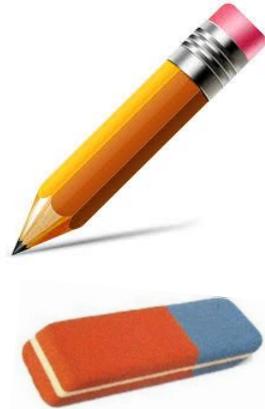
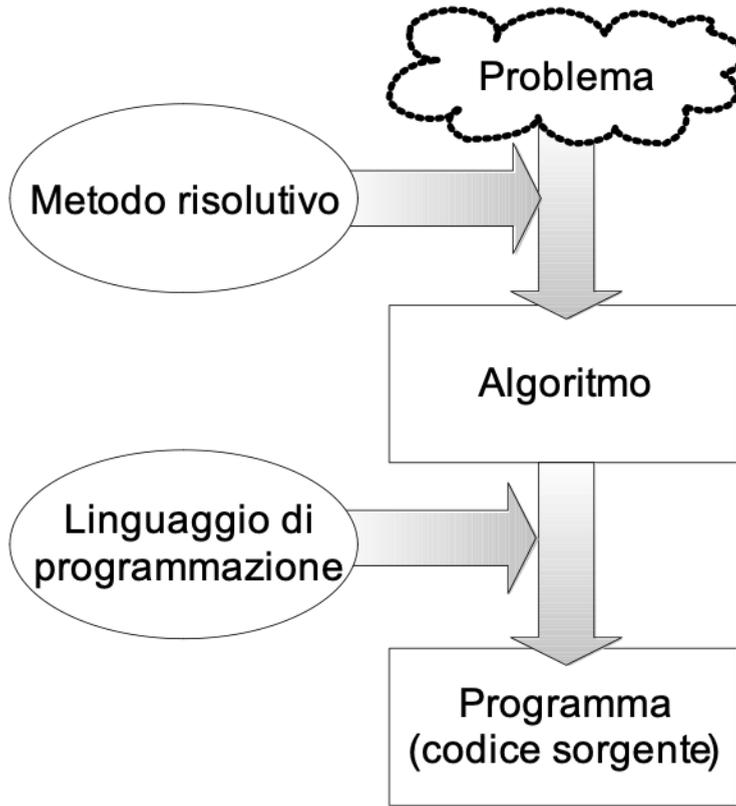
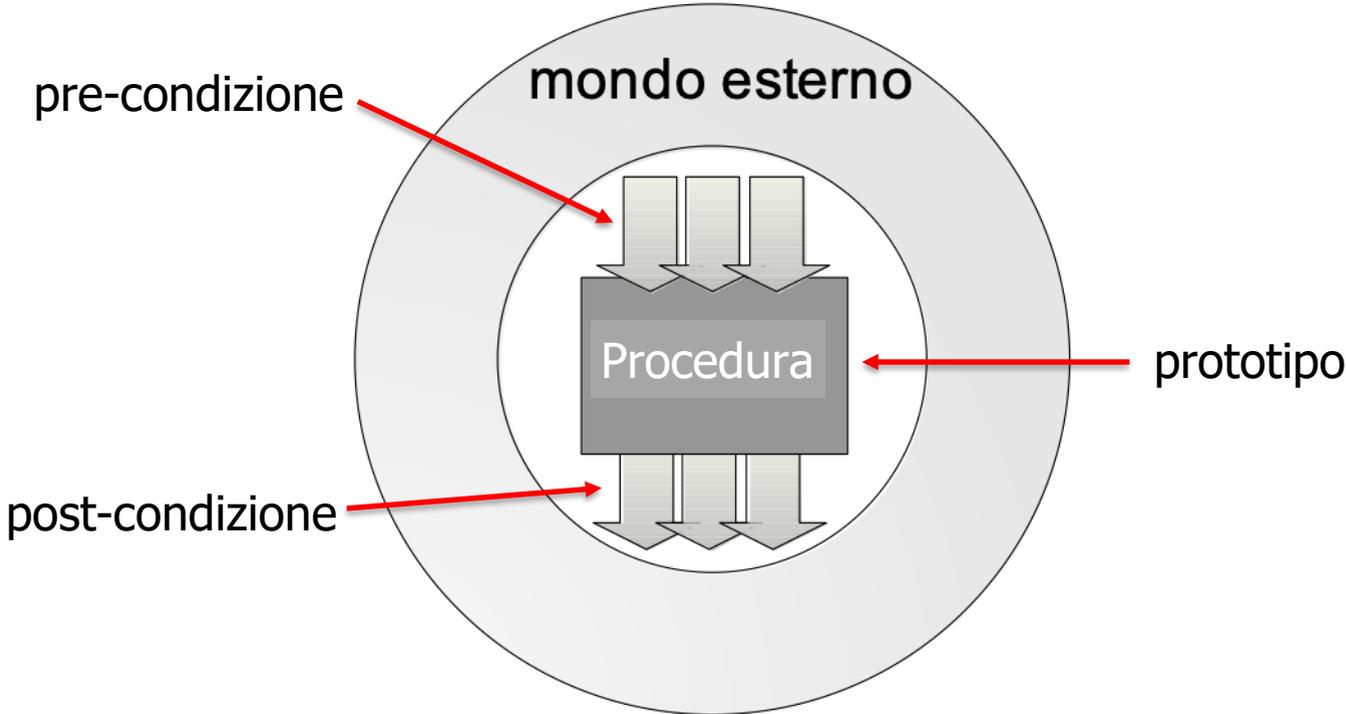


