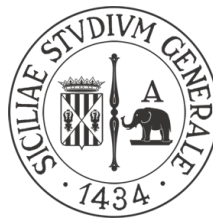


Corso di Architettura degli Elaboratori e Laboratorio (F-N)

Insieme di istruzioni macchina

Massimo Orazio Spata

Dipartimento di Matematica e Informatica



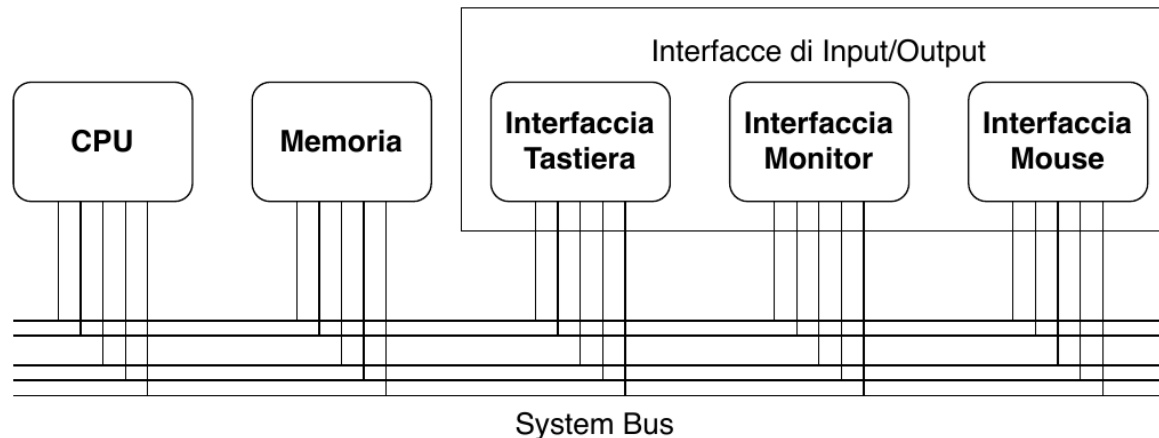
UNIVERSITÀ
degli STUDI
di CATANIA

.CPU: esegue istruzioni elementari

.MEMORIA: contiene il programma (sequenza di istruzioni elementari) che la CPU deve eseguire e i dati necessari

.INTERFACCE DI INPUT/OUTPUT: circuiti elettronici che permettono di connettere la CPU al mondo esterno

.BUS DI SISTEMA: insieme di collegamenti elettrici che interconnettono i vari componenti di un calcolatore

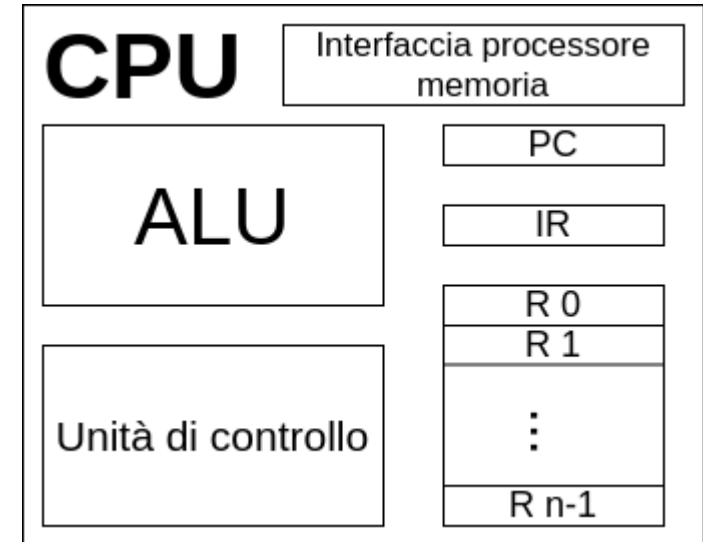


- Il calcolatore elettronico esegue **SEQUENZIALMENTE** una serie di **ISTRUZIONI**
- Le istruzioni definiscono delle operazioni da eseguire e sono raggruppate in **PROGRAMMI**
- Spesso le operazioni devono essere eseguite su dei **DATI**
- L'utente può interagire con il calcolatore tramite le **INTERFACCE DI I/O** (**PERIFERICHE**)

•È un **CIRCUITO ELETTRONICO INTEGRATO** (chip) con il ruolo di **CERVELLO** del calcolatore

•Capace di caricare ed eseguire le **ISTRUZIONI ELEMENTARI** necessarie per eseguire i **PROGRAMMI**

•Esempi di istruzioni elementari: operazioni aritmetiche, operazioni logiche, confronti, salti incondizionati e condizionati.



Passi operativi elementari per ciascuna istruzione:

.PRELIEVO: prelievo della prossima istruzione dalla memoria (scrivere la prossima istruzione nel registro di istruzione IR)

.DECODIFICA: decodifica dell'istruzione (quale operazione bisogna eseguire? Dove si trovano i dati da usare?)

.ESECUZIONE: esecuzione dell'istruzione (leggere o scrivere un dato in memoria, eseguire operazioni matematiche e logiche sui registri)

- Le unità memoria sono usate per immagazzinare informazione necessaria per eseguire i programmi
- Sono circuiti elettronici in grado di preservare l'informazione che può essere costituita da:
 - **ISTRUZIONI**, eseguite dalla CPU
 - **DATI**, utilizzati dalle istruzioni eseguite
- La memoria si può dividere in **MEMORIA CENTRALE** e **MEMORIA DI MASSA**

Rappresentazione di istruzioni e dati

• Le istruzioni e i dati sono rappresentati da **SEQUENZE** di **CIFRE BINARIE (bit)**

• Per convenzione una sequenza di 8 bit è detta **Byte**

• I byte vengono raggruppati in blocchi con un numero di elementi espresso con potenze di 2:

• Kilobyte = KB = $2^{10} = 1024 \approx 10^3$

• Megabyte = MB = $2^{20} = 1024 * 1024 \approx 10^6$

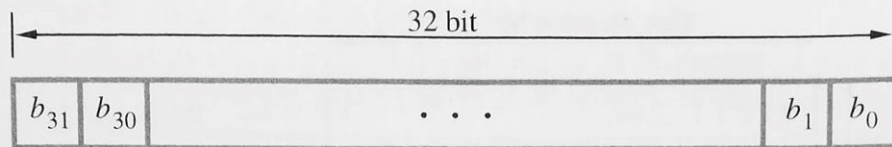
• Gigabyte = GB = $2^{30} = 1024 * 1024 * 1024 \approx 10^9$

• Terabyte = TB = $2^{40} = 1024 * 1024 * 1024 * 1024 \approx 10^{12}$

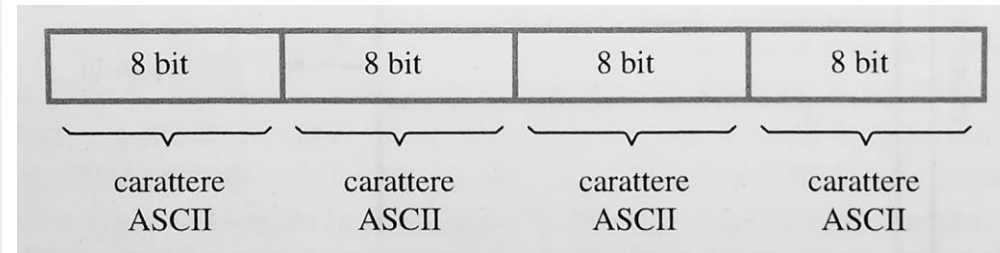


Parola di memoria (memory word)

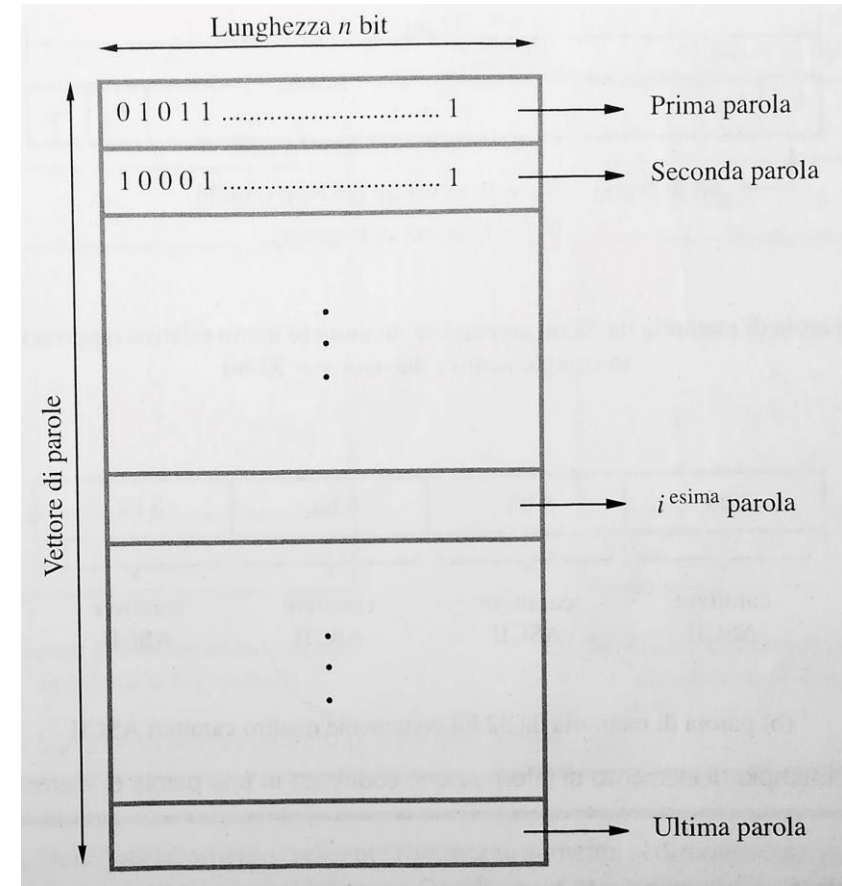
- Il calcolatore non lavora su singoli bit ma su gruppi di bit detti **PAROLE** di lunghezza da 8 a 64 bit (sempre potenze di 2)
- La dimensione delle parole dipende dall'architettura del calcolatore
- I dati posso occupare da un singolo byte a diverse parole
- Le istruzioni possono occupare una o più parole



bit di segno: $b_{31} = 0$ se valore positivo o nullo
 $b_{31} = 1$ se valore negativo



