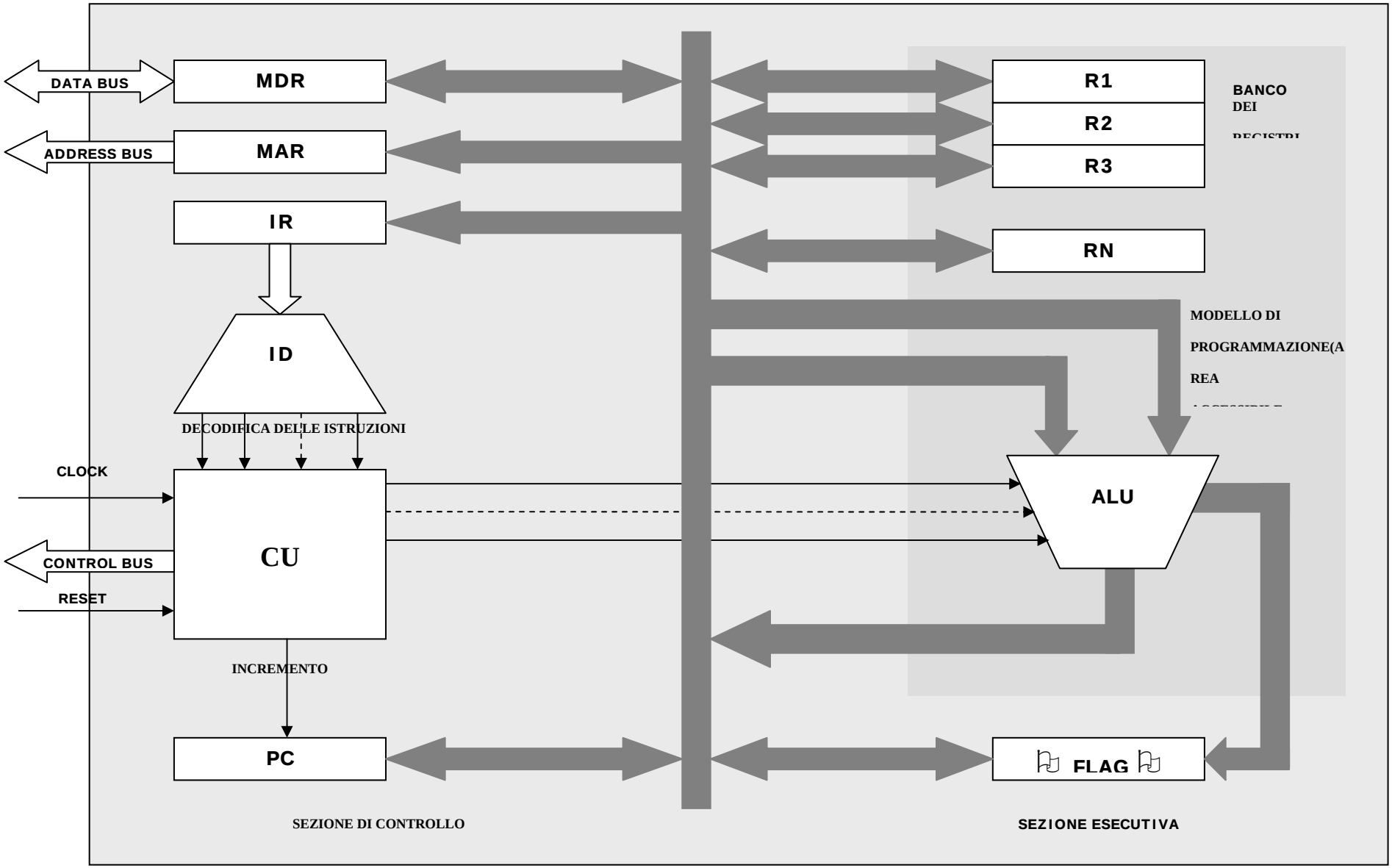

DATA BUS
MDR
ADDRESS BUS
MAR
IR
ID
DECODIFICA DELLE ISTRUZIONI
CLOCK
CONTROL BUS
RESET
CU
INCREMENTO
PC
R1
R2
R3
RN
BANCO
DEI
MODELLO DI
PROGRAMMAZIONE(A
REA
ALU
FLAG
SEZIONE DI CONTROLLO
SEZIONE ESECUTIVA