La programmazione in linguaggio Assembly ARM

Passi da seguire



Il «contratto»



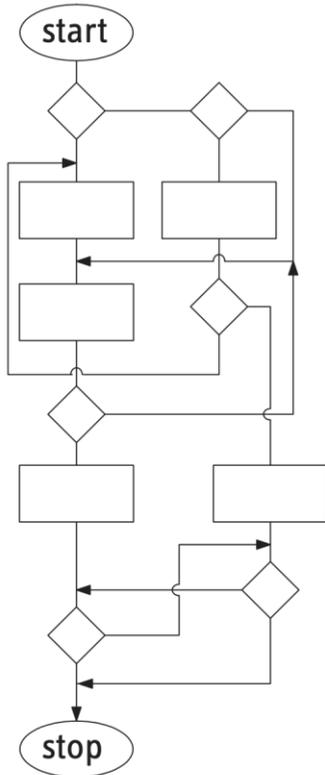
Teorema di Böhm-Jacopini

Il teorema di Böhm-Jacopini, enunciato nel 1966 da due informatici italiani dai quale prende il nome, afferma che:

«qualunque algoritmo può essere implementato utilizzando tre sole strutture, la **sequenza**, la **selezione** e il **iterazione**, che possono essere tra loro innestati fino a giungere ad un qualsivoglia livello di profondità finito (come le scatole cinesi)»

In altre parole, qualsiasi procedimento risolutivo non strutturato («codice a spaghetti») può essere trasformato in un equivalente algoritmo strutturato.

L'aspetto complementare



...d'altra parte un algoritmo destrutturato potrebbe non essere traducibile in un linguaggio di programmazione imperativo in assenza di istruzioni di salto incondizionato (goto, break, exit, end, loop,...)