- L'informazione è immagazzinata in memoria sotto forma di un vettore di parole (parole in successione)
- Ad ogni parola nel vettore è associato un indirizzo binario univoco
- Un numero binario di m bit può rappresentare 2^m indirizzi
- Parole consecutive sono associate ad indirizzi consecutivi
- L'insieme degli indirizzi associati a ciascuna parola di memoria è chiamato spazio di indirizzamento

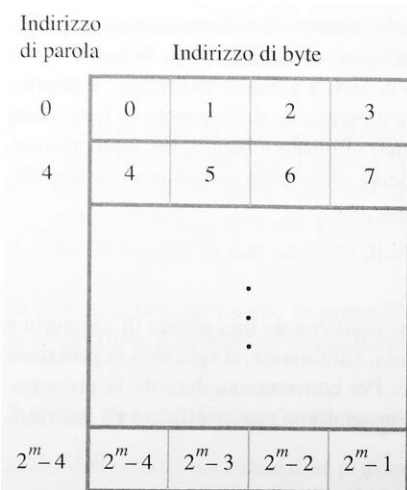


- Di norma l'unità minima di informazione indirizzabile in memoria è il byte
- Si assegnano indirizzi consecutivi ai byte contenuti in ciascuna parola
- Gli indirizzi delle parole saranno quindi multipli della loro lunghezza in byte

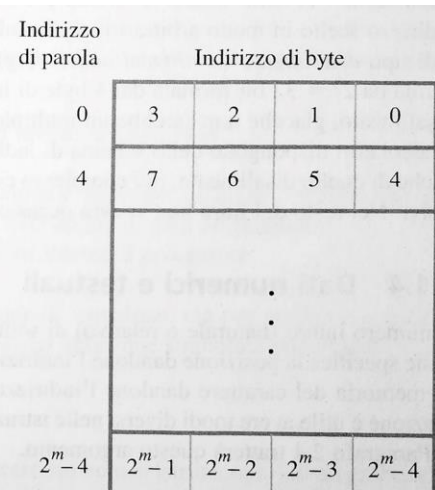
• Vi sono 2 schemi di indirizzamento di byte:

• **Crescente (big-endian)**: indirizzo aumenta al diminuire del peso aritmetico del byte

• **Decrescente (little-endian)**: indirizzo aumenta all'aumentare del peso aritmetico del byte



(a) schema di indirizzamento di byte
crescente o big-endian

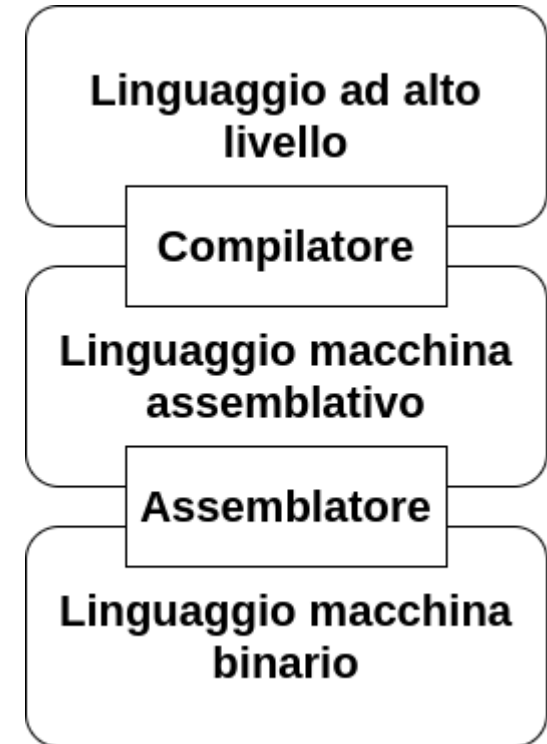


(b) schema di indirizzamento di byte
decrescente o little-endian

- Il processore è in grado di eseguire un insieme di operazioni base chiamate istruzioni macchina
- L'insieme delle istruzioni eseguibili da un processore e le loro modalità d'uso è chiamato ISA (Instruction Set Architecture)
- Ogni processore commerciale ha il suo specifico ISA
- Il linguaggio macchina permette di definire le istruzioni attraverso un alfabeto binario $\{0, 1\}$
- Il linguaggio assembly è una rappresentazione simbolica leggibile del linguaggio macchina

Come si programma un elaboratore?

- Il programmatore scrive i programmi in **LINGUAGGIO ASSEMBLATIVO (ASSEMBLY)**
- Il programma assembly viene tradotto in sequenze binarie dall'**ASSEMBLATORE**
- Linguaggi ad alto livello (C, C++, etc.) ancora più espressivi
- Il **COMPILATORE** traduce il codice ad alto livello in codice assembly



Esistono due approcci nella progettazione dell'insieme di istruzioni dei calcolatori:

Reduced Instruction Set Computer (RISC):

- Insieme di istruzioni base ridotto
- Ogni istruzione occupa una sola parola di memoria
- Gli operandi delle istruzioni aritmetiche e logiche devono trovarsi nei registri del processore
- Prestazioni elevate grazie ad un'elaborazione a stadi (pipeline)

Complex Instruction Set Computer (CISC):

- Insieme di istruzioni base complesse
- Ogni istruzione può occupare più di una parola di memoria
- Gli operandi delle istruzioni aritmetiche e logiche possono trovarsi in memoria

- I programmi eseguiti da un calcolatore sono composti da una sequenza di istruzioni base (addizione, confronto, caricamento di dati, ecc.)
- L'insieme di istruzioni riconosciute deve comprendere almeno queste quattro tipologie:
 - Trasferimento dati tra memoria e registri del processore
 - Operazioni aritmetiche e logiche sui dati
 - Operazioni di controllo dell'ordine di esecuzione delle istruzioni
 - Trasferimento dati tra unità di I/O e registri del processore

- Le diverse ISA dei processori commerciali posseggono linguaggi assemblativi con formalismi differenti (sebbene simili)
- Nella teoria di questo corso useremo un linguaggio assemblativo generico non appartenente a nessun processore commerciale
- Utile per capire i concetti base che possono essere applicati a qualsiasi architettura
- Verrà presentato un set di istruzioni base per programmare un processore nella pratica (non il set di istruzioni completo)

•È necessario definire una notazione formale per riferirsi ai registri e alle locazioni di memoria nel linguaggio assemblativo generico

•I registri sono identificati attraverso il loro nome:

•Registri generici del processore: $R0, R1, \dots, Rn$

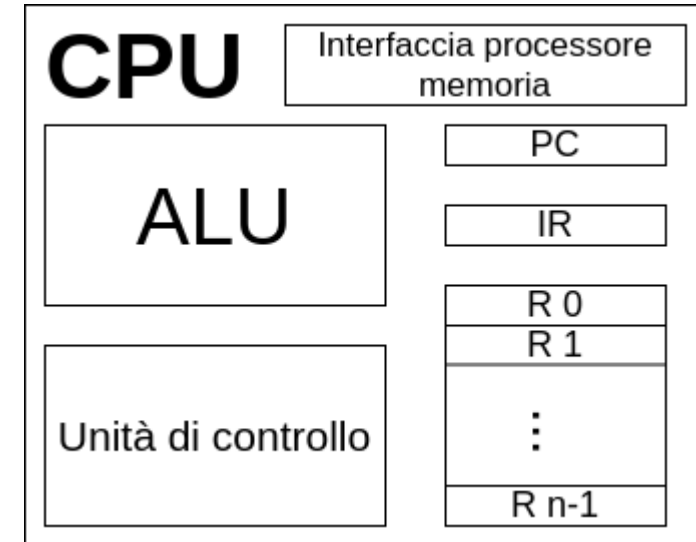
•Registri speciali del processore: PC, IR, ecc.

•Registri di I/O: INGRESSO_DATO, USCITA_DATO, ecc.

•Le locazioni di memoria sono identificate attraverso il loro indirizzo in forma:

•Di costante numerica

•Di costante simbolica dichiarata in precedenza: VAR1, IND, CICLO, ecc.



Load *destinazione sorgente*

- Istruzione usata per caricare un dato dalla memoria ad un registro del processore
- Il campo destinazione è il nome di un registro del processore
- Il campo sorgente è una locazione di memoria
- La locazione di memoria può essere indicata in vari modi a seconda del modi di indirizzamento usato

Store *sorgente destinazione*

- Istruzione usata per salvare in memoria un dato presente in un registro del processore
- Il campo sorgente è il nome di un registro del processore
- Il campo destinazione è una locazione di memoria
- La locazione di memoria può essere indicata in vari modi a seconda del modi di indirizzamento usato

Add *destinazione sorgente1 sorgente2*

- Istruzione usata per sommare il contenuto di due registri
- Il campo destinazione è il nome di un registro del processore su cui scrivere la somma
- I campi sorgente1 e sorgente2 rappresentano i numeri da sommare
- Gli addendi possono essere espressi come nomi di registri o direttamente come valore

Subtract *destinazione sorgente1 sorgente2*

- Istruzione usata per sottrarre il contenuto di due registri
- Il campo destinazione è il nome di un registro del processore su cui scrivere la differenza
- I campi sorgente1 e sorgente2 rappresentano i numeri da sottrarre
- Gli operandi possono essere espressi come nomi di registri o direttamente come valore