Architettura interna del microprocessore

La figura precedente mostra la struttura di principio di un microprocessore. Come si può notare essa si compone di più sottosistemi collegati da un bus interno. Questo bus consente lo scambio delle informazioni fra i vari sottosistemi e si interfaccia con il bus indirizzi ed il bus dati esterni mediante due registri tampone detti rispettivamente MAR (Memory Address Register) e MDR (Memory Data Register). Questi registri conservano gli indirizzi ed i dati da scambiare con l'esterno e sono invisibili al programmatore nel senso che questo non ne può influenzare il comportamento mediante l'esecuzione di istruzioni. Nella parte sinistra della figura è poi simboleggiato il cervello del microprocessore che è l'unità di controllo ed esecuzione. Tale unità si compone del registro IR nel quale viene depositato il codice operativo dell'istruzione prelevato dalla memoria (è quella parte dell'istruzione che dice cosa occorre fare). Il codice operativo è, come ogni dato elaborato dal microprocessore, una stringa di bit che va interpretata per comprendere quali operazioni vengono richieste al microprocessore. A tale scopo viene passato al blocco

ID (Instruction Decoder) il quale decodifica l'istruzione e passa l'informazione alla unita di controllo (Control Unit CU) che si occupa di generare i segnali di controllo che costringono le varie unità del microprocessore a cooperare per eseguire l'istruzione.

Per poter operare il microprocessore ha poi bisogno di aree in cui memorizzare le informazioni: i registri.

Il termine registro indica un blocco di memoria, equivalente nella sua struttura ad una cella di memoria ma contenuto all'interno della CPU. Le dimensioni (in bit) dipendono dal tipo di CPU.

La funzione dei registri è di realizzare una memorizzazione temporaneo di una informazione, per il solo tempo necessario per portare a termine l'operazione in cui l'informazione è coinvolta. Si può quindi immaginare un registro come una lavagna su cui viene scritta una informazione per la durata della loro spiegazione e poi viene cancellata quando si passa alla spiegazione successiva; se si vuole che l'informazione sia conservata permanentemente è necessario salvarla da qualche altra parte.

Diversamente dalle celle di memoria i registri non hanno un indirizzo (non sono in memoria centrale ma dentro alla CPU); sono invece identificati da

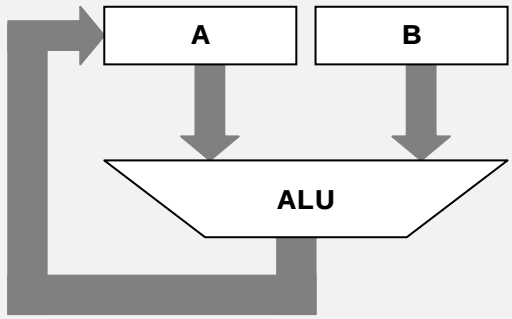
un nome. Alcuni registri sono accessibili al programmatore attraverso il loro nome mentre altri sono invisibili e servono per attuare le azioni di controllo.

Alcuni registri sono specializzati, cioè svolgono solo una specifica funzione mentre altri sono di uso generale, possono cioè essere usati dal programmatore per un qualsiasi scopo.

Arithmetic Logic Unit: è il blocco che esegue le trasformazioni sui dati eseguendo operazioni logiche (AND, OR, XOR) e d aritmetiche (somma, sottrazione, confronto, ecc.). Poiché i dati, indipendentemente dal loro significato sono codificati attraverso numeri binari qualsiasi trasformazione da fare sui dati si riduce ad una operazione aritmetica o logica. Opera su due operandi che possono essere contenuti nei registri interni o provenire dalla memoria.

La ALU si comporta in genere da "operatore binario ad accumulo". Il termine "binario" in questo caso non sta ad indicare che i dati sono in codice binario ma che la ALU esegue le operazioni a partire da due operandi. Il termine "accumulo" sta ad indicare che il risultato

dell'operazione va a finire in uno dei due operandi. In questo modo nell'operando che è dapprima origine e poi destinazione dei dati si forma un accumulo di dati.