DNS – Diagrammi Nassi-Shneiderman



Isaac "Ike" Nassi

<http://www.nassi.com/ike.html>



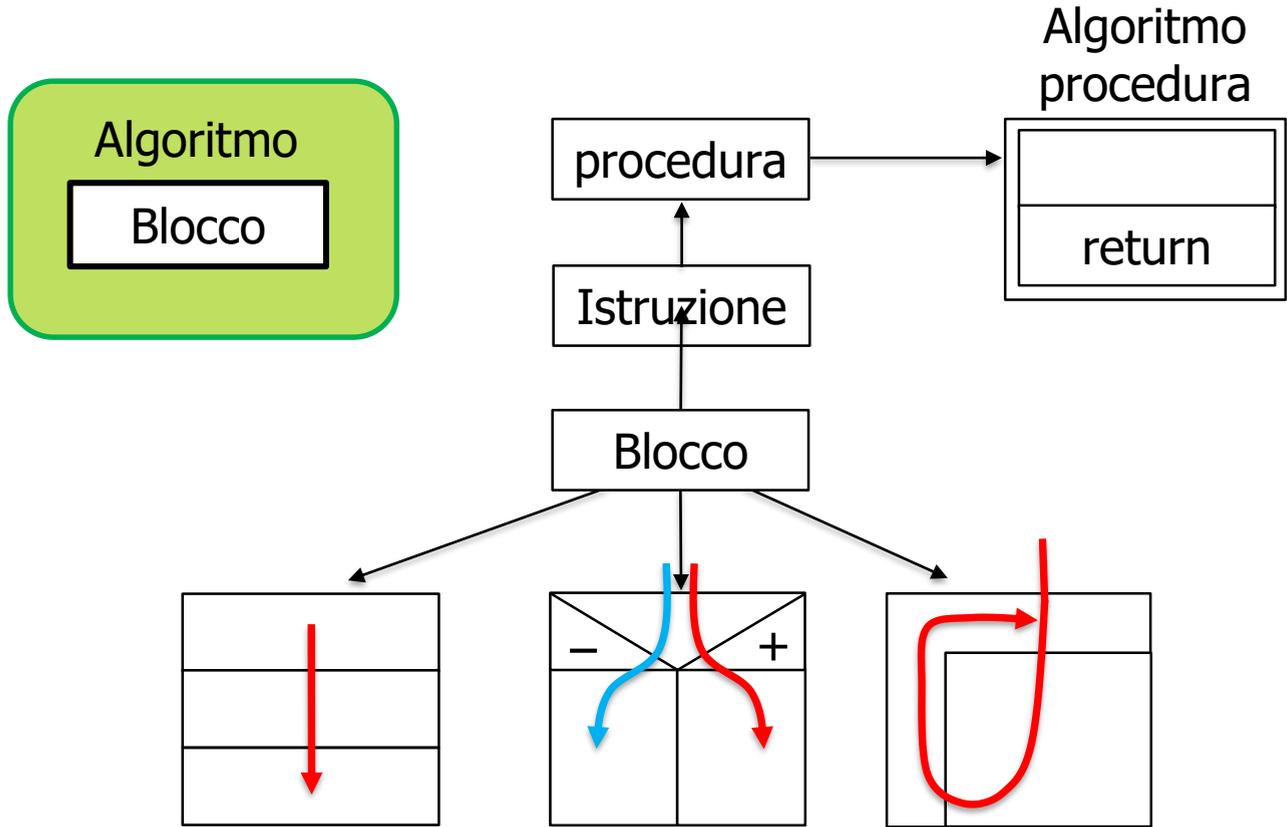
Ben Shneiderman

<http://www.cs.umd.edu/~ben/>

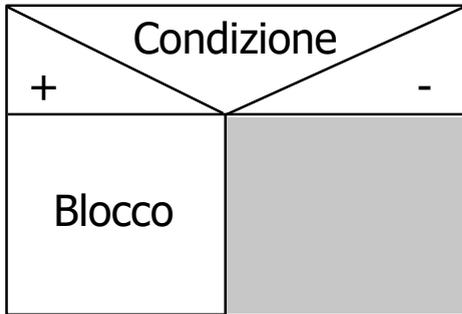
DNS – Diagrammi Nassi-Shneiderman

- sono un valido strumento didattico per scrivere codice strutturato
- hanno intrinsecamente il concetto di indentazione del codice quindi ne aumentano la leggibilità
- nascono con il linguaggio Pascal ma ne possono essere utilizzati in tutti i linguaggi procedurali (C++, java o assembly)
- si integrano perfettamente nelle metodologie di progettazione del software (es. UML) e costituiscono una valida alternativa ai diagrammi di flusso (che generano la programmazione "a spaghetti")
- la fase di traduzione nel linguaggio di programmazione è estremamente semplice

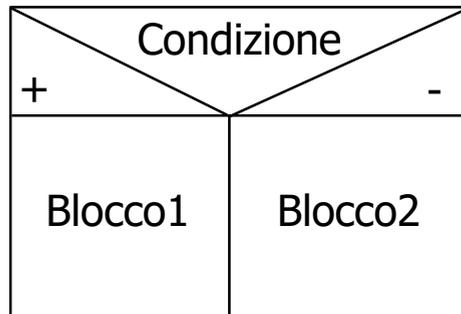
DNS – Definizione ricorsiva



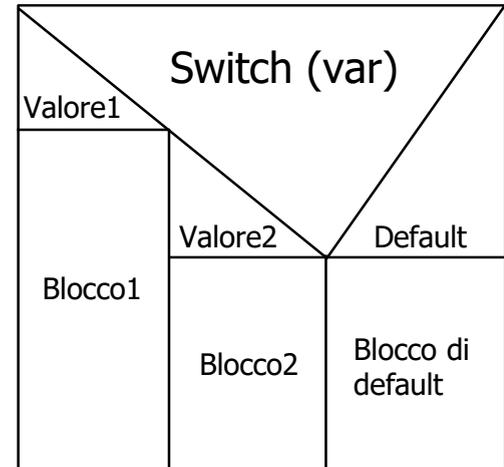
Blocco di selezione



Selezione singola



Selezione binaria



Selezione multipla

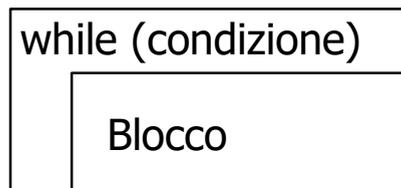
Blocchi iterativi

Un blocco di iterazione permette di eseguire ripetutamente un blocco di istruzioni fino a quando una condizione logica non è verificata.

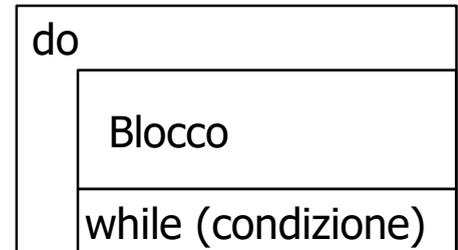
Per via della loro natura ripetitiva un blocco di iterazione è anche detto **ciclo**.



Ciclo for



Ciclo while



Ciclo do - while

Assembly ARM

Architettura ARM (Advanced Risc Machine)

Un processore con architettura ARM fa parte di una famiglia di processori di tipo **RISC** (Reduced Instruction Set Computer) generalmente con **registri a 32-bit**.

L'architettura di tipo RISC garantisce ottime prestazioni con un basso consumo energetico, per questo i processori ARM trovano un largo uso nel mercato dei dispositivi mobili dove il risparmio energetico è di fondamentale importanza. I processori ARM si trovano nei più comuni dispositivi come cellulari, tablet, televisori, videogiochi portatili, raspberry, ecc...

Registri

I principali registri dei processori ARM sono 16, nominati da R0 a R15, e si distinguono in base al loro utilizzo:

- **R0 - R12** sono i registri di **uso generale**;
- **R13** (o **SP**) è generalmente il registro dello **Stack Pointer**, cioè l'indirizzo di memoria dello stack. Non è però obbligatorio il suo utilizzo come Stack pointer; viene usato solo per convenzione;
- **R14** (o **LR**) è il registro del **Link Register**, nel quale viene salvato l'indirizzo di **ritorno** nel momento in cui viene chiamata una procedura;
- **R15** (o **PC**) è il **Program Counter**, il registro contiene l'**indirizzo di memoria** della prossima istruzione da eseguire.

