- ORG permette di caricare il programma ad uno specifico indirizzo di memoria (la prima parte della memoria è riservata)
- Dichiarazioni di costanti e variabili:
  - La keyword `EQU` consente di dichiarare delle costanti (label)
  - Una label può essere utilizzata sia come dato che come indirizzo:

```
costante EQU 0fffh ; costante associato a 4095
movb costante, R0 ; Il byte all'indirizzo 4095 è copiato in R0
movb #costante, R0 ; R0 = 4095
```

  - Una variabile (globale) si dichiara con la keyword `Ds`:

```
A DL 123 ; A contiene il numero 123 a 32 bit
B DW 123 ; B contiene il numero 123 a 16 bit
C DB 123 ; C contiene il numero 123 a 8 bit
```

  - La keyword `Ds` può essere utilizzata per dichiarare un array:

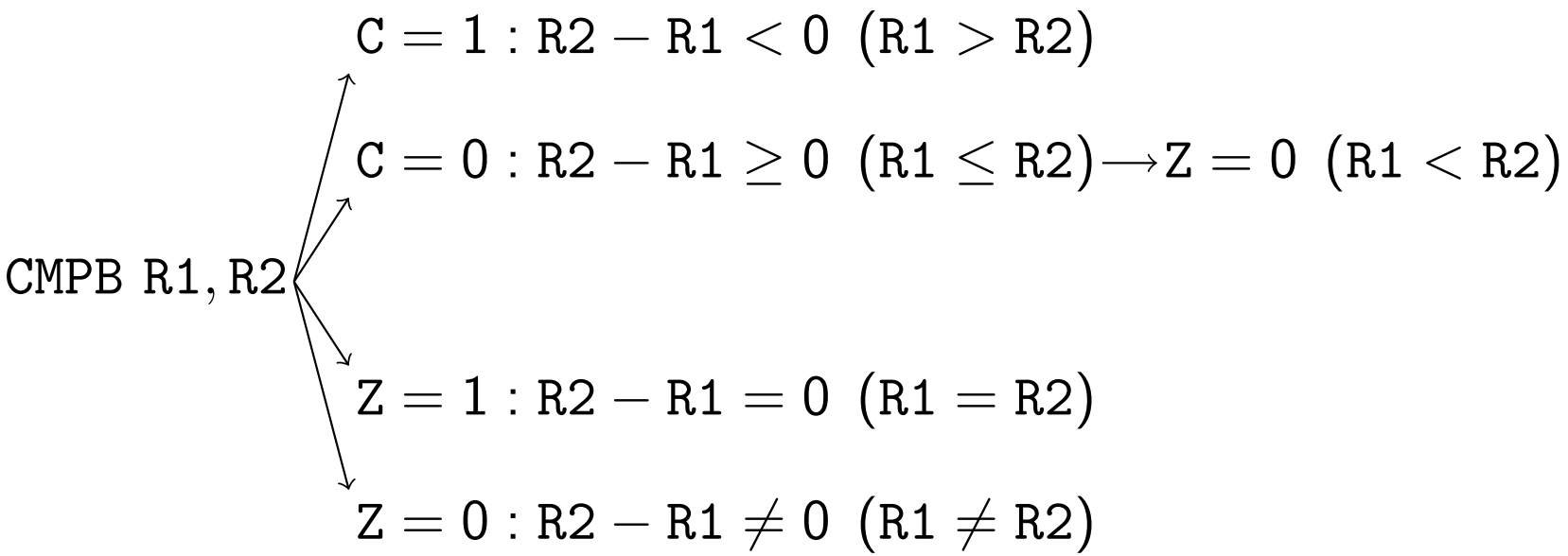
```
ARR DL 0, 1, 2, 3, 4 ; Gli elementi successivi sono posti in
modo contiguo in memoria
```

Scheletro di un programma assembly (3)

```
1  ORG 400h
2
3  dim EQU 3
4  arr DL 10, 9, 8
5
6  CODE
7      movb #dim, R0
8      movl #arr, R1
9
10 repeat:
11     movl (R1), R2
12     addl #4, R1
13     subb #1, R0
14     jnz repeat
15     halt
16 END
```

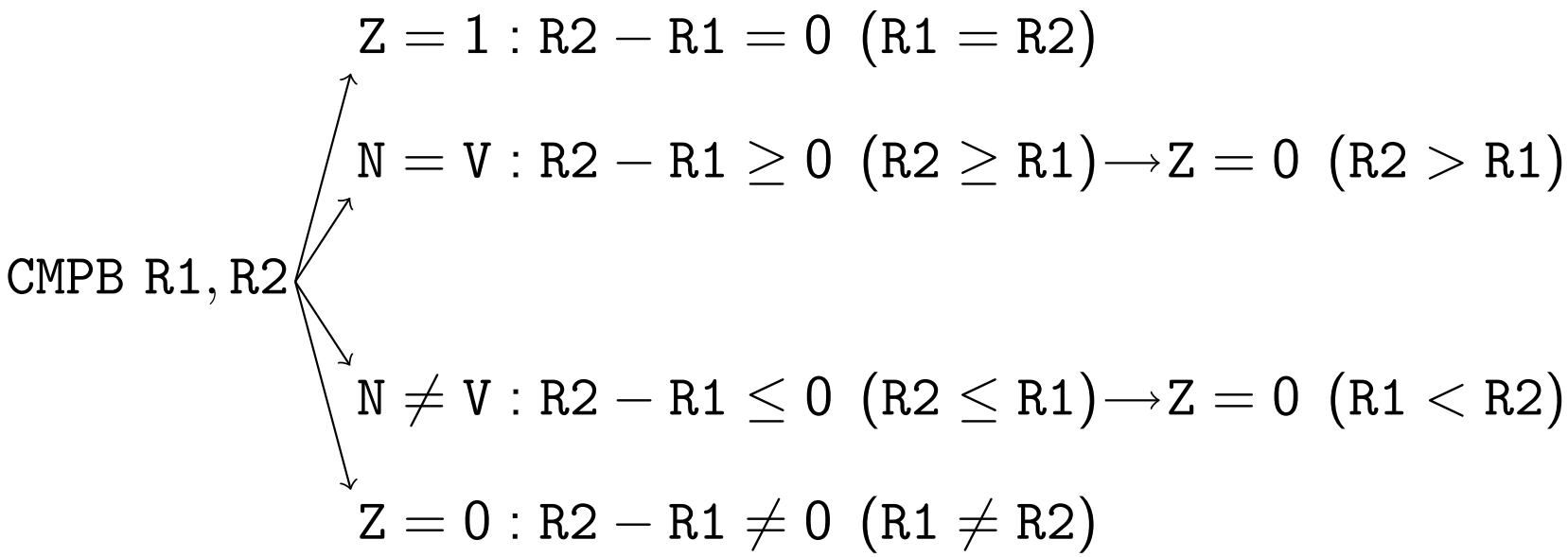
Confronti: aritmetica non segnata

L'aritmetica non segnata impone che R1, R2 siano ≥ 0.