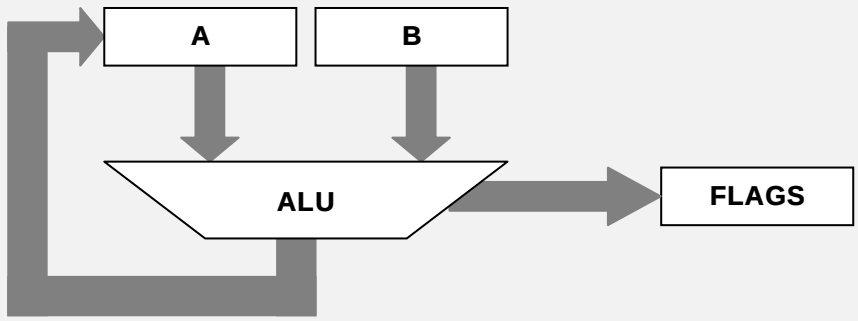
Oltre al risultato della operazione la ALU produce anche altre informazioni logiche dette FLAGS. Queste informazioni formano un pacchetto di bit che riflettono con il loro stato informazioni sul risultato della ultima operazione aritmetico logica eseguita.

Quindi questi bit si aggiornano ad ogni operazione aritmetico/logica mentre restano inalterati al valore della ultima operazione aritmetico/logica mentre la CPU fa altri tipi di operazione che non coivolgono la ALU (ad esempio trasferimenti sul bus).

Queste informazioni sono essenziali per la costruzione degli algoritmi dei programmi. Infatti spesso il programma deve fare delle scelte in

funzione dei risultati che si sono avuti. Il programma deve trovare allora il modo di verificare cosa è successo durante le elaborazioni precedenti e ciò avviene interrogando il registro dei flag.

Lo schema completo della ALU quindi diventa:



I principali flags sono:

- ZERO (ZF): è vero quando il risultato dell'ultima operazione vale 0 mentre è falso quando il risultato dell'operazione è diverso da 0. E' il flag più usato.
- CARRY (CF): è vero quando c'è stato un riporto dal bit più significativo del risultato. Le operazioni di somma e le operarazioni derivate da

questa (sottrazione, moltiplicazione ...) possono determinare un riporto da un bit al bit di peso successivo (vedere FULL ADDER). Se la somma del bit più significativo determina un riporto questo bit non può essere sommato al bit successivo e viene memorizzato nel CF per segnalare che l'operazione nel suo complesso ha determinato un riporto.

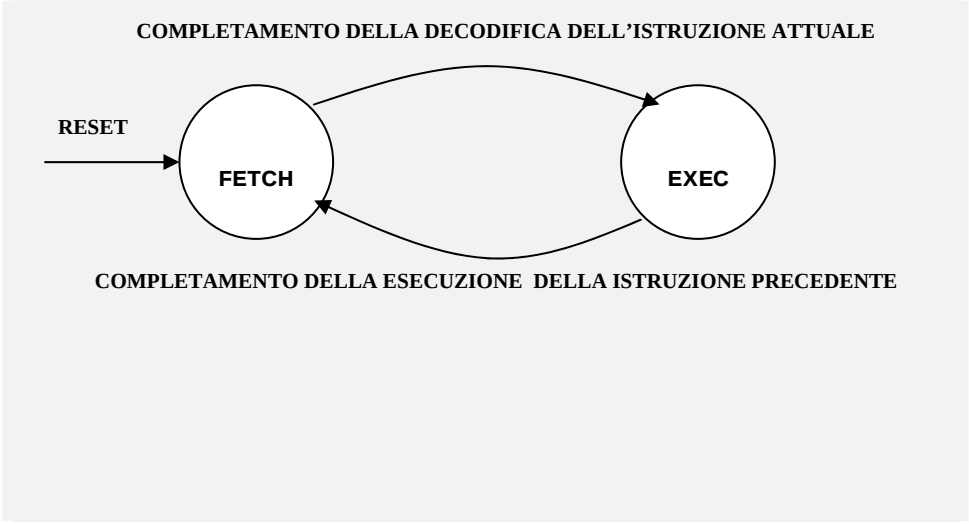
- SIGN (SF): è vero quando il risultato è negativo. Materialmente viene riportato nel SF il bit più significativo del risultato che riflette il segno.
- OVERFLOW(OF): è vero se il segno del risultato discorda dal segno degli operandi. Segnala la presenza di un overflow rispetto alla precisione; infatti se si sommano due operandi positivi il risultato dovrebbe essere sempre positivo, se il bit più significativo del risultato è negativo significa che il risultato ha un'ampiezza superiore alla capacità del risultato; la stessa cosa vale per due numeri negativi mentre se i due numeri hanno segni discordi un overflow non si può verificare quindi OF diventa sempre falso.