Registri

R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
SP	0x00000000
LR	0x00000000
PC	0x00000000

Registro di stato

Un altro registro molto importante è il **registro di stato CPSR** (Current Program Status Register) il quale tiene traccia dello stato del programma. Molto utili sono i **4 bit di stato** (detti **flags di stato**) i quali esprimono le seguenti condizioni aritmetiche:

- **Bit N** negativo;
- **Bit Z** zero;
- **Bit C** carry/overflow;
- **Bit V** overflow.

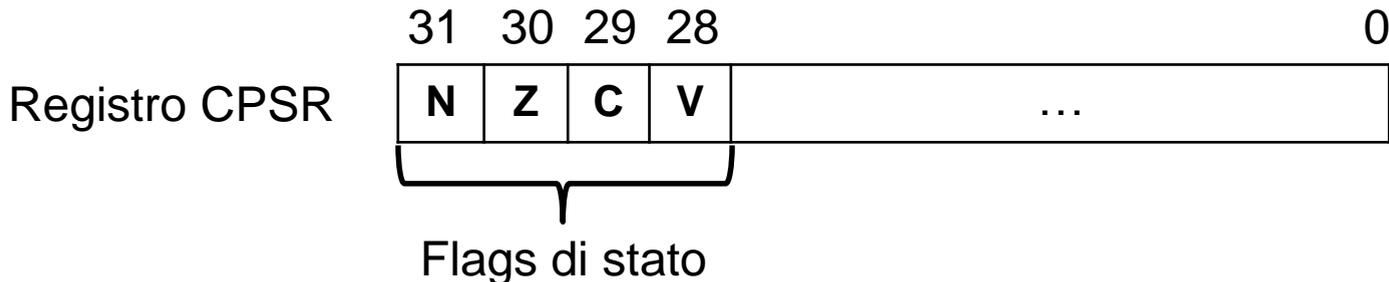
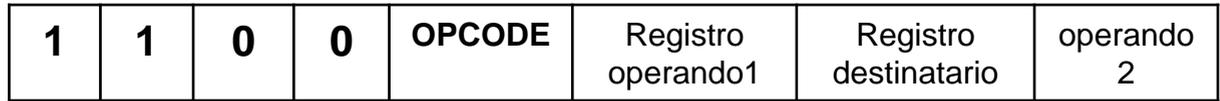


Tabella delle condizioni aritmetico-logiche

Suffisso	Descrizione	Flag testato	Condition Code
EQ	Equal (==)	Z = 1	0000
NE	Not Equal (!=)	Z = 0	0001
CS / HS	Unsigned maggiore uguale (>=)	C = 1	0010
CC / LO	Unsigned minore stretto (<)	C = 0	0011
MI	Negativo (< 0)	N = 1	0100
PL	Positivo o 0 (>= 0)	N = 0	0101
VS	Overflow	V = 1	0110
VC	No overflow	V = 0	0111
HI	Unsigned maggiore stretto (>)	C = 1 AND Z = 0	1000
LS	Unsigned minore uguale (<=)	C = 0 OR Z = 1	1001
GE	Maggiore uguale (>=)	N = V	1010
LT	Minore stretto (<)	N != V	1011
GT	Maggiore stretto (>)	Z = 0 AND N = V	1100
LE	Minore uguale (<=)	Z = 1 OR N != V	1101
AL	Always	----	1110

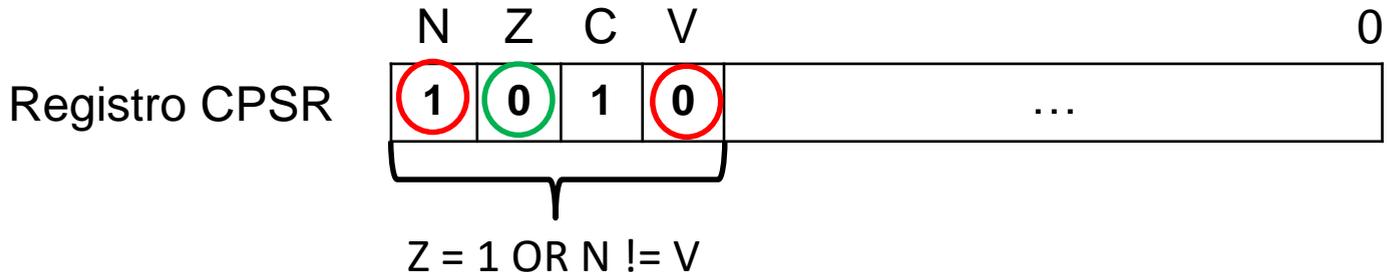
Istruzione:
ADDGT



Condition Code: GT

In questo esempio l'istruzione **ADD** verrà eseguita solo se la condizione **GT** (maggiore stretto) sarà verificata. La condizione GT è identificata dal **codice 1100** nella sezione del registro chiamata **condition code**.

In questo esempio la condizione è verificata in quanto nel **registro di stato CPSR** i flags NZCV rispettano le condizioni in tabella (**Z = 1 OR N != V**).