Confronti: aritmetica segnata

L'aritmetica segnata implica che R1 ed R2 sono rappresentati in complemento a 2.



Confronti in breve

`CMPL R1, R2`

Condizione	*Aritmetica non segnata*	*Aritmetica segnata*
R2 $>$ R1	C $=$ 0 e Z $=$ 0	Z $=$ 0 e N $=$ V
R2 $\geq$ R1	C $=$ 0	N $=$ V
R2 $=$ R1	Z $=$ 1	Z $=$ 1
R2 $\leq$ R1	C $=$ 1 o Z $=$ 1	Z $=$ 1 o N $\neq$ V
R2 $<$ R1	C $=$ 1	N $\neq$ V
R2 $\neq$ R1	Z $=$ 0	Z $=$ 0

Confronti: un esempio

```
1  ORG 400h
2
3  x DW 3
4  y DW -2
5
6  CODE
7      ; Imposta a 1 l'indirizzo 1280h solo se x > y
8      ; Assumo che x ed y possano assumere valori negativi
9      movw x, R0
10     movw y, R1
11     cmpw R1, R0
12     jz nonImpostare
13
14     jn nset
15     jv nonImpostare ; eseguita se N=0. Se V=1 allora N != V
```

Confronti: un esempio (2)

```
16      jmp imposta
17  nset:
18      jnv nonImpostare ; eseguita se N=1. Se V=0 allora N != V
19
20  imposta:
21       movb #1, 1280h
22
23  nonImpostare:
24      halt
25  END
```

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 = 0;
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 400h
2
3  x DB 1
4  val DB 0
5
6  CODE
7
8      movb x, R0
9
10     cmpb #2, R0 ; Test prima condizione
11     jnz elseif
12     movb #2, val ; blocco A
13     jmp endif
14
15
```

If-Then-Else (2)

```
16 elseif:
17     cmpb #1, R0 ; Test seconda condizione
18     jnz else
19     movb #1, val ; blocco B
20     jmp endif
21
22 else:
23     movb #0, val
24
25 endif:
26     halt
27 END
```

Le variabili booleane?

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

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

Le variabili booleane?

```
1  ORG 400h
2
3  var DB 1 ; considerato come 'true'
4
5  CODE
6      movb var, R0
7      cmpb #0, R0
8      jz elsebranch
9      nop ; blocco A
10     jmp endif
11 elsebranch: nop ; blocco B
12 endif: halt
13 END
```

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: halt
```

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:

```
1 while(<condizione>) {
2     <codice>
3 }
```

```
1 test: jnz skip
2 ; <codice>
3 jmp test
4 skip:
```

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

```
1 begin: ; <codice>
2 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:

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

```
  movl #0, R0; R0 corrisponde a i
  movl #3, R1; R1 usato nel test
test: cmpl R0, R1
  jz skip
  ; <codice>
  addl #1, R0
  jmp test
skip:
```

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

```
cmpl #3, R0
```

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 `07h`
- Si può quindi eseguire un `AND` tra il dato e la maschera di bit

```
1 ; il dato si trova in in R0
2 movl #7, R1; 07h va anche bene
3 andl R0, R1; se si invertono i registri si perde il dato originale!
```

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 #80000000h, R0
```

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

```
1 andl #7FFFFFFFh, R0
```

Per invertire l'ultimo bit in `R0`:

```
1 xorl #80000000h, R0
```

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 movl #0, R0
2 xorl R0, R0
```

- Queste due istruzioni producono lo stesso risultato
- Tuttavia, è preferibile la seconda, perché è più efficiente (evita un accesso a memoria per caricare il valore 0)

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 utilizza la modalità di indirizzamento con postincremento 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**: come il modo due, ma l'elemento corrente del vettore viene aggiornato manualmente

Manipolazioni di vettori

Modo 1:

```
1   movl #array, R0 ; array: indirizzo primo elemento del vettore
2   movl #num, R1 ; num: numero di elementi
3 cycle:
4   movl (R0)+, ? ; carica un dato, usa il postincremento per andare
        all'elemento successivo
5   <processa i dati>
6   subl #1, R1
7   jnz cycle
```

Manipolazioni di vettori

Modo 2:

```
1  ORG 400h
2
3      array dl 1,2,3,4,5,6,7,8,9,10
4      endarr dl 0deadc0deh
5
6  CODE
7
8      movl #array, R0 ; array: indirizzo primo elemento del vettore
9      movl #endarr, R1 ; end: primo indirizzo fuori dal vettore
10   cycle:
11     addl (R0)+, R2 ; processa un dato e vai al successivo
12     cmpl R0, R1
13     jnz cycle
14     halt
15 END
```

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 movl #array, R0 ; array: indirizzo primo elemento del vettore
2 movl #num, R1 ; num: numero di elementi
3 lsll #2, R1 ; left shift per trasformare il numero in una taglia
4 ; Ogni elemento ha dimensione 4 byte ed uno shift di due posizioni
      corrisponde a moltiplicare per 4
5 ; R1 contiene quindi la dimensione (in byte) dell'array. Sommando
      il valore dell'indirizzo di base si ottiene il primo indirizzo
      fuori dall'array
6 addl R0, R1
```

e si può poi iterare utilizzando il modo 2

Manipolazioni di vettori

Modo 3:

```
1   movl #num, R1 ; num: numero di elementi
2   lsll #2, R1 ; calcola la dimensione dell'array. lsll #1 se gli
        elementi fossero da 2 byte
3   xorl R0, R0 ; azzera il contatore
4 cycle:
5   movl array(R0), ? ; sposta i dati dove serve
6   <processa i dati>
7   addl #4, R0 ; oppure 2, o 1, in funzione della dimensione
8   cmpl R0, R1
9   jnz cycle
```