Esempio di programma di somma

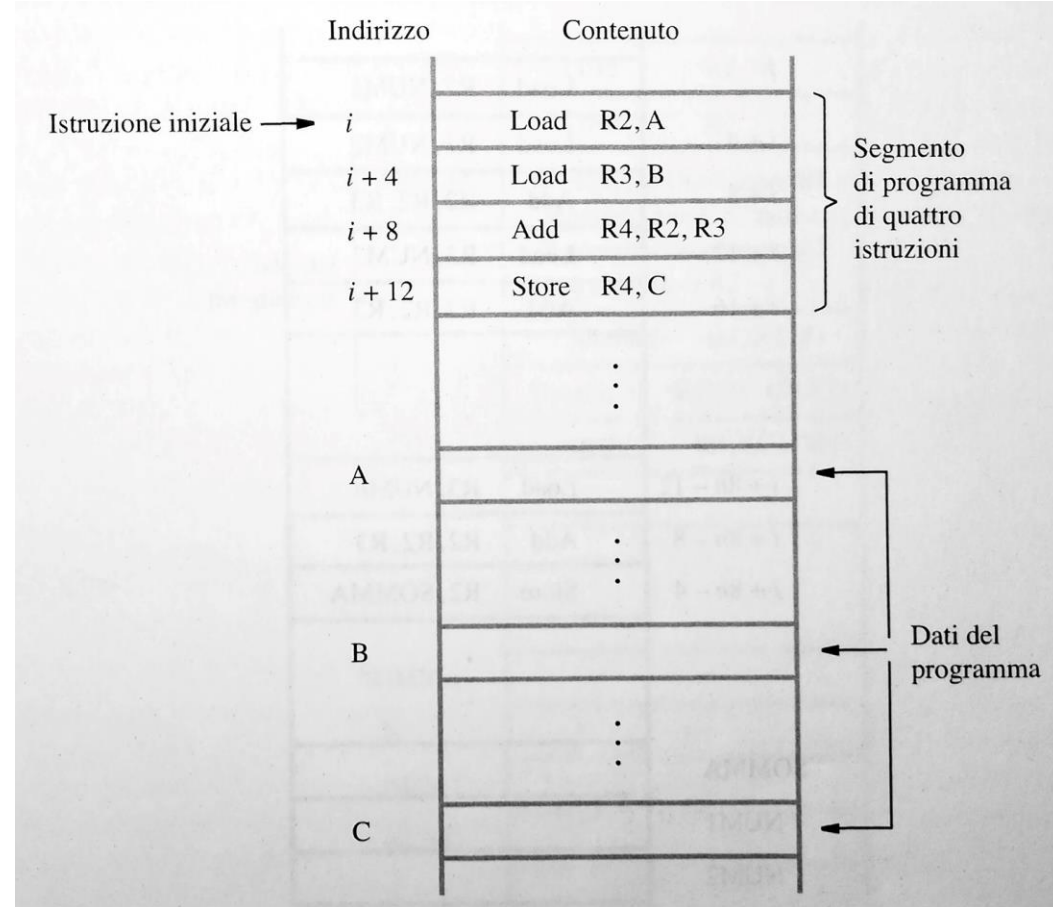
• Esempio di programma che somma due valori presenti in memoria e ne salva il risultato

• Programma composto da 4 istruzioni (2 Load, 1 Add e 1 Store)

• Le quattro istruzioni sono memorizzate in parole di memoria consecutive

• Istruzioni lette sequenzialmente

• PC contiene l'indirizzo della prossima istruzione da eseguire e IR contiene l'istruzione in esecuzione



- .Nel linguaggio assemblativo, gli operandi e il risultato delle istruzioni possono essere espressi in modi diversi
- .I metodi con cui specificare operandi e risultato vengono chiamati modi di indirizzamento
- .I modi di indirizzamento base di un'architettura RISC sono:
 - .Modo immediato**
 - .Modo di registro**
 - .Modo assoluto (diretto)**
 - .Indiretto da registro**
 - .Con indice e spiazzamento**
 - .Con base e indice**

• I modi di indirizzamento visti fino ad ora sono:

• **Modo di registro:** Il nome (= indirizzo) di un registro di processore contenente l'operando o il risultato è dato nell'istruzione

• **Modo assoluto (diretto):** L'indirizzo di una parola di memoria contenente l'operando o il risultato è dato nell'istruzione

• Nei processori RISC c'è un limite al numero di bit per un indirizzo assoluto (un'istruzione = una parola)

• Per processori a 32 bit = indirizzo assoluto 16 bit

Load R2, NUM1

• Per usare una costante numerica come operando si ricorre al modo di indirizzamento immediato

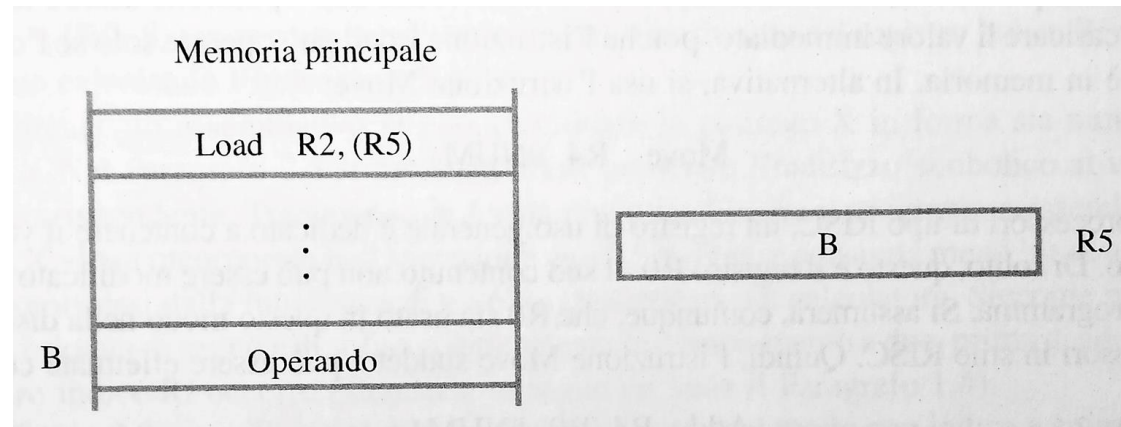
• **Modo immediato:** L'operando è dato esplicitamente nell'istruzione

• Si precede la costante dal simbolo cancelletto: **#valore**

• Esempio in cui si aggiunge il valore 200 al contenuto di R6 e si pone il risultato in R4:

Add R4, R6, #200

- **Modo indiretto:** Il nome di un registro di processore contenente l'INDIRIZZO di memoria dell'operando o del risultato è dato nell'istruzione
- Viene rappresentato con il nome del registro tra parentesi tonde (·)
- Usato in casi in cui si voglia riutilizzare una stessa istruzione in memoria più volte cambiando gli operandi

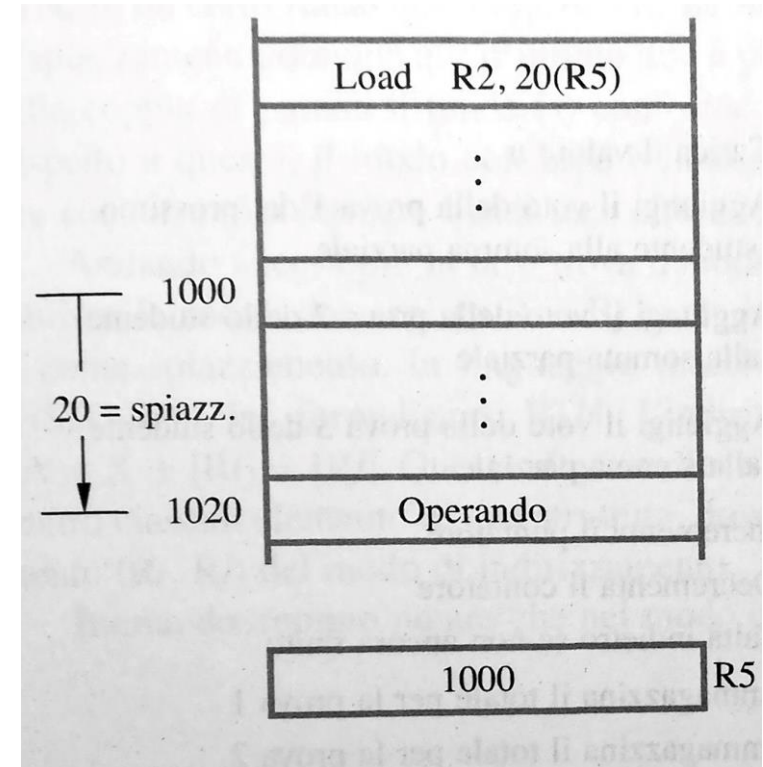


.Modo con indice e spiazzamento: L'indirizzo effettivo di operando o risultato è ottenuto addizionando un valore costante (spiazzamento) al contenuto di un registro (indirizzo)

• Per indicare indice e spiazzamento si usa la scrittura $X(R_i)$, dove X è lo spiazzamento e R_i è il nome del registro contenente l'indirizzo

• Utile nel gestire vettori o liste

• Esistono versioni più complesse come il **modo con base e indice** dove l'indirizzo effettivo è ottenuto sommando il contenuto di due registri, denotato così: (R_i, R_j)



Branch_if_condizione *destinazione_salto*

- Istruzione usata per saltare all'esecuzione di un'istruzione specifica nel caso la condizione di salto sia vera
- La condizione di salto può essere tra valori contenuti nei registri (espressi tra quadre: [Ri]) o valori espressi esplicitamente
- La destinazione del salto è espressa come locazione di memoria contenente l'istruzione da eseguire nel caso la condizione sia vera
- Esempio di salto all'istruzione CICLO nel caso il contenuto di R2 sia maggiore di 0:

Branch_if_[R2]>0 CICLO

Esempio somma di n numeri

i	Load	R2, NUM1
$i + 4$	Load	R3, NUM2
$i + 8$	Add	R2, R2, R3
$i + 12$	Load	R3, NUM3
$i + 16$	Add	R2, R2, R3
		⋮
$i + 8n - 12$	Load	R3, NUM n
$i + 8n - 8$	Add	R2, R2, R3
$i + 8n - 4$	Store	R2, SOMMA
		⋮
SOMMA		
NUM1		
NUM2		
		⋮
NUM n		