Program Counter: è un registro interno, che contiene in ogni fase di avanzamento del programma l'indirizzo di memoria in cui si trova la

prossima istruzione da eseguire.

Viene incrementato automaticamente dalla CU (Control Unit) ogni volta che l'esecuzione di una istruzione è completata in modo da potere leggere in memoria l'istruzione successiva.

Per comprendere il funzionamento della CPU conviene partire da una automa a stati finiti molto semplice formato da due soli stati: FETCH ed EXECUTE.



Si può immaginare la CPU come un automa a stati finiti che si muove continuamente tra i due suoi stati:

FETCH (INTERPRETAZIONE): Supponendo che PC stia puntando al codice operativo di una istruzione memorizzato in memoria centrale la CU comanda, attraverso i bus, la lettura dalla memoria del codice operativo

(OPCODE) della istruzione; questa informazione attraverso il MDR arriva all' IR che lo manda al circuito di decodifica ID per ottenere la decodifica della istruzione. Inoltre CU incrementa il PC in modo che al prossimo FETCH punti all'istruzione successiva e passa allo stato di EXECUTE.

EXECUTE (ESECUZIONE): Si entra in stato EXECUTE con l'evento "Completamento della decodifica della istruzione attuale". Ottenuta la decodifica dell'istruzione la CU comanda ad ALU la parte esecutiva che può coinvolgere sia trasferimenti di bus (lettura di dati dalla memoria, scrittura di dati in memoria, acquisizione di dati dall'ingresso, emissione di dati sull'uscita) in cui l'altro estremo del trasferimento è un registro interno della CPU sia elaborazioni interne fra registri eseguite dalla ALU che portano a risultati memorizzati sui registri. Esistono anche elaborazioni miste che coinvolgono contemporaneamente entrambe le attività (ad esempio una somma tra il contenuto di un registro ed il contenuto di una cella di memoria che viene reso disponibile sul MDR mediante una lettura con risultato in un registro). Al termine della esecuzione l'automa si riporta in FETCH per interpretare l'istruzione

successiva il cui indirizzo si trova in PC.

Stack e nidificazione dei sottoprogrammi

Cos'è un sottoprogramma?

Nella programmazione è frequente il ricorso all'utilizzo di sottoprogrammi. Vediamo cos'è un sottoprogramma. Facciamolo con un esempio. Supponiamo di voler scrivere un programma che si occupi di risolvere una equazione di secondo grado con la formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Occupiamoci della parte del programma che si occupa del calcolo del delta. Supponiamo in particolare che stiamo scrivendo il programma in assembly per un microprocessore generico che abbia una serie di registri di utilizzo generale denominati da A a G, che questo microprocessore comprenda istruzioni tipo LEGGI, SPOSTA e così via e che ogni istruzione si traduca in una stringa di bit di lunghezza pari a quella delle locazioni di memoria utilizzate dal microprocessore. Supponiamo inoltre che i parametri dell'equazione a, b, c siano conservati già nei registri del microprocessore denominati C, D, E. il programma per il calcolo del delta sarà allora il seguente

1. **COPIA IL CONTENUTO DI D IN A** [1]
2. **COPIA A IN B**[2]
3. DEC B[3]
4. COPIA A IN F [4]
5. SOMMA A,F
6. DECREMENTA B[5]
7. SE B <> 0 TORNA ALL'ISTRUZIONE 5[6]

---

[1] il contenuto del registro D, cioè il parametro b viene spostato nell'accumulatore

[2] il registro B verrà usato come contatore nel segmento di programma che effettua il quadrato di b mediante una serie di somme

[3] il contenuto del contatore va decrementato inizialmente. Infatti se dobbiamo fare il quadrato, ad esempio, del numero 8, dobbiamo fare tante somme 8+8=16, 16+8=24, 24+8=32 e così via e questo 7 volte; in generale il quadrato del numero N si effettuerà effettuando N-1 somme parziali

[4] per effettuare la moltiplicazione abbiamo bisogno di un registro che conterrà le somme parziali (16, 24, 32, eccetera) e questo sarà il registro A, ed un registro che conterrà sempre il numero di cui si vuol fare il quadrato (nell'esempio 8) e questo sarà il registro F

[5] ad ogni somma decremento B, quando questo arriva a zero, vuol dire che ho finito di fare le somme e in A vi è il risultato del quadrato