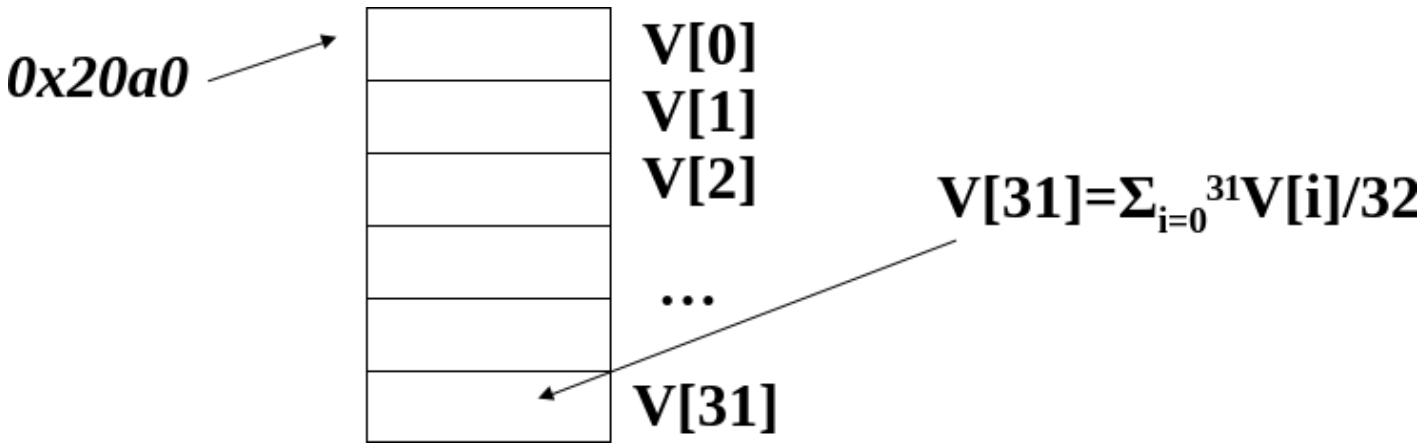
Manipolazioni di vettori

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   movl #array, R0
2   movl #num, R1
3 cycle:
4   addl (R0)+, R2
5   addl #4, R0; Salto un elemento!
6   subl #2, R1; Considero solo gli elmenti in posizione dispari
7   jn skip; R1 puo' diventare negativo senza passare per 0!
8   jnz cycle
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 400h
2      array EQU 20a0h ; indirizzo base vettore
3      dim EQU 32 ; num elementi array
4      log2dim EQU 5 ; log base 2 di dim
5  code
6      movl #dim, R0 ; R0: dimensione array
7      movl #array, R1 ; R1: indirizzo primo elemento del vettore
8      xorl R2, R2 ; R2: contiene risultato parziale, all'inizio 0
9    ciclo:
10     addb (R1)+, R2 ; somma i-esimo elem. i=0..DIM-1
11     jc error ; se bit c settato => Overflow
12     subl #1, R0 ; decrementa contatore
13     jnz ciclo ; se contatore != 0 continua
14     lsrb #log2dim, R2 ;dividi per il numero di elementi (lsrb per
           gli unsigned)
```

Esercizio: calcolo della media di un vettore (2)

```
15      movb R2, -(R1) ;memorizza la media in ARRAY[DIM-1]
16      jmp fine
17    error:
18      movl #dim, R1 ; gestione overflow, calcola in R1
19      addl #array, R1 ; R1=ARRAY+DIM
20      xorl R3, R3 ; R3 = 00000000
21      notl R3, R3 ; R3 = 11111111
22      movb R3, -(R1) ; ARRAY[DIM] = 11111111
23    fine:
24      halt
25  end
```

Esercizio: ordinamento di un vettore

```
1  org 400h
2
3      array EQU 1200h ; indirizzo base vettore
4      dim EQU 4 ; num elementi vettore
5
6  code
7      xorl R0, R0 ; indice del vettore
8      xorl R1, R1 ; minimo temporaneo
9      xorl R4, R4 ; registro contenente il dato da confrontare
10
11     movb #8, 1200h ; inizializzo il vettore
12     movb #3, 1201h
13     movb #10, 1202h
14     movb #5, 1203h
15
```

Esercizio: ordinamento di un vettore (2)

```
16  iniziomin:
17    cmpb #dim, R0
18    jz fine
19    xorl R3, R3 ; registro di spiazzamento
20    movl R0, R3 ; copio per usare R0 come spiazzamento
21    movb array(R3), R1 ; inizializzo il minimo parziale
22
23  ciclomin:
24    cmpl #dim, R3
25    jz finemin
26    movb array(R3), R4 ; R4 <= elemento corrente del vettore
27    addl #1, R3
28    cmpb R4, R1 ; se R4 < R1 allora Carry = 0 ed R4 nuovo minimo
29    jc ciclomin
30    movb R4, R1 ; scambia minimo
31    movb R3, R5 ; salvo la posizione
```

Esercizio: ordinamento di un vettore (3)

```
32      jmp ciclomin
33
34    finemin:
35      subb #1, R5
36      movb array(R0), array(R5) ; scambia con il valore da ordinare
37      movb R1, array(R0)
38
39      addl #1, R0 ; passo all'elemento successivo
40      jmp iniziomin
41
42  fine:
43      halt
44  end
```

Esercizio: Elemento maggiore tra due vettori

Dati due vettori, v1 e v2, costruire un terzo vettore v3 che contenga all'interno di ciascun elemento i-esimo il massimo tra gli elementi in posizione i-esima dei vettori v1 e v2

```
1  org 600h
2
3      v1 dl 0,1,2,3,4,5,6,7,8,9
4      v2 dl 9,8,7,6,5,4,3,2,1,0
5      v3 dl 0,0,0,0,0,0,0,0,0,0
6      dim equ 5
7
8  code
9      xorl R0, R0 ; Resetta il registro R0
10     movl #dim, R1
11     lsll #2, R1 ; R1 ora contiene lo spiazzamento dell'ultimo
           elemento
```

Esercizio: Elemento maggiore tra due vettori (2)

```
12
13  ciclo:
14      cmpl R0, R1
15      jz fine ; Siamo alla fine del vettore
16      movl v2(R0), R2
17      cmpl v1(R0), R2 ; Confronta gli elementi correnti di v1 e v2
18      jn Ntrue ; v2[i] - v1[i] < 0
19      jnv V2magg ; v2[i] - v1[i] > 0 e no overflow
20      jmp V1magg ; overflow
21
22   Ntrue:
23      jv V2magg ; overflow
24
25   V1magg:
26      movl v1(R0), v3(R0)
27      jmp Inc
```

Esercizio: Elemento maggiore tra due vettori (3)