Tipi di dati

I tipi di dati sulle quali le istruzioni assembly ARM operano sono:

- **Byte** 1-byte, 8-bit
- **Half word** 2-byte, 16-bit
- **Full word** o **word** 4-byte, 32-bit

Il linguaggio assembly può interpretare questi tipi di dati ed esprimere diversi formati quali:

- Numeri interi espressi in diverse basi
- Numeri in virgola mobile
- Valori booleani
- Singoli caratteri
- Stringhe di caratteri

Set di istruzioni

Le istruzioni dell'architettura ARM possono essere suddivise in tre insiemi:

- **Istruzioni aritmetico-logiche** implementano le operazioni matematiche. Queste possono essere di tipo **aritmetiche** (somma, differenza, prodotto), **logiche** (operazioni booleane) o **relazionali** (comparazioni tra due valori).
- **Istruzioni di Branch** cambiano il controllo del flusso delle istruzioni, modificando il contenuto del Program Counter (R15). Le istruzioni di branch sono necessarie per l'implementazione di istruzioni di condizione (if-then-else), cicli e chiamate di funzioni.
- **Istruzioni di Load/Store** muovono dati da (load) e verso (store) la memoria principale.

Istruzioni aritmetico-logiche

In linguaggio assembly ARM le istruzioni aritmetico-logiche hanno una **sintassi** del tipo

opcode{S}{condition} dest, op1, op2

- Per **opcode** si intende l'istruzione che deve essere eseguita.
- Il campo **{S}** è un suffisso opzionale che, se specificato, carica nel registro di stato CPSR il risultato dell'operazione compresi i flag di stato.

Istruzioni aritmetico-logiche

- Anche il campo **{condition}** è un suffisso opzionale il quale definisce la condizione logica necessaria all'esecuzione dell'istruzione; se la condizione è verificata l'istruzione verrà eseguita altrimenti verrà ignorata.
- Il campo **dest** indica il **registro destinatario** nel quale verrà salvato il risultato dell'operazione.
- In fine **op1** e **op2** sono gli **operandi** dell'operazione che si desidera eseguire.

Bisogna specificare però che op1 fa riferimento all'operando soltanto mediante **indirizzamento a registro**, mentre op2 può indirizzare sia in maniera **immediata** che a **registro**.

Esempi istruzioni aritmetico-logiche

ADD R0, R1, R2

Carico in R0 la somma tra il contenuto del registro R1 e del registro R2 (indirizzamento a registro).

ADD R0, R1, #16

Carico in R0 la somma tra il contenuto del registro R1 e il numero decimale 16 (indirizzamento immediato).

ADD R0, R1, #0xF

Carico in R0 la somma tra il contenuto del registro R1 e il numero esadecimale F (indirizzamento immediato).

Esempi istruzioni aritmetico-logiche

ADDLT R0, R1, R2

Carico in R0 la somma tra il contenuto del registro R1 e del registro R2 solamente se la condizione indicata con il suffisso **LT** è verificata.

ADDS R0, R1, R2

Carico in R0 la somma tra il contenuto del registro R1 e del registro R2, inoltre carico lo stato del risultato nei flags NZCV del registro di stato CPSR.

Per esempio, se il risultato della somma va in overflow i flag C (carry) e V (overflow) verranno settati a 1.

Principali istruzioni aritmetico-logiche

Descrizione	Opcode	Sintassi	Semantica
Addizione	ADD	ADD R _d , R ₁ , R ₂ /#imm	$R_d = R_1 + R_2/\#imm$
Sottrazione	SUB	SUB R _d , R ₁ , R ₂ /#imm	$R_d = R_1 - R_2/\#imm$
Moltiplicazione	MUL	MUL R _d , R ₁ , R ₂ /#imm	$R_d = R_1 \times R_2/\#imm$
Carica nel registro	MOV	MOV R _d , R ₁ /#imm	$R_d \leftarrow R_1/\#imm$
Carica negato nel registro	MVN	MVN R _d , R ₁ /#imm	$R_d \leftarrow -(R_1/\#imm)$
AND logico	AND	AND R _d , R ₁ , R ₂ /#imm	$R_d = R_1 \wedge R_2/\#imm$
OR logico	ORR	ORR R _d , R ₁ , R ₂ /#imm	$R_d = R_1 \vee R_2/\#imm$
OR esclusivo	EOR	EOR R _d , R ₁ , R ₂ /#imm	$R_d = R_1 \oplus R_2/\#imm$
AND con complemento del secondo operando	BIC	BIC R _d , R ₁ , R ₂ /#imm	$R_d = R_1 \wedge \neg R_2/\#imm$
Shift logico sinistro	LSL	LSL R _d , R ₁ , R ₂ /#imm	$R_d = R_1 \ll R_2/\#imm$
Shift logico destro	LSR	LSR R _d , R ₁ , R ₂ /#imm	$R_d = R_1 \gg R_2/\#imm$
Rotazione destra	ROR	ROR R _d , R ₁ , R ₂ /#imm	$R_d = R_1 \cup R_2/\#imm$
Confronto	CMP	CMP R ₁ , R ₂	$NZCV \leftarrow R_1 - R_2$
Negato del confronto	CMN	CMN R ₁ , R ₂	$NZCV \leftarrow R_1 + R_2$

Istruzioni di branch

Le istruzioni di **branch** (o **jump**) sono utili per il controllo del flusso di esecuzione delle istruzioni.