8. COPIA A IN G[7]

9. **COPIA C IN A[8]**

10. **COPIA E IN B[9]**

11. DEC B

12. COPIA A IN F

13. SOMMA A,F

14. DECREMENTA B

15. SE B <> 0 TORNA ALL'ISTRUZIONE 13 [10]

16. COPIA 4 IN B

17. DEC B

---

[6] se il contatore non si è azzerato torna ad eseguire la somma altrimenti vai avanti

[7] il registro A mi serve di nuovo per cui il risultato del quadrato di b viene parcheggiato in G

[8] ora devo fare il prodotto a*c e seguirò la stessa logica precedente, il parametro a va spostato in A

[9] per fare il prodotto di a per c, devo fare tante somme che coinvolgono a per un numero di volte pari a c-1, ad esempio il prodotto 7*5 si ottiene facendo le somme 7+7=14, 14+7=21, 21+7=28 eccetera per 5-1 =4 volte

[10] ora devo fare il prodotto fra il valore a*b che si trova nell'accumulatore A e il numero 4

18. *COPIA A IN F*

19. *SOMMA A,F*

20. *DECREMENTA B*

21. *SE B <> 0 TORNA ALL'ISTRUZIONE 19*

Notate ora che nel programma abbiamo dovuto risolvere per tre volte il problema della moltiplicazione, per cui, al di là delle istruzioni che ci consentivano di selezionare gli operandi della moltiplicazione (e che ho evidenziato in grassetto) ci sono alcune istruzioni (in corsivo sottolineato) che si ripetono sempre.

Immaginiamo di prendere queste istruzioni e racchiuderle in un programma a parte

1. DEC B ; sottoprogramma MOLTIPLICA
2. COPIA A IN F
3. SOMMA A,F
4. DECREMENTA B
5. SE B <> 0 TORNA ALL'ISTRUZIONE 3

Abbiamo costruito un programma che effettua il prodotto fra il numero presente nel registro A e il numero presente nel registro B.

L'idea è ora la seguente: quando un programma si trova di fronte al compito di effettuare la moltiplicazione, in luogo di svilupparla direttamente farà riferimento al programma MOLTIPLICA, affidando ad esso tale compito. Il programma principale si deve soltanto occupare di porre un moltiplicando nel registro A ed uno nel registro B. Se disponiamo di un'istruzione del tipo CHIAMA sottoprogramma il programma principale diventa semplicemente

1. COPIA IL CONTENUTO DI D IN A
2. COPIA A IN B
3. CHIAMA MOLTIPLICA
4. COPIA A IN G
5. COPIA C IN A
6. COPIA E IN B
7. CHIAMA MOLTIPLICA
8. COPIA 4 IN B
9. CHIAMA MOLTIPLICA

Vediamo che abbiamo tutta una serie di vantaggi

- Il programma principale è più semplice

- Se ho già una volta affrontato il problema risolto dal sottoprogramma (nell'esempio la moltiplicazione fra due numeri) non devo risolverlo di nuovo ma applicare nel programma principale il sottoprogramma che ho scritto precedentemente
- Il programma principale è più corto ed occupa uno spazio minore in memoria
- Lo sviluppo di un programma può essere modularizzato: una persona si può occupare del programma principale e lasciare ad altri lo sviluppo dei sottoprogrammi

La chiamata di un sottoprogramma

Ma come avviene la chiamata di un sottoprogramma? Dobbiamo ricordare che le varie istruzioni sono racchiuse in locazioni di memoria

100	COPIA IL CONTENUTO DI D IN A
101	COPIA A IN B
102	CHIAMA MOLTIPLICA
103	COPIA A IN G
104	COPIA C IN A
105	COPIA E IN B
106	CHIAMA MOLTIPLICA
107	COPIA 4 IN B
108	CHIAMA MOLTIPLICA

Nell'ipotesi che ogni istruzione occupi esattamente una locazione di memoria la situazione è quella rappresentata nella tabella precedente dove abbiamo supposto che il programma occupi 9 locazioni di memoria a partire dall'indirizzo 100. il sottoprogramma MOLTIPLICA si troverà invece in un'altra zona di memoria a partire dall'indirizzo 200 (è un esempio).

200	DEC B
201	COPIA A IN F
202	SOMMA A,F
203	DECREMENTA B
204	SE B <> 0 TORNA ALL'ISTRUZIONE 3

Il programma principale viene svolto nella giusta sequenza grazie al Program Counter che, in ogni fase di fetch dell'istruzione, si incrementa automaticamente di 1.