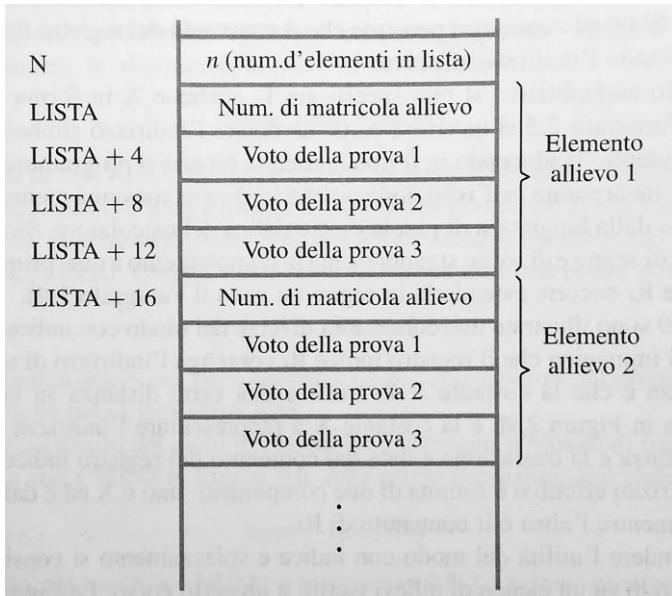
**Esempio sequenziale
(poco efficiente)**

**Esempio con salto
(più efficiente)**

	Load	R2, N
	Clear	R3
Ciclo del programma	CICLO	
	Determina l'indirizzo del prossimo numero, carica il prossimo numero in R5 e aggiungilo a R3	
	Subtract	R2, R2, #1
	Branch_if_[R2]>0	CICLO
	Store	R3, SOMMA
		⋮
SOMMA		
N		n
NUM1		
NUM2		
		⋮
NUM n		

Esempio somma di n numeri con salto

	Load	R2,N	Carica la dimensione della lista
	Clear	R3	Inizializza la somma a 0
	Move	R4,#NUM1	Carica l'indirizzo del primo numero
CICLO:	Load	R5, (R4)	Preleva il prossimo numero
	Add	R3, R3, R5	Aggiungi questo numero alla somma
	Add	R4, R4, #4	Incrementa il puntatore alla lista
	Subtract	R2, R2, #1	Decrementa il contatore
	Branch_if_[R2]>0	CICLO	Salta indietro se non ancora finito
	Store	R3, SOMMA	Immagazzina la somma finale



	Move	R2, #LISTA	Carica l'indirizzo LISTA
	Clear	R3	
	Clear	R4	
	Clear	R5	
	Load	R6, N	Carica il valore n
CICLO:	Load	R7, 4(R2)	Aggiungi il voto della prova 1 del prossimo studente alla somma parziale
	Add	R3, R3, R7	
	Load	R7, 8(R2)	Aggiungi il voto della prova 2 dello studente alla somma parziale
	Add	R4, R4, R7	
	Load	R7, 12(R2)	Aggiungi il voto della prova 3 dello studente alla somma parziale
	Add	R5, R5, R7	
	Add	R2, R2, #16	Incrementa il puntatore
	Subtract	R6, R6, #1	Decrementa il contatore
	Branch_if_[R6]>0	CICLO	Salta indietro se non ancora finito
	Store	R3, SOMMA1	Immagazzina il totale per la prova 1
	Store	R4, SOMMA2	Immagazzina il totale per la prova 2
	Store	R5, SOMMA3	Immagazzina il totale per la prova 3

- L'assemblatore è in grado di produrre il codice macchina binario (programma oggetto) di un programma scritto in codice assembleativo (programma sorgente)
- Per produrre il programma oggetto l'assemblatore deve risolvere inoltre i seguenti problemi:
 - Assegnare valori numerici a nomi e simboli
 - Dove collocare in memoria le istruzioni macchina
 - Dove collocare in memoria gli operandi e i risultati del programma
- Per questo il linguaggio assembleativo non contiene solo le istruzioni del programma, ma anche comandi specifici per l'assemblatore (direttive di assemblatore)
- Le direttive di assemblatore non vengono tradotte nel programma oggetto, ma servono per dare informazioni utili all'assemblatore

- Serve per associare un valore numerico ad un nome usato nel programma sorgente
- La sua sintassi nel nostro linguaggio generico è la seguente:

NOME **EQU** *Valore_numerico*

- Per produrre il programma oggetto, l'assemblatore sostituirà ogni occorrenza della stringa *NOME* nel programma sorgente con il valore *Valore_numerico*

- Indica all'assemblatore l'indirizzo di partenza dove inserire le istruzioni e i dati definiti nelle righe seguenti
- La sua sintassi nel nostro linguaggio generico è la seguente:

ORIGIN *Indirizzo_di_memoria*

- Alle istruzioni e ai dati seguenti la direttiva ORIGIN verranno assegnati gli indirizzi a partire dall'indirizzo *Indirizzo_di_memoria*

- Indica all'assemblatore di riservare uno spazio di memoria espresso in byte
- La sua sintassi nel nostro linguaggio generico è la seguente:

RESERVE

Spazio_in_byte

- La locazione di memoria riservata non viene inizializzata

- Indica all'assemblatore di riservare una parola di memoria e le assegna un contenuto
- La sua sintassi nel nostro linguaggio generico è la seguente:

DATAWORD

Contenuto_da_assegnare

- La parola di memoria viene inizializzata con il valore *Contenuto_da_assegnare*

.Nel maggiore dei casi, una linea di codice assemblativo presenta i seguenti campi:

Etichetta ***Operazione*** ***Operandi*** ***Commento***

.Etichetta: Nome che viene associato all'indirizzo della parola di memoria assegnata all'istruzione o all'indirizzo del blocco di memoria riservato. È facoltativa

.Operazione: Il nome (codice operativo) dell'istruzione oppure una direttiva di assemblatore.

.Operandi: Informazione di indirizzamento per accedere agli operandi

.Commento: Testo di commento, ignorato dall'assemblatore

Memoria

	100	Load	R2, N
	104	Clear	R3
	108	Move	R4, #NUM1
CICLO	112	Load	R5, (R4)
	116	Add	R3, R3, R5
	120	Add	R4, R4, #4
	124	Subtract	R2, R2, #1
	128	Branch_if_[R2]>0	CICLO
	132	Store	R3, SOMMA
	
SOMMA	200		
N	204	150	
NUM1	208		
NUM2	212		
		...	
NUMn	804		

	ORIGIN	100
	Load	R2, N
	Clear	R3
	Move	R4, #NUM1
CICLO:	Load	R5, (R4)
	Add	R3, R3, R5
	Add	R4, R4, #4
	Subtract	R2, R2, #1
	Branch_if_[R2]>0	CICLO
	Store	R3, SOMMA

	ORIGIN	200
SOMMA:	RESERVE	4
N:	DATAWORD	150
NUM1:	RESERVE	600
	END	

- L'assemblatore ci permette di denotare i numeri in diversi formati: binario, decimale, esadecimale

- Per indicare quale rappresentazione si vuole usare si usano dei prefissi:

- Binaria: %

- Decimale: nessun prefisso

- Esadecimale: 0x

- Esempio su come usare un operando della somma in modo immediato:

- Binario: Add R2, R3, #**%01011101**

- Decimale: Add R2, R3, #**93**

- Esadecimale: Add R2, R3, #**0x5D**

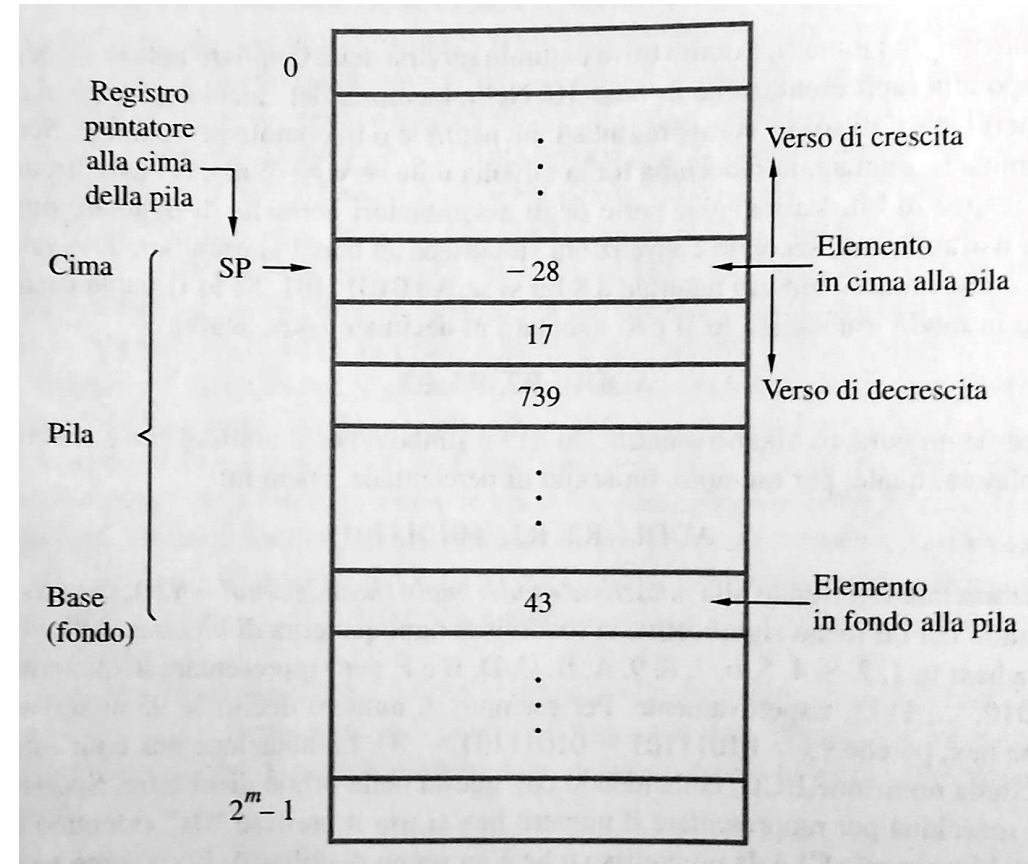
Pila (Stack)