Le principali istruzioni sono **B** (branch) e **BL** (branch with link).

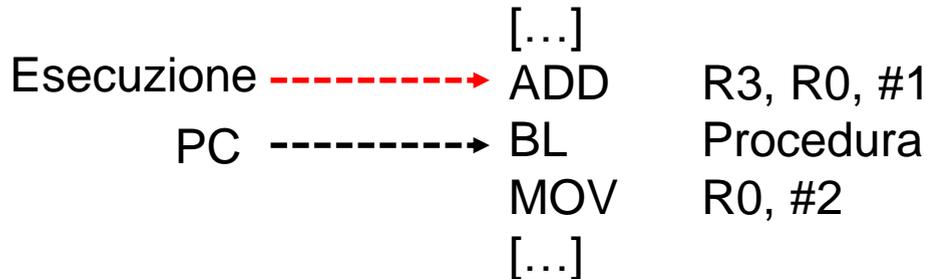
L'istruzione B carica nel PC (R15) l'indirizzo della prima istruzione della procedura che si desidera eseguire, causando così un salto nel flusso di esecuzione delle istruzioni. Per comodità le procedure vengono identificate univocamente con dei nomi detti **label** (o **etichetta**).

L'istruzione BL è simile all'istruzione B, in più però carica nel registro LR (R14) l'indirizzo di ritorno della procedura, ovvero il valore del PC nel momento in cui viene eseguita l'istruzione BL.

Esempio chiamata procedura con istruzione BL

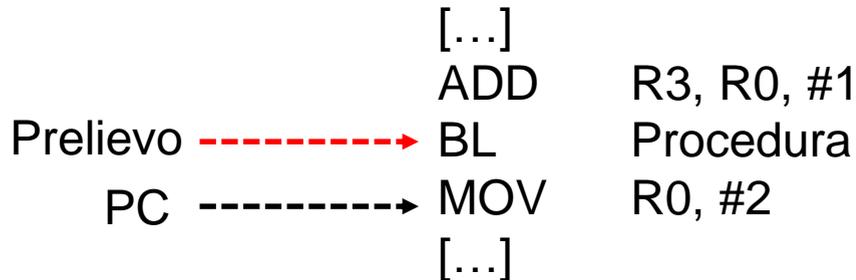
All'istante di tempo **T1** si sta eseguendo l'istruzione di ADD nell'esempio. Il PC in quel momento punta all'indirizzo in memoria che conterrà l'istruzione successiva (**BL Procedura**).

Istante di tempo T1



All'istante di tempo **T2** verrà prelevata l'istruzione BL (precedentemente puntata dal Program Counter). A questo punto il PC punterà all'istruzione successiva (**MOV R0,#2**).

Istante di tempo T2

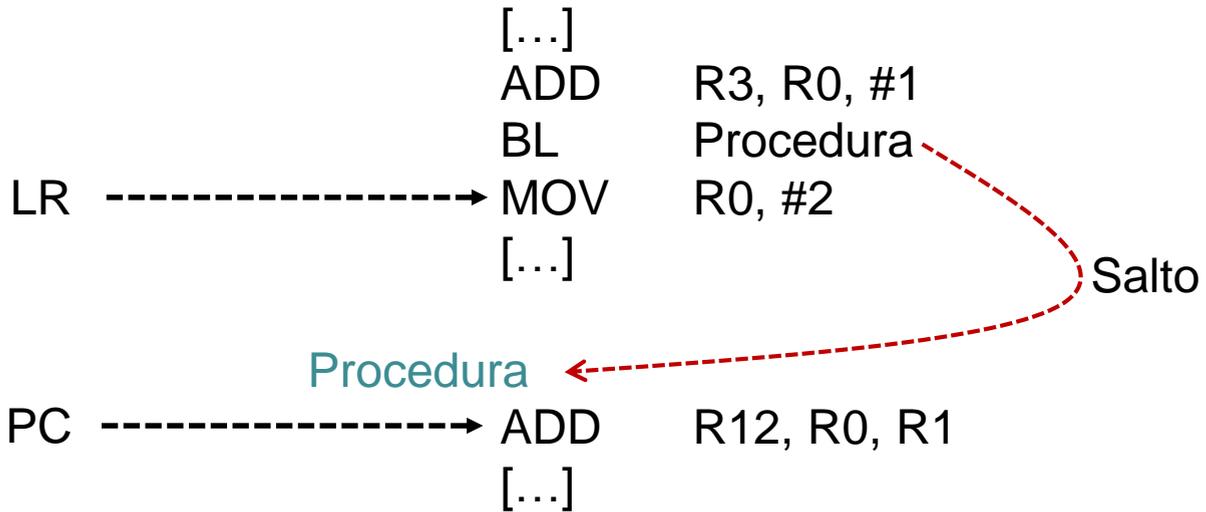


All'istante di tempo **T3** viene eseguita l'istruzione di Branch with link e accadono i seguenti eventi nel seguente ordine:

1. Il **Link Register** verrà caricato con il valore del Program Counter;
2. Il Program Counter verrà caricato con l'indirizzo della prima istruzione di un segmento di codice identificato con l'etichetta **Procedura**.