```
28
29    V2magg:
30      movl v2(R0), v3(R0)
31
32    Inc:
33      addl #4, R0 ; Scorre all'elemento successivo (4 byte)
34      jmp ciclo
35
36    Fine:
37      halt
38  end
```

Esercizio: moltiplicazione tra numeri positivi

- Il PD32 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:

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

Moltiplicazione: soluzione naif

```
1  org 400h
2
3      op1 dl 6
4      op2 dl 3
5
6  code
7      xorl R0, R0 ; Accumulatore del risultato
8      movl op2, R1 ; Registro utilizzato per la terminazione
9     ciclo:
10     addl op1, R0 ; Un gran numero di accessi in memoria!
11     jc overflow ; Se incorriamo in overflow...
12     subl #1, R1
13     cmpl #0, R1
14     jz fine
15     jmp ciclo
```

Moltiplicazione: soluzione naif (2)

```
16
17    overflow:
18      movl #-1, R0 ; ...impostiamo il risultato a -1
19
20    fine:
21      halt
22  end
```

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

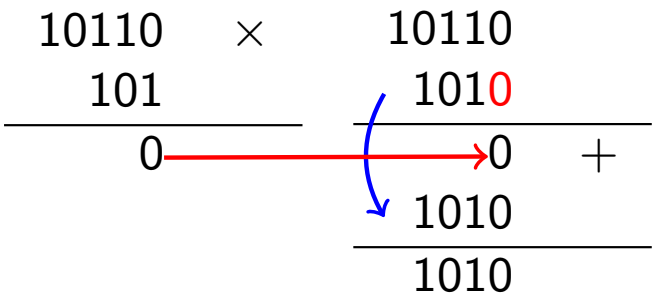
Shift-and-add: un esempio

Eseguiamo la moltiplicazione tra 22 e 5:

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

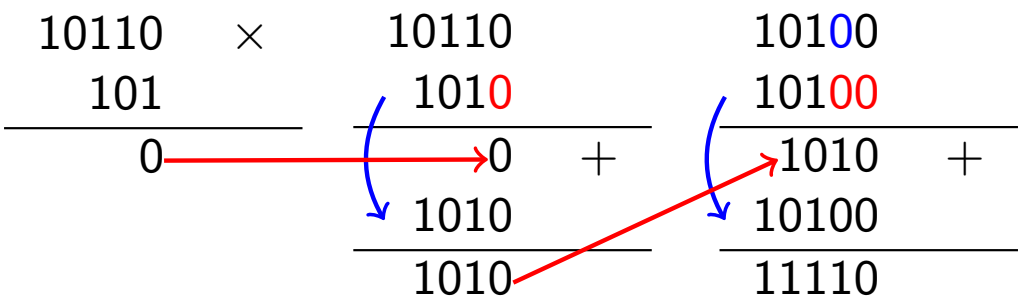
Shift-and-add: un esempio

Eseguiamo la moltiplicazione tra 22 e 5:



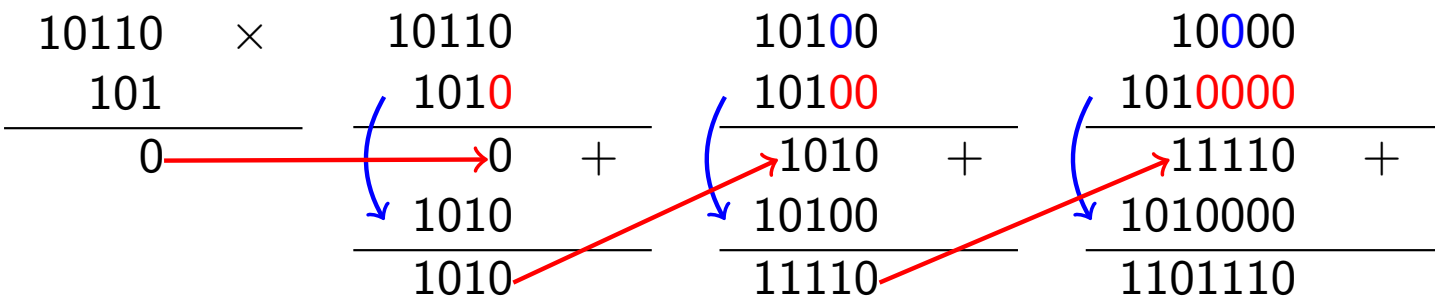
Shift-and-add: un esempio

Eseguiamo la moltiplicazione tra 22 e 5:



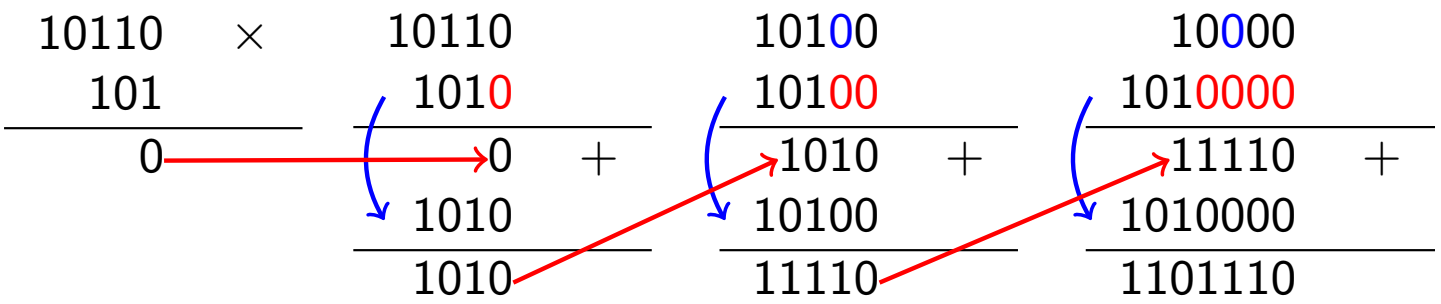
Shift-and-add: un esempio

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
- il PD32 non ha un'istruzione per calcolare il *least significant bit set* (cioè la posizione del bit meno significativo tra quelli impostati a 1)
  - Sarebbe necessario utilizzare un ciclo con all'interno uno shift e un contatore
- Il PD32 obbliga a specificare l'offset dello shift nell'istruzione
  - Dovremmo utilizzare un altro ciclo per shiftare il moltiplicatore

Shift-and-add: adattiamo il codice

- Ogni algoritmo, per essere efficiente, deve essere adattato all'architettura sottostante
- il PD32 non ha un'istruzione per calcolare il *least significant bit set* (cioè la posizione del bit meno significativo tra quelli impostati a 1)
  - Sarebbe necessario utilizzare un ciclo con all'interno uno shift e un contatore
- Il PD32 obbliga a specificare l'offset dello shift nell'istruzione
  - Dovremmo utilizzare un altro ciclo per shiftare il moltiplicatore
- 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 $\texttt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	101	$\texttt{C} = 1$
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 $\mathtt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	10	$\mathtt{C} = 1$
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:	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 $\texttt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

Moltiplicando:	1	$\texttt{C} = 0$
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 $\texttt{C} = 1$, devo sommare il moltiplicatore
  - Shift a sinistra del moltiplicatore
  - Ripeto il ciclo

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

Shift-and-add: il codice