•Lista di elementi (parole) dove si immaginano i dati posizionati uno sull'altro

•Gli elementi possono essere solo aggiunti e prelevati dalla cima della pila: l'ultimo elemento inserito è il primo ad essere prelevato (Last In First Out – LIFO)

•**Stack Pointer – SP:** registro che punta alla cima della pila

•Gli elementi della pila hanno indirizzi in ordine **decrescente** dalla base alla cima

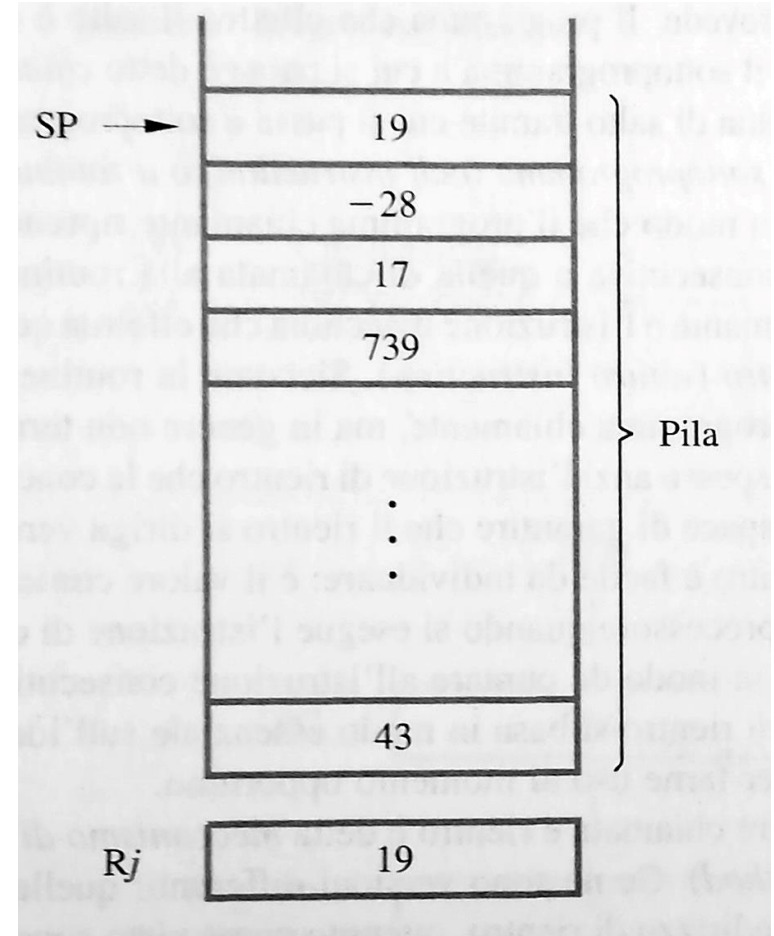


Push (Impila)

- L'operazione di **Push** aggiunge un elemento in cima alla pila
- In un architettura RISC si realizza con due istruzioni:
- Diminuire l'indirizzo contenuto in SP di una parola per puntare alla nuova cima
- Scrivere il valore richiesto nella parola puntata da SP

Subtract SP, SP, #4

Store Rj, (SP)

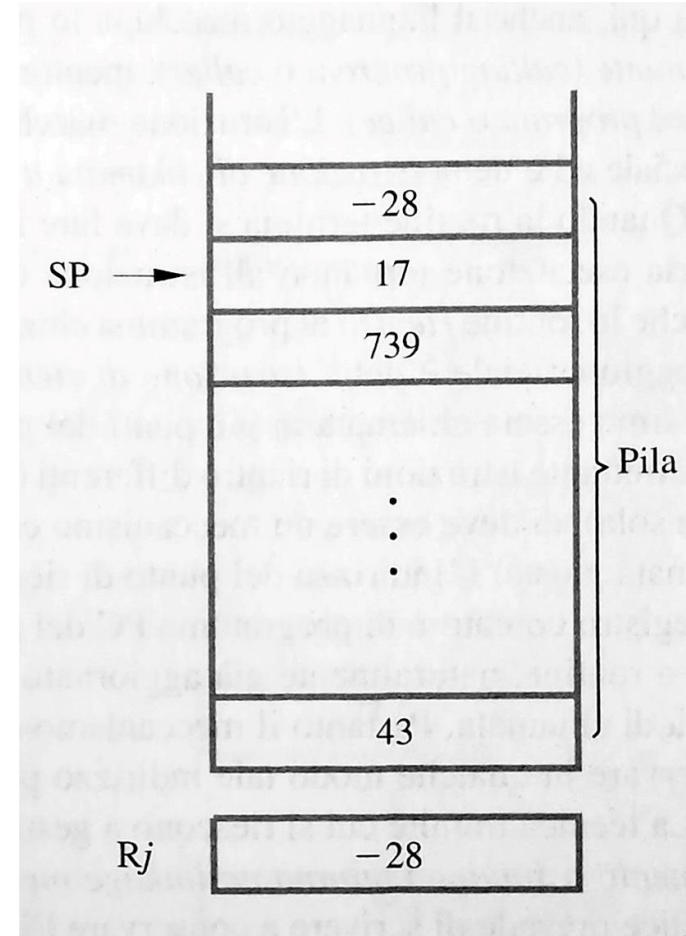


Pop (Spila)

- L'operazione di **Pop** preleva un elemento dalla cima della pila
- In un architettura RISC si realizza con due istruzioni:
 - Copiare il valore contenuto nella locazione di memoria puntata da SP in un registro del processore
 - Aumentare l'indirizzo contenuto in SP di una parola per puntare alla nuova cima

Load Rj, (SP)

Add SP, SP, #4



- Un **sottoprogramma o routine** è una lista di istruzioni che eseguono un compito specifico e che possono essere richiamate in un qualsiasi momento durante l'esecuzione di un programma
- Per richiamare una routine, un programma usa una funzione di salto particolare detta **Call instruction**
- Per ritornare dalla routine all'istruzione successiva alla Call nel programma principale si usa una funzione di salto particolare detta **Return instruction**
- Il **Link Register** è un registro speciale in cui si memorizza l'indirizzo dell'istruzione di rientro durante l'esecuzione di un sottoprogramma

•L'operazione di chiamata a sottoprogramma presenta questa sintassi:

Call INDIRIZZO

•L'operazione di chiamata esegue due passi:

- 1) Salva il contenuto del registro **PC** nel **Link Register**
- 2) Salta all'indirizzo di destinazione indicato nell'istruzione di chiamata

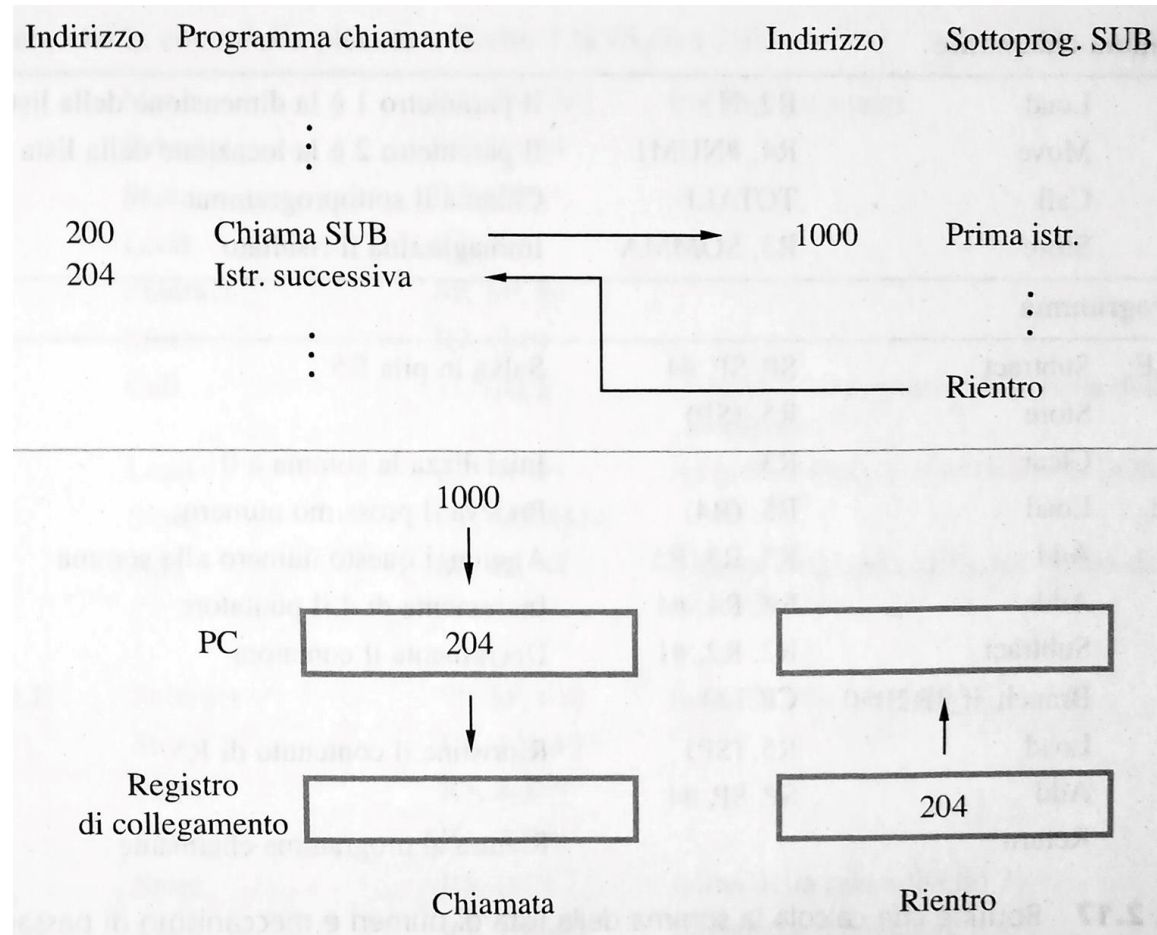
•L'istruzione di rientro da sottoprogramma presenta questa sintassi:

Return

•L'istruzione di rientro salta all'indirizzo di rientro contenuto nel Link Register nel seguente modo:

- 1) Salva il contenuto del **Link Register** nel registro **PC**

Meccanismo di collegamento a routine



- Una routine ha spesso bisogno di:
- Parametri di ingresso su cui operare
- Restituire un risultato al programma chiamante
- Esistono 2 tecniche di **passaggio di parametri**:
 - 1) Passaggio tramite registri del processore (si usano alcuni registri generici per salvare i parametri). Numero di parametri limitato dal numero di registri
 - 2) Passaggio attraverso la pila (si impilano i parametri nella pila). Numero di parametri virtualmente illimitato

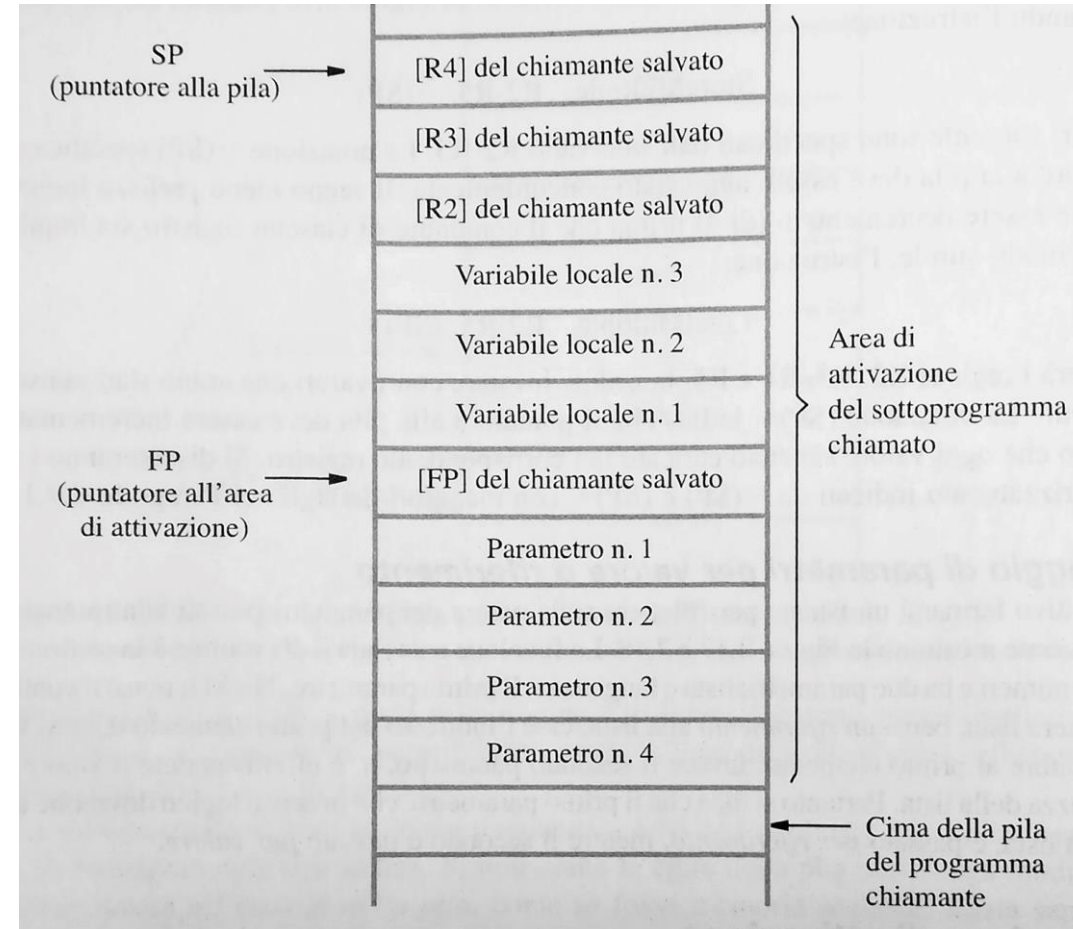
Area di attivazione in pila (stack frame)

• Il blocco di memoria nella pila riservato al sottoprogramma è chiamato **Area di attivazione (Stack Frame)**

• Il **Frame Pointer FP** è un registro che punta allo Stack Frame del sottoprogramma in esecuzione

• Lo Stack Frame contiene i parametri, il FP del programma chiamante, le variabili locali e valori di registri salvati

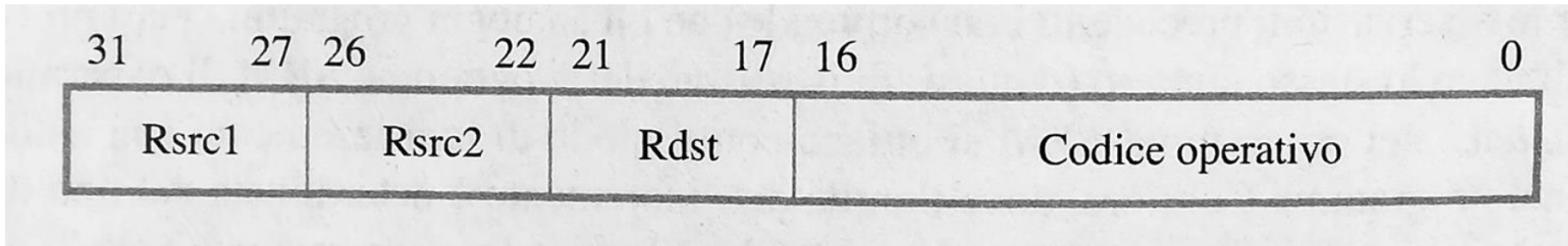
• Il FP punta alla parola dove è memorizzato il FP del programma chiamante



- Nel caso si abbiano diversi sottoprogrammi annidati, prima di chiamare una seconda routine è necessario salvare il contenuto del registro LINK_reg per recuperarlo in seguito
- Si può usare la pila per salvare gli indirizzi di rientro delle chiamate annidate all'interno dell'area di attivazione dei programmi chiamanti
- Il LINK_reg è comunque sempre usato dalle funzioni di Call e Return
- Alcune architetture non usano il LINK_reg, ma salvano l'indirizzo di rientro solo nella pila.
- Questa strategia è chiamata **pila a modo implicito**

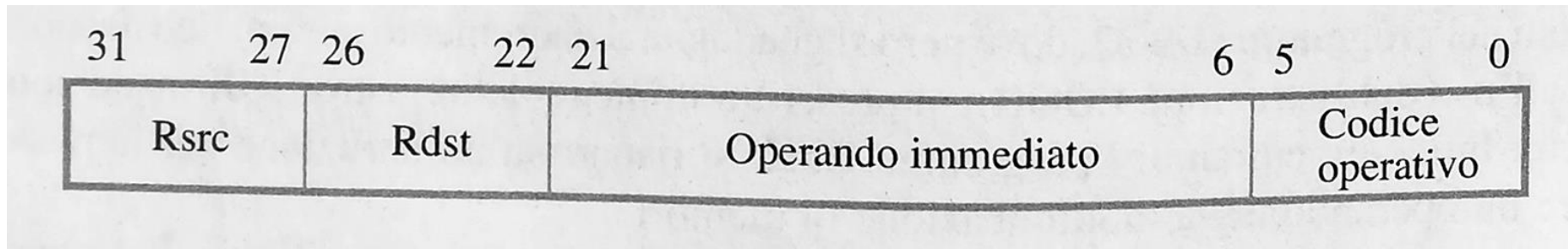
- Nello stile RISC le istruzioni macchina devono essere codificate in una parola di n bit
- A seconda delle istruzioni e del tipo di operandi usati esistono codifiche differenti
- Nel caso di CPU a 32 bit possono essere usate le seguenti codifiche:
 - Formato con operandi in registri
 - Formato con operando immediato
 - Formato per chiamata

- Nel caso si abbiano istruzioni a tre indirizzi di registro si assegnano 5 bit per ogni operando
- Esempio tipico sono le istruzioni aritmetiche e logiche
- I 17 bit meno significativi rappresentano il codice operativo dell'istruzione
- I 15 bit più significativi rappresentano gli indirizzi dei registri dei 3 operandi (5 bit per operando)

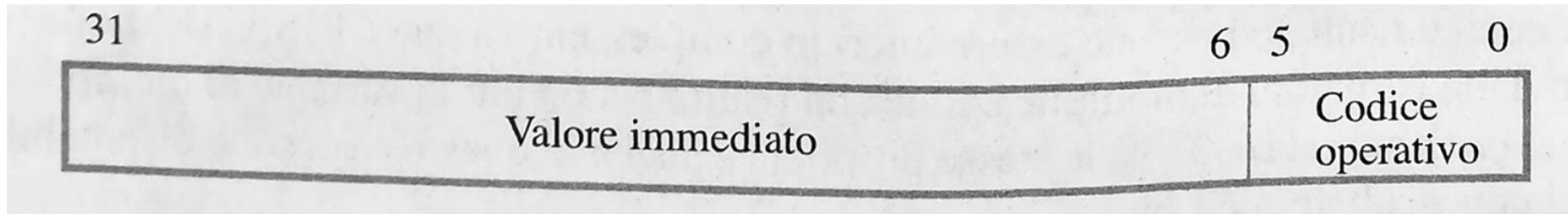


Istruzioni con operando immediato

- Caso in cui si hanno istruzioni a tre indirizzi in cui un operando è fornito tramite modo immediato oppure rappresenta un indirizzo di memoria
- Esempio tipico sono le istruzioni aritmetiche e logiche con operando immediato, load e store o funzioni di salto
- I 6 bit meno significativi rappresentano il codice operativo dell'istruzione
- I 16 seguenti rappresentano il valore immediato, l'indirizzo di memoria o lo spiazzamento
- I 10 bit più significativi rappresentano gli indirizzi dei registri