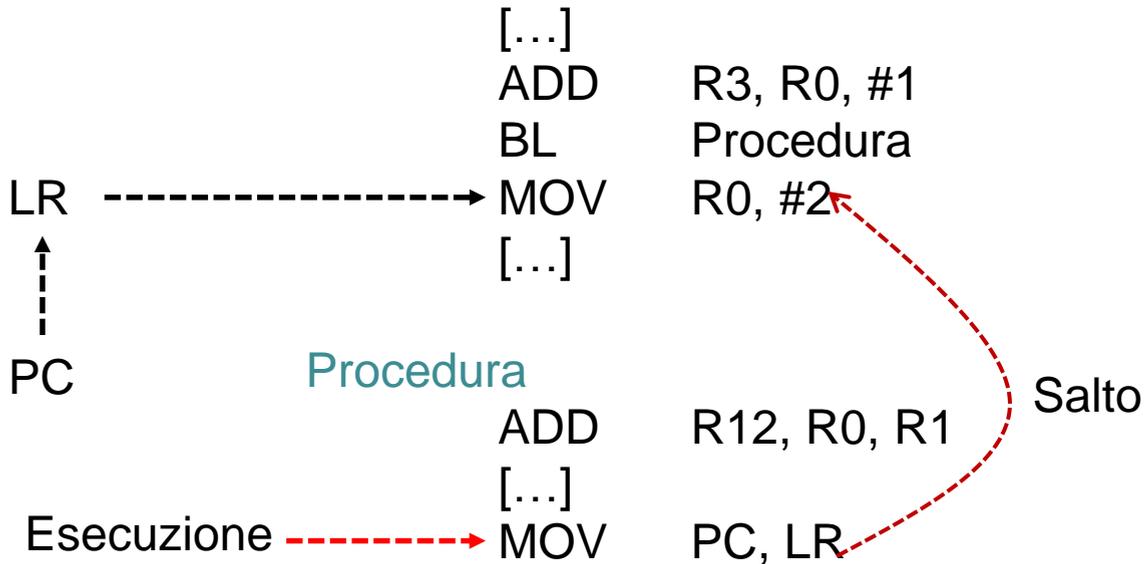
Così facendo avverrà un **salto** nel flusso di esecuzione delle istruzioni.

Istante di tempo T3

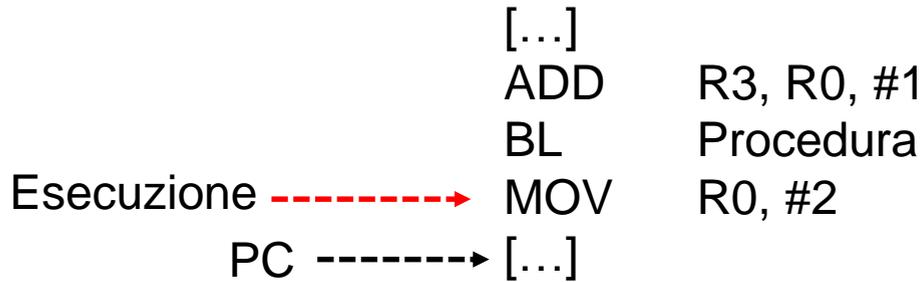


Dall'istante **T4** in poi verrà eseguito il segmento di codice identificato con l'etichetta **Procedura**. All'istante **Tn**, quando si arriverà all'ultima istruzione di **Procedura**, per tornare al normale flusso di istruzioni al momento del **salto** basterà caricare nel Program Counter il contenuto del Link Registr. Così dall'istante **Tn+1** in poi il programma riprenderà il normale svolgimento.