Quando il PC conterrà l'indirizzo 102 verrà effettuato il fetch dell'istruzione CHIAMA MOLTIPLICA, automaticamente il PC si incrementa al valore 103. Cosa deve avvenire nella fase di esecuzione? Se ci pensiamo bene, chiamare il sottoprogramma significa semplicemente che non deve essere prelevata l'istruzione all'indirizzo 103 ma quella all'indirizzo 200, e continuare ad eseguire le istruzioni da questo punto in poi della memoria. Quindi l'istruzione CHIAMA non deve far altro che sostituire, all'interno del PC l'indirizzo 103 con l'indirizzo 200.

C'è però un particolare: quando termina l'esecuzione del programma MOLTIPLICA, si deve ritornare al programma principale dall'indirizzo 103

in poi.

Ogni sottoprogramma termina con un'istruzione di RITORNO che deve garantire il ritorno corretto all'esecuzione della parte restante del programma principale. Ciò avviene perché l'indirizzo che era contenuto nel PC, prima dell'esecuzione del salto, (nel nostro esempio 103) viene salvato in una zona di memoria detta STACK[11], prima di essere sostituito dall'indirizzo della prima istruzione del sottoprogramma (nel nostro esempio 200). L'istruzione RITORNO deve, invece, prelevare l'indirizzo di rientro dallo STACK e inserirlo nel PC. In tal modo, in maniera automatica il microprocessore tornerà ad eseguire il programma principale.

Lo stack

Lo stack è un'area di memoria particolare ricavata nella RAM collegata al microprocessore.

In generale le memorie RAM hanno una organizzazione di tipo ad accesso casuale. Questa dizione sta a significare che il microprocessore può accedere (leggere o scrivere) a qualunque locazione della memoria senza limiti.

[11] Tale indirizzo prende il nome di **indirizzo di rientro**

Lo stack ha invece un'organizzazione detta LIFO (last in first out: l'ultimo ad entrare è il primo ad uscire). Il microprocessore, in tal caso, non può accedere ad una qualunque delle locazioni di memoria dello stack, ma può soltanto accedere alla cima dello stack. Immaginatevi, ad esempio, una catasta di libri: non potete tirare un libro dal centro della catasta, ma potete prendere soltanto quello che si trova in cima così come potete depositarne uno soltanto in cima alla catasta.

Supponiamo, ad esempio, uno stack costituito da 5 locazioni, e che esso sia inizialmente vuoto, a partire dall'indirizzo 100.

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto
103	Vuoto
104	Vuoto

Se inseriamo un dato, esso verrà necessariamente posto nella cima dello stack che è la locazione 104[12]

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto
103	Vuoto
104	Primo dato

La locazione 104 viene detta cima dello stack. Se inseriamo un secondo dato, esso va posto necessariamente nella locazione che viene subito sopra cioè la 103

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto
103	Secondo dato

[12] di solito uno stack è organizzato in modo da crescere verso gli indirizzi più bassi: la prima locazione è la 104, l'ultima è la 100

104	Primo dato

Adesso la cima dello stack è la locazione 103.

Se inseriamo un terzo dato, esso dovrà essere inserito necessariamente nella locazione 102

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Terzo dato
103	Secondo dato
104	Primo dato

Quando vogliamo prelevare dati, invece, possiamo soltanto prelevare il dato in cima allo stack (nell'esempio il contenuto della locazione 103): non possiamo accedere, ad esempio, alla locazione 103 o 104. Dopo il prelievo la situazione è la seguente

Indirizzo	Locazione
100	Vuoto

101	Vuoto
102	Vuoto
103	Secondo dato
104	Primo dato

La cima dello stack è ora la 103. Soltanto adesso possiamo prelevare il secondo dato. In ogni caso non possiamo accedere alla locazione 104. Soltanto se preleviamo il secondo dato possiamo accedere al primo dato.