- Per le istruzioni di chiamata si può usare la seguente codifica:
- 26 bit per rappresentare l'indirizzo della prima istruzione del sottoprogramma
- 6 bit per il codice operativo dell'istruzione



- La notazione a trasferimento di registro (RTN) serve a descrivere formalmente il trasferimento di informazione tra parole di memoria e registri (processore e I/O).
- Il simbolo \leftarrow indica il trasferimento di valore
- Alla sinistra di \leftarrow si trova un indirizzo di memoria (simbolico o numerico) o il nome di un registro
- Alla destra di \leftarrow si trovano un valore semplice o un'espressione
- Per indicare il valore contenuto da un registro o da una parola di memoria si mette il nome o l'indirizzo tra parentesi quadre [.]
- Esempio somma del contenuto di due registri (A e B). Nel registro C viene memorizzata la somma:

$$C \leftarrow [A] + [B]$$

•Le istruzioni logiche in linguaggio macchina agiscono in modo bit a bit (bitwise)

•AND e OR hanno lo stesso formato delle operazioni aritmetiche viste finora:

AND **Rdst, Rsrc1, Rsrc2**

OR **Rdst, Rsrc1, Rsrc2**

•Possono essere usate in forma diretta:

AND **Rdst, Rsrc1, #Valore**

Dove #Valore è un valore logico a 16 bit esteso a 32 bit aggiungendo 16 zeri nelle posizioni più significative

• Il registro R2 contiene 4 caratteri ASCII da 8 bit. Stabilire se il carattere più a destra contiene la lettera Z (0x5A in esadecimale)

• Usare **And** per selezionare solo una parte del contenuto del registro

• Soluzione:

And R2, R2, #0xFF

Move R3, #0x5A

Branch_if_[R2]=[R3] TROVATOZ

•Le operazioni di scorrimento logico fanno scorrere i bit di un registro a destra o a sinistra di n posizioni

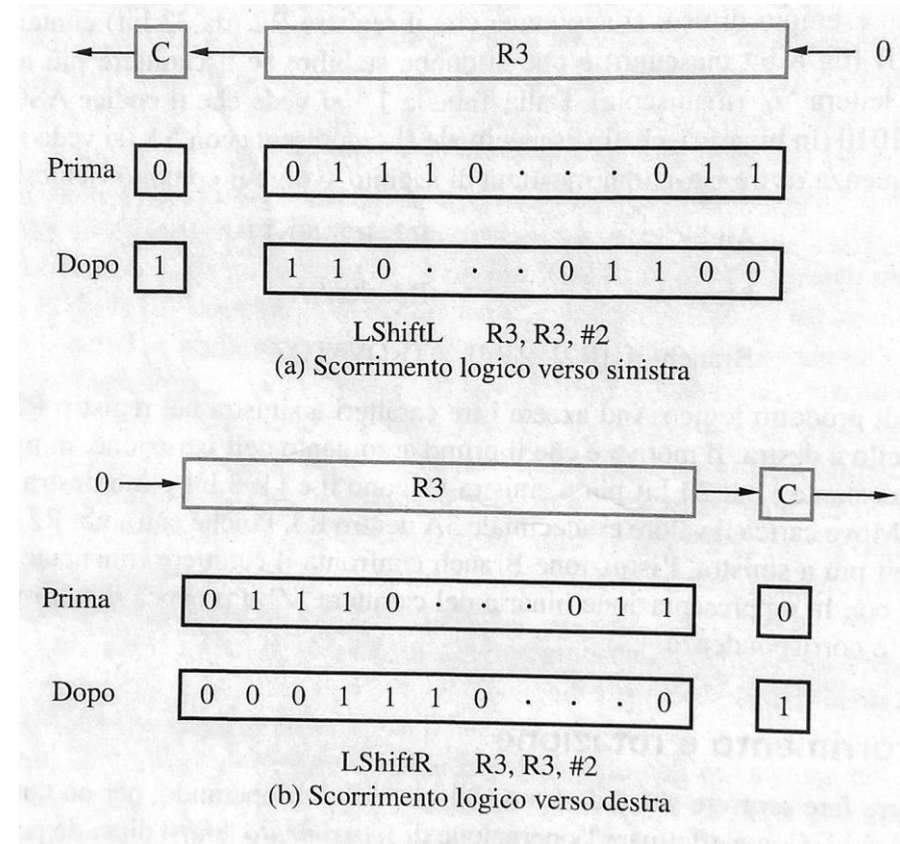
•I bit in uscita vengono persi, ad eccezione dell'ultimo bit che viene memorizzato nel bit di riporto c

•Le posizioni lasciate libere vengono riempite con 0

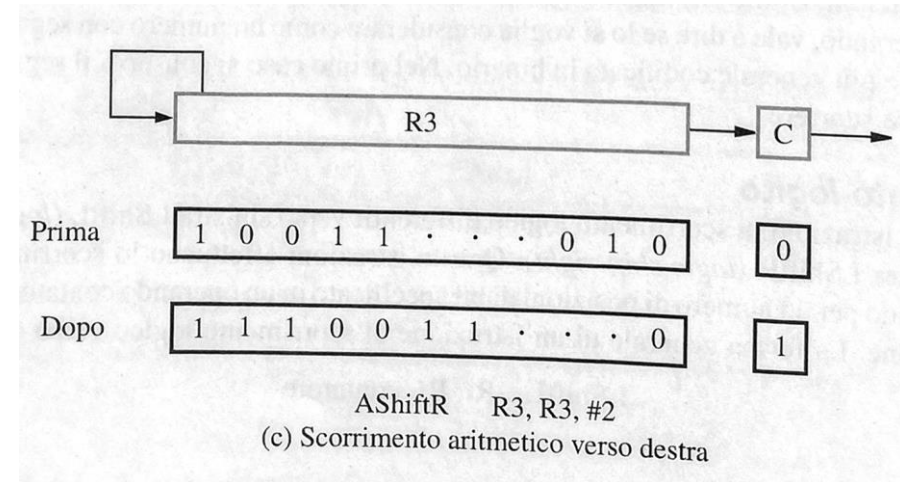
1)Primo operatore = registro destinazione

2)Secondo operatore = registro sorgente

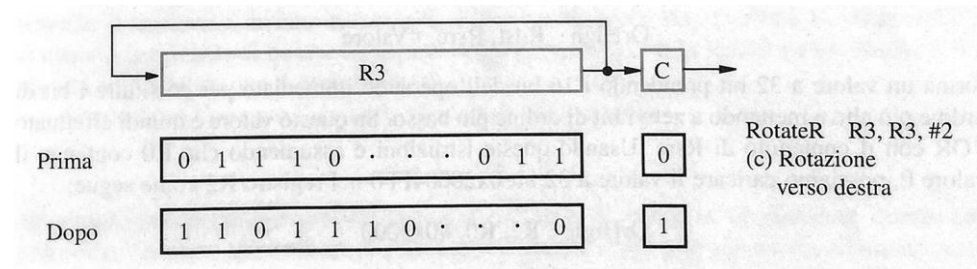
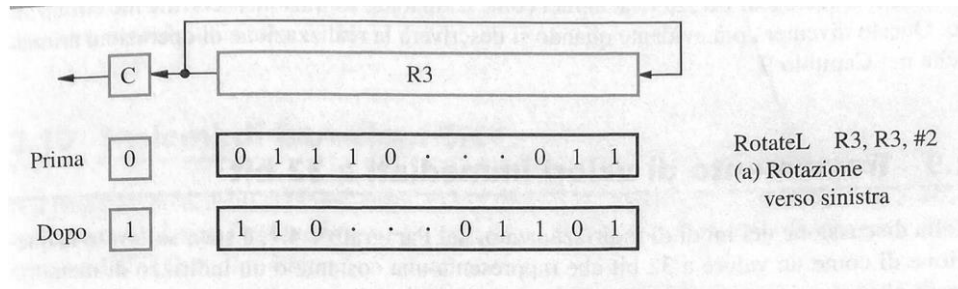
3)Terzo operatore = numero di bit da far scorrere



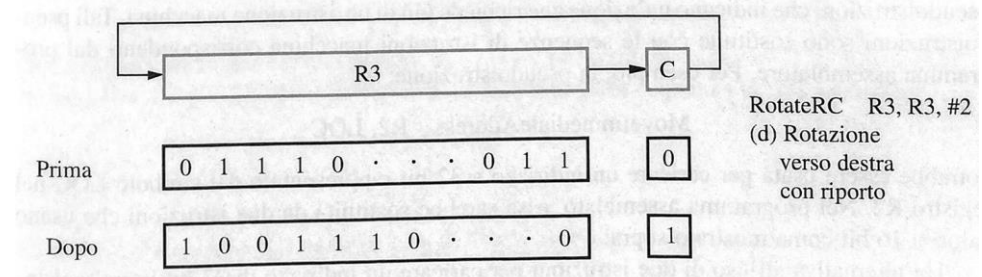
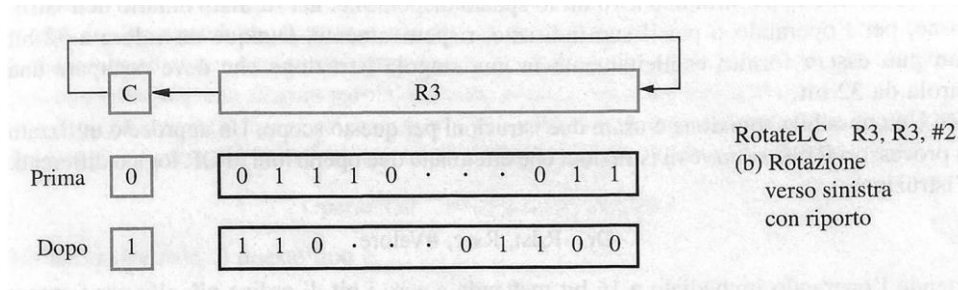
- Se il registro da scorrere contiene un numero in complemento a 2 bisogna preservare il bit di segno
- Lo scorrimento aritmetico verso destra funziona come quello logico, ma riempie le posizioni lasciate libere con il valore del bit più significativo
- Scorrimento logico e aritmetico verso sinistra sono uguali