```
1  org 400h
2      op1 dl 3
3      op2 dl 2
4
5  code
6      movl op1, R0 ; carico il moltiplicando
7      movl op2, R1 ; carico il moltiplicatore
8      xorl R2, R2 ; R2 conterra' il risultato
9
10     cmpl #0, R1
11     jz fine ; Se moltiplico per zero, ho finito
12
13   moltiplica:
14     cmpl #0, R0
15     jz fine ; Quando il moltiplicando e' zero ho finito
```

Shift-and-add: il codice (2)

```
16      lsrl #1, R0
17      jnc nosomma ; Se C = 0 non sommo
18      addl R1, R2
19    nosomma:
20      lsll #1, R1
21      jc overflow ; Basta controllare l'overflow qui
22      jmp moltiplica
23
24    overflow:
25      movl #-1, R2 ; In caso di overflow, imposto il risultato a -1
26
27    fine:
28      halt
29  end
```

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

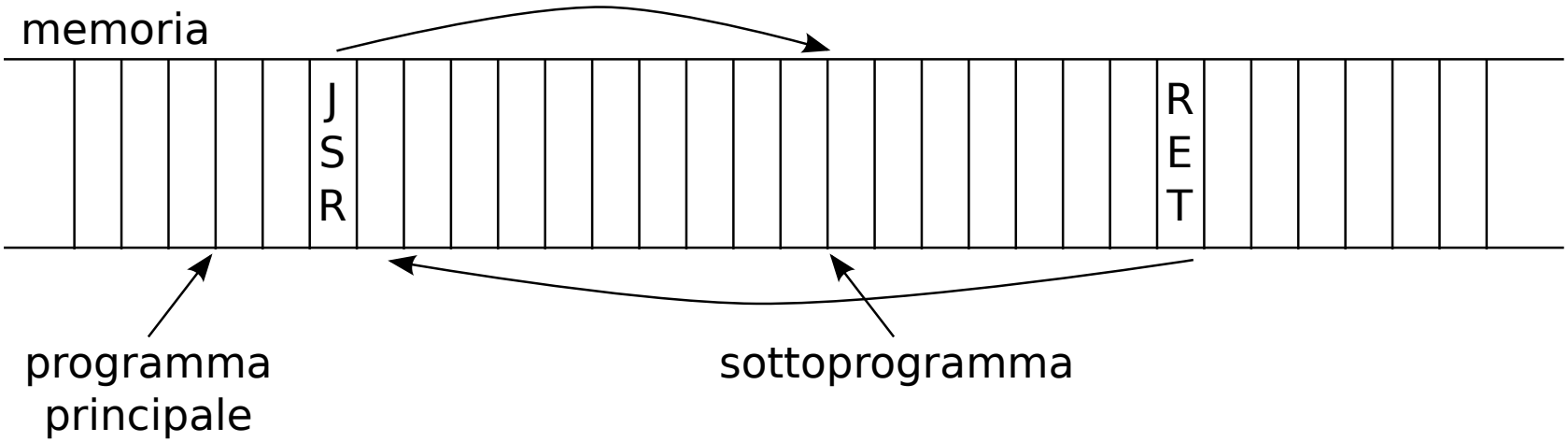
Salto a sottoprogramma

- Per eseguire un salto a sottoprogramma si utilizza l'istruzione *jump to subroutine* (`JSR`)
- La sintassi è la stessa del salto incondizionato:

```
JSR sottoprogramma
```

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

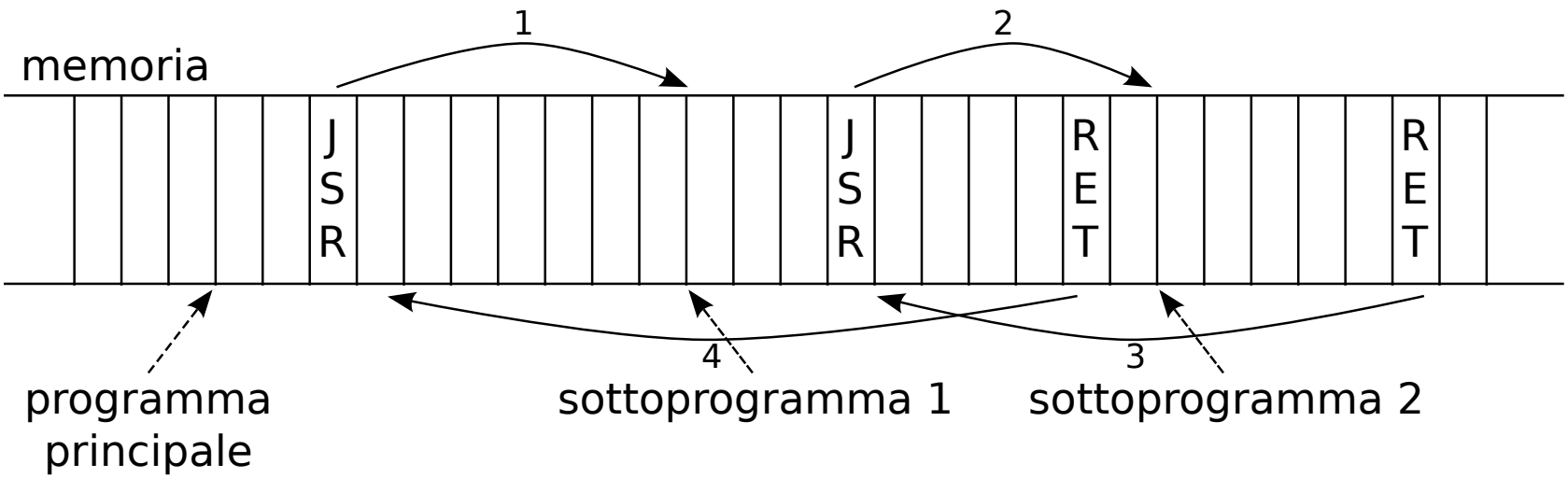
Salto a sottoprogramma (2)



Differenza tra JMP e JSR

- A differenza della JMP, il microcodice della JSR memorizza l'indirizzo dell'istruzione successiva (*indirizzo di ritorno*) prima di aggiornare il valore contenuto nel registro PC
- 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 `JSR` 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 `JSR` possono essere inseriti/prelevati altri elementi

Istruzioni di manipolazione dello stack

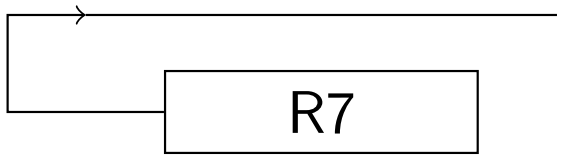
Tipo	Codice	Operandi	C N Z V P I	Commento
0	PUSH	S	- - - - - -	Sinonimo di MOVL S, -(R7)
0	POP	D	- - - - - -	Sinonimo di MOVL (R7)+, D
1	PUSHSR	–	- - - - - -	Sinonimo di MOVFRSR -(R7)
2	POPSR	–	- - - - - -	Sinonimo di MOVTOSR (R7)+

- Non si tratta di vere istruzioni, ma di *pseudoistruzioni*
  - Non sono implementate direttamente a livello hardware
  - Vengono tradotte dall'assemblatore in regolari istruzioni di movimento dati

Gestione dello stack

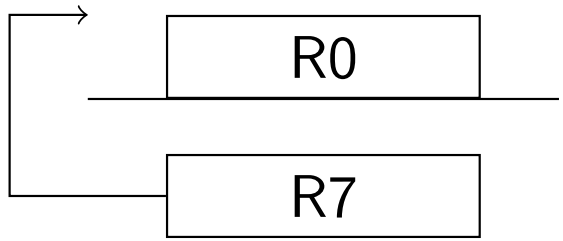
- Lo stack è composto da *longword* (non si può inserire nello stack un singolo byte)
- La cima dello stack è individuata dall'indirizzo memorizzato in un registro specifico chiamato SP (*Stack Pointer*)
- Nel PD32 lo SP coincide con il registro `R7`
  - Modificare il valore di `R7` coincide con il perdere il riferimento alla cima dello stack, e quindi a tutto il suo contenuto
- Lo stack "*cresce*" se il valore contenuto in `R7` diminuisce, "*decresce*" se il valore contenuto in `R7` cresce
  - Lo stack è posto in fondo alla memoria e cresce "*all'indietro*"

Gestione dello stack



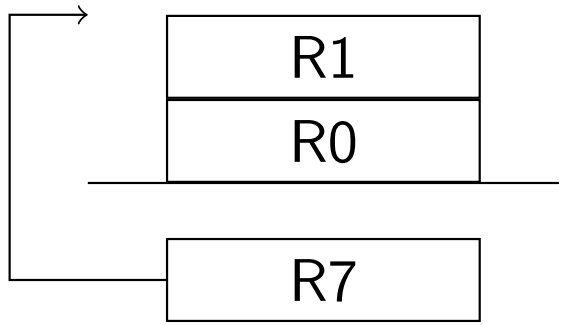
Gestione dello stack

```
push R0
```



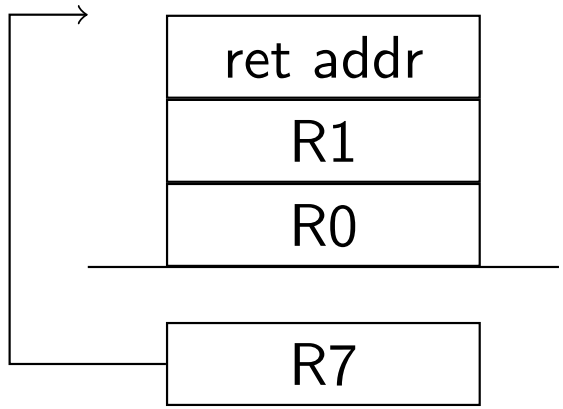
Gestione dello stack

```
push R0
push R1
```



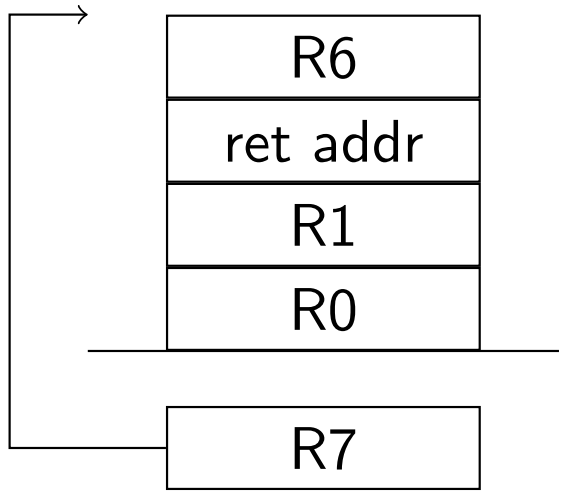
Gestione dello stack

```
push R0
push R1
jsr subroutine
```



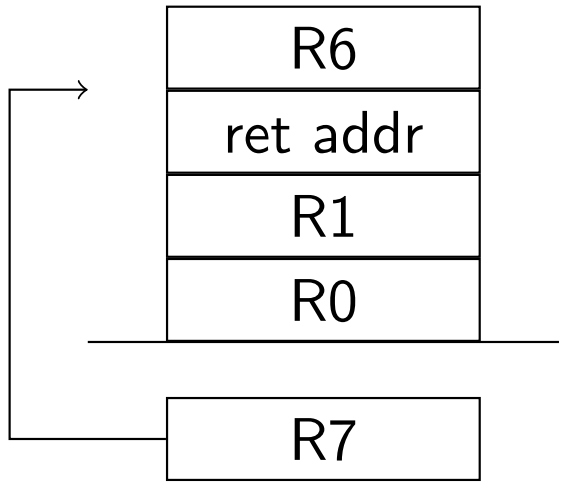
Gestione dello stack