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto
103	Vuoto
104	Primo dato

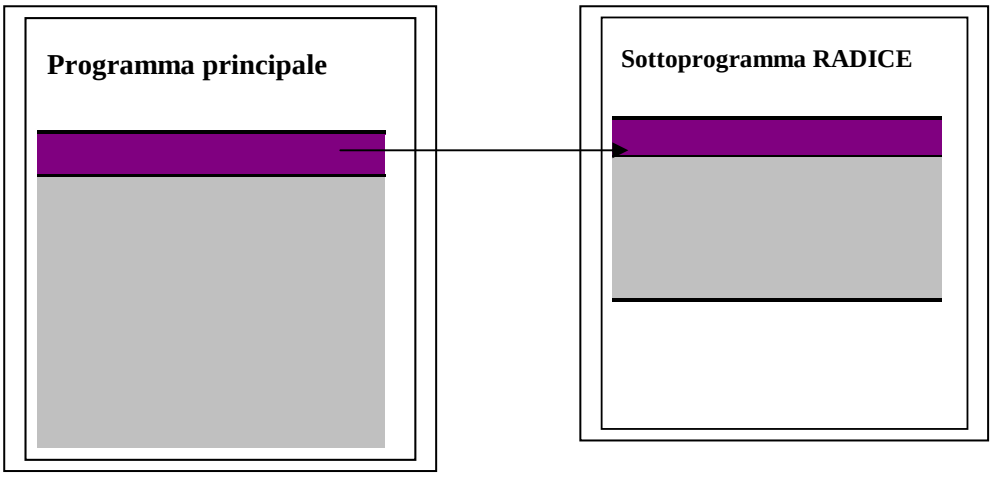
Il nesting dei sottoprogrammi.

Perché abbiamo bisogno di una struttura come lo stack? Esso è necessario

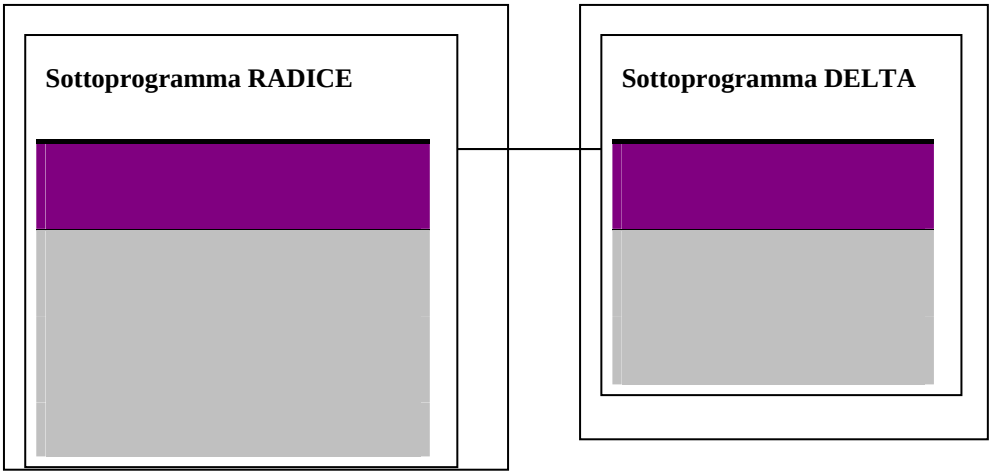
per realizzare il nesting o nidificazione dei sottoprogrammi.

Questo concetto significa semplicemente che spesso un programma chiama un sottoprogramma ma questi, a sua volta, per l'esecuzione di compiti specifici chiama un altro sottoprogramma.

Ad esempio, il programma che risolve l'equazione di secondo grado chiama un sottoprogramma che fa la radice quadrata del delta

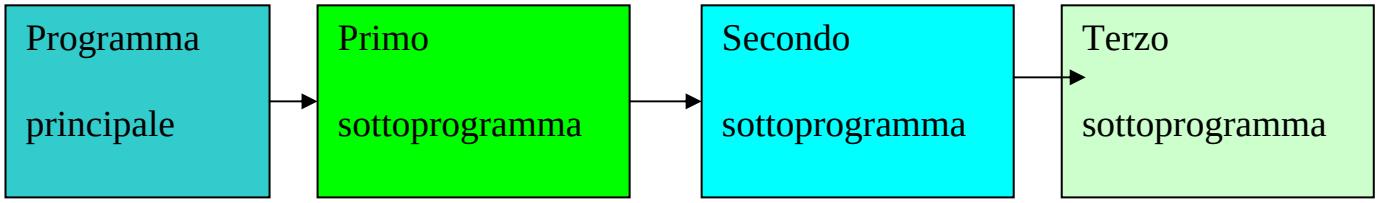


Il sottoprogramma radice chiama a sua volta il sottoprogramma che calcola il delta