- La rotazione è un'operazione di scorrimento dove i bit in uscita da un lato vengono fatti rientrare dall'altro
- L'ultimo bit in uscita viene scritto nel bit di riporto c



- Nella rotazione con riporto il bit di riporto viene incluso nella sequenza di bit da ruotare
- Nel caso di rotazione a sinistra il bit di riporto viene aggiunto alla sinistra dei bit del registro da ruotare
- Nel caso di rotazione a destra il bit di riporto viene aggiunto alla destra dei bit del registro da ruotare



- Spesso è necessario leggere/scrivere singoli byte dalla/nella memoria
- LoadByte legge un singolo byte dalla memoria e lo registra negli 8 bit meno significativi del registro destinazione mettendo a 0 gli altri bit

LoadByte Rdst, LOCBYTE

- StoreByte salva gli 8 bit meno significativi del registro sorgente nella locazione di memoria specificata

StoreByte Rsrc, LOCBYTE

•L'istruzione Multiply effettua la moltiplicazione tra due numeri in complemento a due contenuti in due registri.

Multiply **R_k, R_i, R_j**

- Occorrono $2n$ bit per rappresentare il prodotto
- Vengono salvati in R_k i bit meno significativi del prodotto e i più significativi non vengono calcolati
- In alcune architetture vengono registrati i bit più significativi in R_{k+1}

•L'istruzione Divide effettua la divisione intera tra due numeri in complemento a due contenuti in due registri.

Divide **R_k, R_i, R_j**

- Il quoziente di $[R_j]/[R_i]$ viene salvato in R_k e il resto non viene calcolato
- In alcune architetture il resto viene salvato nel registro R_{k+1}
- Non tutte le architetture posseggono le istruzioni di moltiplicazione e divisione

RISC:

- .Modi di indirizzamento semplici
- .Meno istruzioni, tutte occupanti una singola parola
- .Operazioni aritmetiche e logiche solo su registri
- .Non sono possibili trasferimenti diretti tra due locazioni di memoria
- .Possibile l'elaborazione a stadi
- .Programmi di dimensioni maggiori

CISC:

- .Modi di indirizzamento complessi
- .Tante istruzioni complesse, occupanti più parole di memoria
- .Operazioni aritmetiche e logiche con operandi sia su registri che locazioni di memoria
- .Trasferimenti diretti tra due locazioni di memoria tramite istruzione Move
- .Programmi di dimensioni ridotte

•L'istruzione Move permette il trasferimento di dati tra registri ed indirizzi di memoria

Move destinazione, sorgente

•L'operando destinazione può essere il nome di un registro o un indirizzo di memoria

•L'operando sorgente può essere il nome di un registro, un indirizzo di memoria o un valore immediato

•Nel caso che il sorgente sia un registro o un indirizzo di memoria l'istruzione esegue la seguente funzione espressa in RTN: **destinazione ← [sorgente]**

•Nel caso che il sorgente sia un valore immediato l'istruzione esegue la seguente funzione espressa in RTN: **destinazione ← sorgente**

.Tipicamente le istruzioni aritmetiche e logiche CISC usano un formato a 2 indirizzi, dove gli operandi possono essere sia registri che locazioni di memoria

Operazione destinazione, sorgente

.Nel caso di operazioni a due ingressi e un risultato, l'operando destinazione servirà sia da ingresso che da risultato

.Per esempio, l'operazione:

Add B, A

.Effettua l'operazione RTN: **B** \leftarrow **[A] + [B]**

- Nell'indirizzamento per autoincremento, l'indirizzo dell'operando è contenuto in un registro il cui nome viene specificato nell'istruzione. Alla fine dell'istruzione il contenuto del registro viene incrementato di un'unità
- L'unità di incremento solitamente è specificata nel nome dell'istruzione
- Sintassi simile al modo indiretto, nome del registro tra parentesi tonde, ma con l'aggiunta di un + alla fine: **(.)+**
- Può essere usata per eseguire l'operazione di Pop:

Move ELEMENTO, (SP)+

- Nell'indirizzamento per autoincremento, l'indirizzo dell'operando è contenuto in un registro il cui nome viene specificato nell'istruzione. Prima di eseguire l'istruzione, il contenuto del registro viene decrementato di un'unità
- L'unità di decremento solitamente è specificata nel nome dell'istruzione
- Sintassi simile al modo indiretto, nome del registro tra parentesi tonde, ma con l'aggiunta di un - all'inizio: **-(.)**
- Può essere usata per eseguire l'operazione di Push:

Move -(SP), NUOVOELEMENTO

- Modo di indirizzamento simile al modo per indice e spiazzamento, ma applicato registro PC (program counter)
- Sintassi identica a quella di indice e spiazzamento **X(PC)**, dove X è il valore di spiazzamento
- Utile per rappresentare indirizzi in dimensione ridotta (sempre relativi allo spiazzamento da PC)

•I **bit di esito o condizione** sono bit speciali immagazzinati in un registro interno al processore chiamato **registro di stato**.

•I bit di esito tengono traccia dell'esito di svariate operazioni, utili per valutare le condizioni di salto. Vengono aggiornati quando avviene un'operazione aritmetica e logica o trasferimento di dato.

•I bit di esito più comuni sono:

Bit di esito	Significato
N (negativo)	1 se risultato negativo, 0 se positivo o nullo
Z (zero)	1 se risultato nullo, 0 altrimenti
V (trabocco)	1 se trabocco in comp. a due, 0 altrimenti (oVerflow)
C (riporto)	1 se trabocco in binario naturale, 0 altrimenti (Carry)

Esempi RISC/CISC (Ricerca di una stringa)

.Si prendano due stringhe ASCII:

- 1) Stringa T lunga n
- 2) Stringa P lunga m (con $m < n$)

.Scrivere un programma assembly che determini se P è contenuta in T e trovi l'indice della prima occorrenza.

.Un programma in pseudocodice che risolve il problema in maniera brute-force può essere il seguente:

```
for   $i \leftarrow 0$  to  $n - m$  do
     $j \leftarrow 0$ 
    while  $j < m$  and  $P[j] = T[i + j]$  do
         $j \leftarrow j + 1$ 
    if  $j = m$  return  $i$ 
return  $-1$ 
```

Esempio Ricerca di una stringa (RISC)

	Move	R2, #T	R2 punta alla stringa <i>T</i>
	Move	R3, #P	R3 punta alla stringa <i>P</i>
	Load	R4, N	Preleva il valore <i>n</i>
	Load	R5, M	Preleva il valore <i>m</i>
	Subtract	R4, R4, R5	Calcola $n - m$
	Add	R4, R2, R4	Indirizzo di <i>T</i> ($n - m$)
	Add	R5, R3, R5	Indirizzo di <i>P</i> (<i>m</i>)
CICLO1:	Move	R6, R2	Usa R6 per scandire la stringa <i>T</i>
	Move	R7, R3	Usa R7 per scandire la stringa <i>P</i>
CICLO2:	LoadByte	R8, (R6)	Confronta una coppia di caratteri delle stringhe <i>T</i> e <i>P</i>
	LoadByte	R9, (R7)	
	Branch_if_[R8]≠[R9]	NONUGUALI	
	Add	R6, R6, #1	Punta al successivo carattere di <i>T</i>
	Add	R7, R7, #1	Punta al successivo carattere di <i>P</i>
	Branch_if_[R5]>[R7]	CICLO2	Reitera se non ancora finito
	Store	R2, RISULTATO	Immagazzina l'indirizzo di <i>T</i> (<i>i</i>)
	Branch	FATTO	
NONUGUALI:	Add	R2, R2, #1	Punta al successivo carattere di <i>T</i>
	Branch_if_[R4]≥[R2]	CICLO1	Reitera se non ancora finito
	Move	R8, # -1	Scrivi -1 per indicare che non è stata trovata alcuna corrispondenza
	Store	R8, RISULTATO	
FATTO:	prossima istruzione		

Esempio Ricerca di una stringa (CISC)



	Move	R2, #T	R2 punta alla stringa <i>T</i>
	Move	R3, #P	R3 punta alla stringa <i>P</i>
	Move	R4, N	Preleva il valore <i>n</i>
	Move	R5, M	Preleva il valore <i>m</i>
	Subtract	R4, R5	Calcola $n - m$
	Add	R4, R2	Indirizzo di <i>T</i> ($n - m$)
	Add	R5, R3	Indirizzo di <i>P</i> (<i>m</i>)
CICLO1:	Move	R6, R2	Usa R6 per scandire la stringa <i>T</i>
	Move	R7, R3	Usa R7 per scandire la stringa <i>P</i>
CICLO2:	MoveByte	R8, (R6)+	Confronta una coppia di caratteri delle stringhe <i>T</i> e <i>P</i>
	CompareByte	R8, (R7)+	
	Branch $\neq 0$	NONUGUALI	
	Compare	R5, R7	Confronta con l'indirizzo di <i>P</i> (<i>m</i>)
	Branch > 0	CICLO2	Reitera se non ancora finito
	Move	RISULTATO, R2	Immagazzina l'indirizzo di <i>T</i> (<i>i</i>)
	Branch	FATTO	
NONUGUALI:	Add	R2, #1	Punta al successivo carattere di <i>T</i>
	Compare	R4, R2	Confronta con l'indirizzo di <i>T</i> ($n - m$)
	Branch ≥ 0	CICLO1	Reitera se non ancora finito
	Move	RISULTATO, # -1	Non è stata trovata alcuna corrispondenza
FATTO:	prossima istruzione		