```
push R0
push R1
jsr subroutine
push R6
```



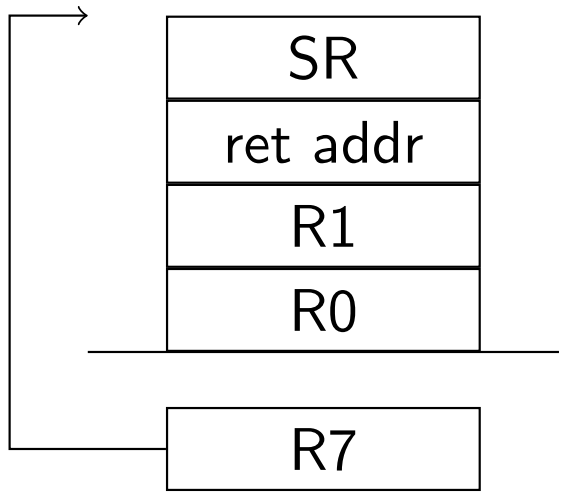
Gestione dello stack

```
push R0
push R1
jsr subroutine
push R6
pop R6
```



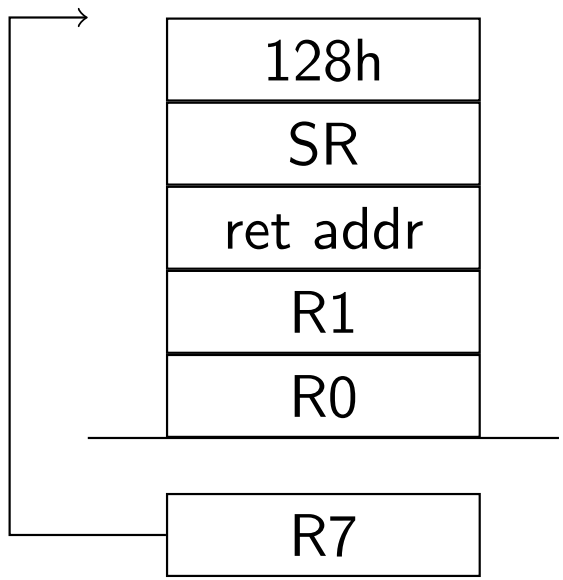
Gestione dello stack

```
push R0
push R1
jsr subroutine
push R6
pop R6
pushsr
```



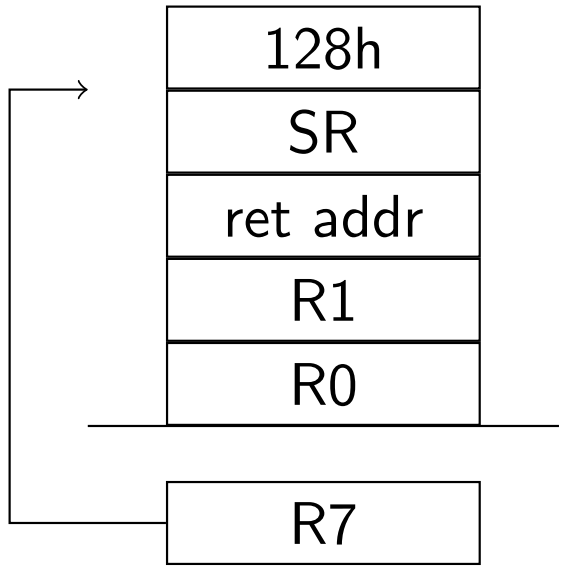
Gestione dello stack

```
push R0
push R1
jsr subroutine
push R6
pop R6
pushsr
push #128h
```



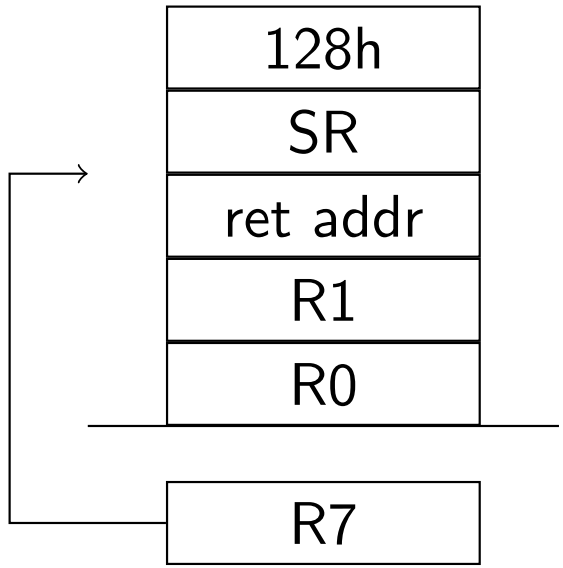
Gestione dello stack

```
push R0
push R1
jsr subroutine
push R6
pop R6
pushsr
push #128h
pop R0
```



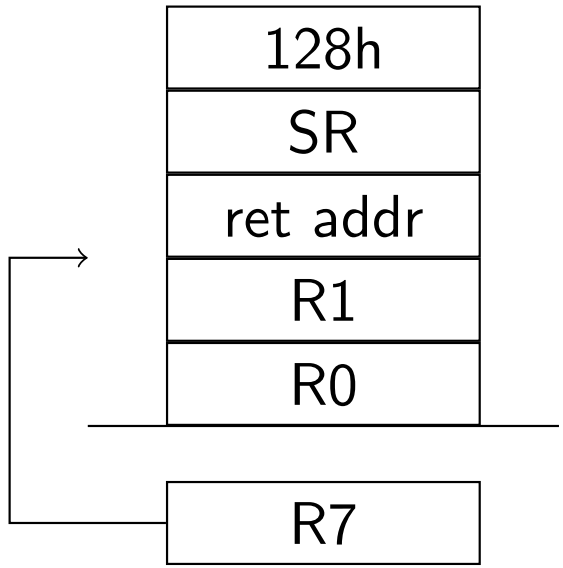
Gestione dello stack

```
push R0
push R1
jsr subroutine
push R6
pop R6
pushsr
push #128h
pop R0
popsr
```



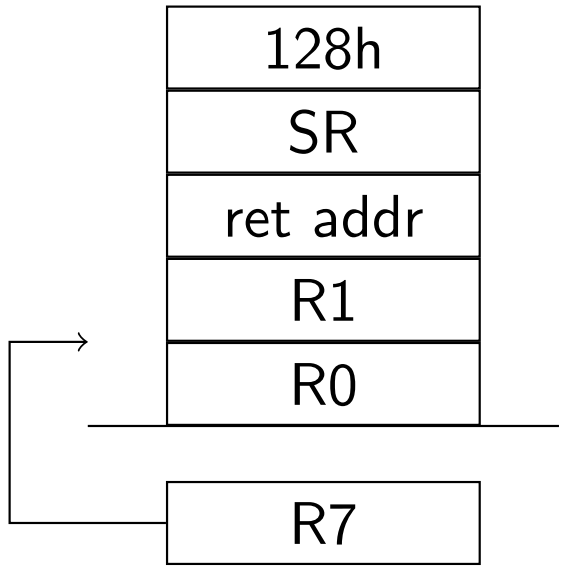
Gestione dello stack