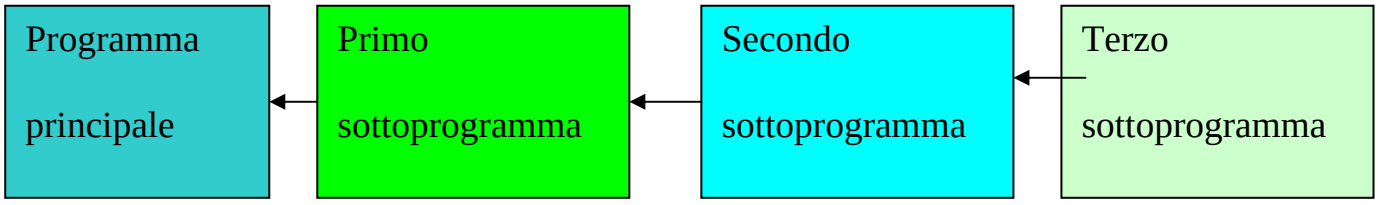


Il sottoprogramma delta può chiamare un altro sottoprogramma, ecc.

Il problema del nesting dei sottoprogrammi è che si deve poter ritornare indietro nell'ordine giusto. Supponiamo di aver questo caso



Quando il terzo sottoprogramma termina occorre che il controllo passi al secondo sottoprogramma; quando questo termina occorre passare a completare il primo e soltanto quando questo termina si passa a terminare il programma principale



Ciò è garantito da un meccanismo automatico basato sullo stack. Infatti quando si passa dal programma principale al primo sottoprogramma, l'indirizzo di rientro del programma principale viene salvato nello stack

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto
103	Vuoto
104	Indirizzo di rientro del programma principale

Quando il primo sottoprogramma chiama il secondo sottoprogramma, nello stack viene salvato l'indirizzo di rientro del primo sottoprogramma

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto

103	Indirizzo di rientro del primo sottoprogramma
104	Indirizzo di rientro del programma principale

Quando il secondo sottoprogramma chiama il secondo sottoprogramma, nello stack viene salvato l'indirizzo di rientro del secondo sottoprogramma

Tabella 3

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Indirizzo di rientro del secondo sottoprogramma
103	Indirizzo di rientro del primo sottoprogramma
104	Indirizzo di rientro del programma principale

IL terzo sottoprogramma termina con un'istruzione di ritorno, quest'ultima non fa nient'altro che accedere alla cima dello stack , prelevarne il contenuto e metterlo nel Program Counter. Automaticamente si ritorna ad eseguire il secondo sottoprogramma. Però poiché si è effettuato un prelievo dallo stack, la cima dello stack si è spostata all'indirizzo 103

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto
103	Indirizzo di rientro del primo sottoprogramma
104	Indirizzo di rientro del programma principale

Quando termina il secondo sottoprogramma, viene eseguita di nuovo un'istruzione di ritorno, la quale va in cima allo stack e automaticamente preleva l'indirizzo di rientro del primo sottoprogramma. Ora la cima dello stack è la locazione 103

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto
103	Vuoto

104	Indirizzo di rientro del programma principale

Al termine del primo sottoprogramma verrà prelevato dalla cima dello stack l'indirizzo di rientro del programma principale.

Lo stack pointer

Lo stack è una struttura dinamica, che cresce e diminuisce come abbiamo visto dagli esempi precedenti. Occorre allora un meccanismo per informare il microprocessore sulla posizione in cui si trova la cima dello stack. Questo meccanismo si basa su un registro del microprocessore che si chiama Stack pointer. Tale registro conterrà sempre l'indirizzo della cima dello stack.

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto
103	Vuoto
104	Vuoto

Quando lo stack è vuoto lo SP contiene l'indirizzo 105. Se volgiamo inserire un dato nello stack esso andrà inserito nella locazione di memoria che si trova all'indirizzo formato dal contenuto dello SP –1 cioè 104

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto
103	Vuoto
104	Primo dato

Poiché lo stack è cresciuto, ora lo SP deve essere decrementato a 104. Un nuovo dato va inserito all'indirizzo SP-1 cioè 103

Indirizzo	Locazione
100	Vuoto
101	Vuoto
102	Vuoto
103	Secondo dato
104	Primo dato

LO SP viene portato a 103

Il terzo dato viene inserito all'indirizzo 102 e così via. Un meccanismo opposto si avrà quando i dati vengono prelevati.