```
push R0
push R1
jsr subroutine
push R6
pop R6
pushsr
push #128h
pop R0
popsr
ret
```



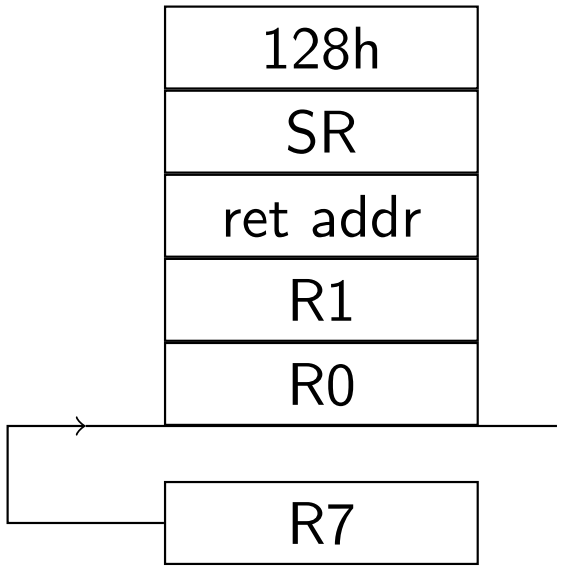
Gestione dello stack

```
push R0
push R1
jsr subroutine
push R6
pop R6
pushsr
push #128h
pop R0
popsr
ret
pop R1
```



Gestione dello stack

```
push R0
push R1
jsr subroutine
push R6
pop R6
pushsr
push #128h
pop R0
popsr
ret
pop R1
pop R0
```



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     push #128
3     ...
4     pop R0 ; !!!
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     push #128
3     ...
4     pop R0 ; !!!
5     ret
```

- Dopo aver fatto il push di tutte le variabili, si può accedere ad esse tramite R7:

$$(\mathtt{R7}),\ 4(\mathtt{R7}),\ 8(\mathtt{R7}),\ \ldots$$

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 (per esempio `R0`) 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 `jsr`

Esempio: calcolo del fattoriale

```
1  org 400h
2     numero dl 20
3     risultato equ 1200h
4
5  code ; Programma principale (main)
6      movl numero, R1
7      jsr FATT
8      jnc corretto
9      movl #-1, risultato
10     halt
11   corretto:
12     movl R0, risultato
13     halt
14
15
```

Esempio: calcolo del fattoriale (2)

```
16 ; Subroutine per la moltiplicazione
17 ; Parametri in R1 e R2
18 ; Valore di ritorno in R0
19 ; Se avviene overflow, ritorna con C=1
20 MULTIPLY:
21     xorl R0, R0
22   loop:
23     cmpl #0, R1
24     jz donemult
25     addl R2, R0
26     jc donemult
27     subl #1, R1
28     jmp loop
29   donemult:
30     ret
31
```

Esempio: calcolo del fattoriale (3)

```
32 ; Subroutine per il fattoriale
33 ; Parametro in R1
34 ; Valore di ritorno in R0
35 ; Se avviene overflow, ritorna con C=1
36 FATT:
37     xorl R0, R0
38     push R1
39
40     cmpl #1, R1 ; Se R0 = 1, esegui il passo base
41     jnz passoricorsivo
42
43     movl R1, R2
44     jmp passobase
45   passoricorsivo:
46     subl #1, R1
47     jsr FATT ; Salta alla stessa subroutine
```

Esempio: calcolo del fattoriale (4)

```
48      jnc nooverflow
49      pop R1 ; Con overflow, tolgo dallo stack i valori salvati
50      jmp doneFATT
51
52    nooverflow:
53      movl R0, R2
54
55    passobase:
56      pop R1
57      jsr MULTIPLY ; R0 = n * fatt(n-1)
58
59    doneFATT:
60      ret
61  end
```

Gli `switch-case`

- Quando il numero di rami all'interno di un costrutto `if-then-else` tende ad aumentare, si può ricorrere al costrutto `switch-case` che è una forma compatta di salto condizionato:

```
1  int x;
2
3  switch(x) {
4    case 0:
5       < blocco >
6       break;
7
8    case 1:
9       < blocco >
10      break;
11
12   ...
13 }
```

- Anche in linguaggio assembly è possibile utilizzare un costrutto analogo
- La tecnica si basa sulla costruzione di tabelle di salto (*branch tables*) e sull'utilizzo dell'indirizzamento

Gli switch-case: tabella in memoria

```
1  org 400h
2      branchTable dl 0, 0, 0
3      var dl 2
4  code
5      ; inizializza branch table
6      xorl R0, R0
7      movl #branchTable, R0
8      movl #case0, (R0)+
9      movl #case1, (R0)+
10     movl #case2, (R0)
11
12     ; Carica il valore della variabile su cui fare switch
13     movl var, R0
14     lsll #2, R0
15
16     ; Effettua il salto condizionato
```

Gli switch-case: tabella in memoria (2)

```
17      movl branchTable(R0), R0
18      jmp (R0)
19
20    case0:
21      ; <codice>
22      jmp fine
23    case1:
24      ; <codice>
25      jmp fine
26    case2:
27      ; <codice>
28      nop
29    fine: halt
30  end
```

Gli switch-case: tabella nel codice

```
1  org 400h
2      var dl 2
3  code
4      ; Carica il valore della variabile su cui fare switch
5      movl var, R0
6      lsll #2, R0
7      ; Effettua il salto condizionato
8      jmp branchTable(R0)
9
10     ; Tabella cablata nel codice
11   branchTable:
12     jmp case0
13     jmp case1
14     jmp case2
15     jmp case3
16
```

Gli switch-case: tabella nel codice (2)

```
17    case0:
18      ; <codice>
19      jmp fine
20    case1:
21      ; <codice>
22      jmp fine
23    case2:
24      ; <codice>
25      jmp fine
26    case3:
27      ; <codice>
28      nop
29    fine: halt
30  end
```