Istante di tempo Tn



Istante di tempo Tn+1



Istruzione di Load e Store

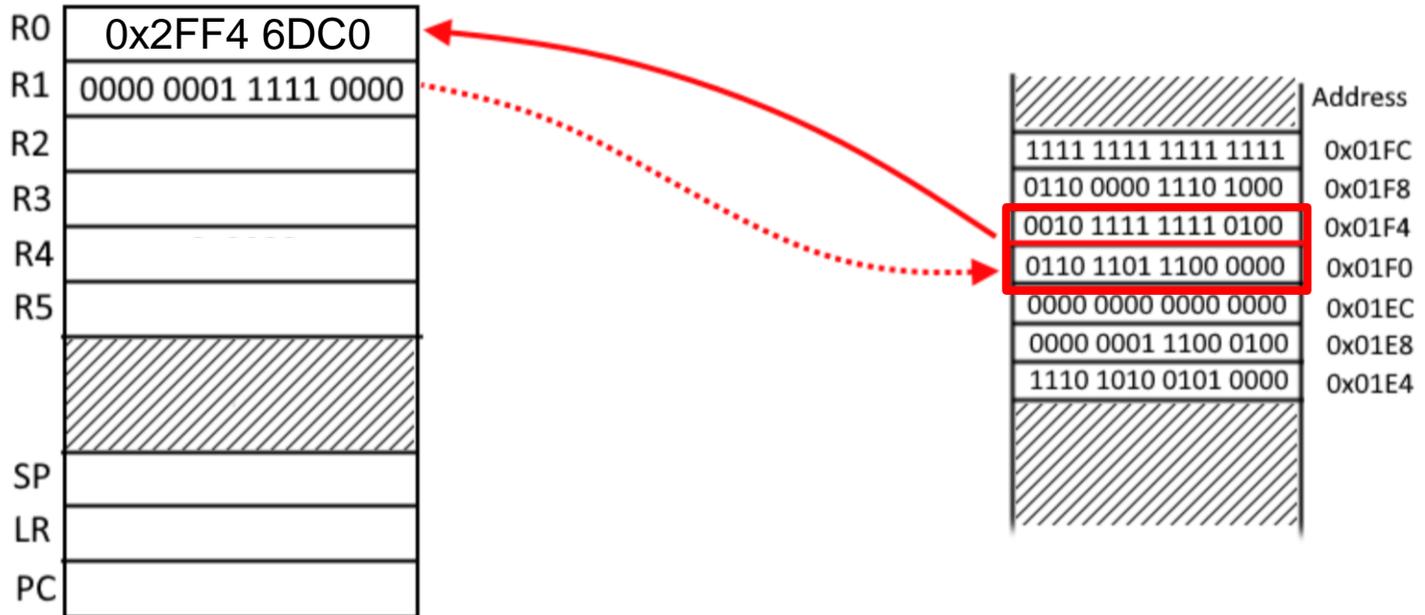
L'istruzione **LDR (Load Register)** carica in un registro destinatario un byte, una half word o una full word contenuta in una data locazione della memoria principale. La sintassi è

LDR{type}{condition} dest, [pointer]
LDR{type}{condition} dest, [pointer,offset]

Il campo **dest** è il registro destinatario nel quale caricare il contenuto prelevato dalla memoria. Il campo **[pointer]** indica un **registro puntatore** il quale definisce contenuto punta all'indirizzo in memoria della **cella** che si desidera referenziare. É possibile inoltre definire un **offset** il quale verrà sommato all'puntatore.

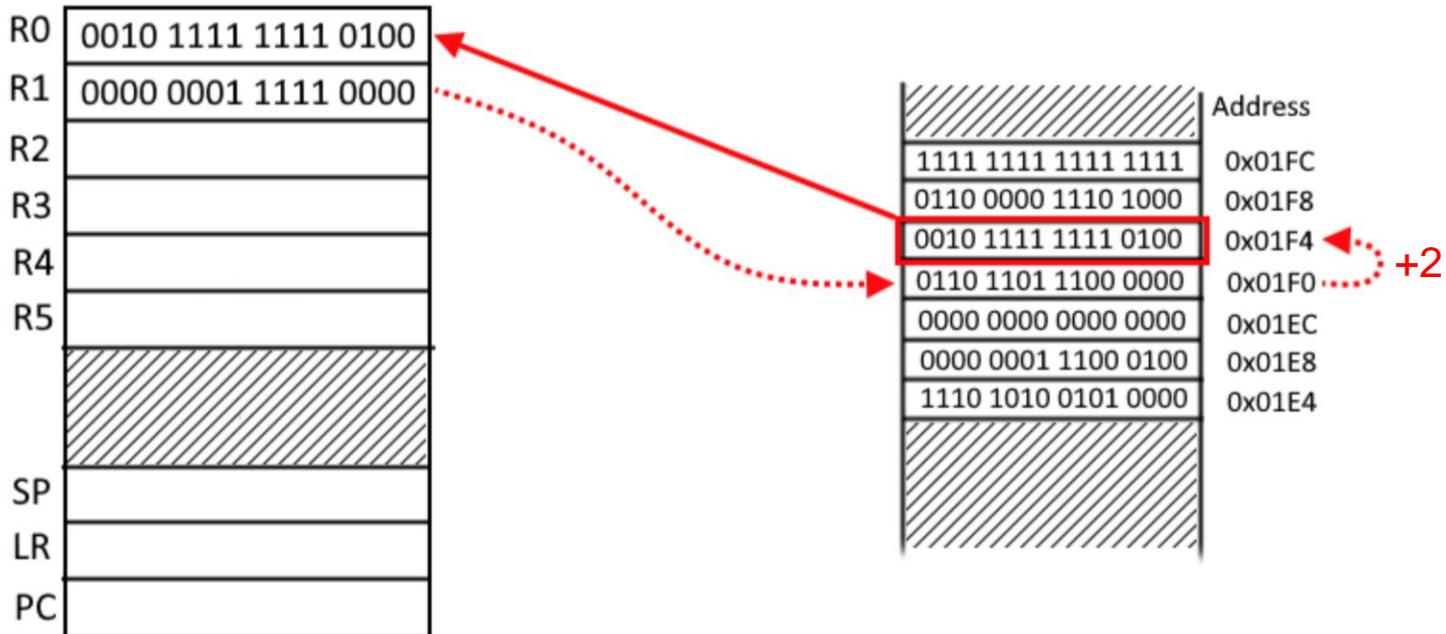
LDR R0, [R1, #2]

Carico in R0 la **parola** in memoria puntata dal registro R1



LDRH R0, [R1, #2]

Carico in R0 la HalfWord (H) 2 byte a partire dalla cella puntata dal registro R1

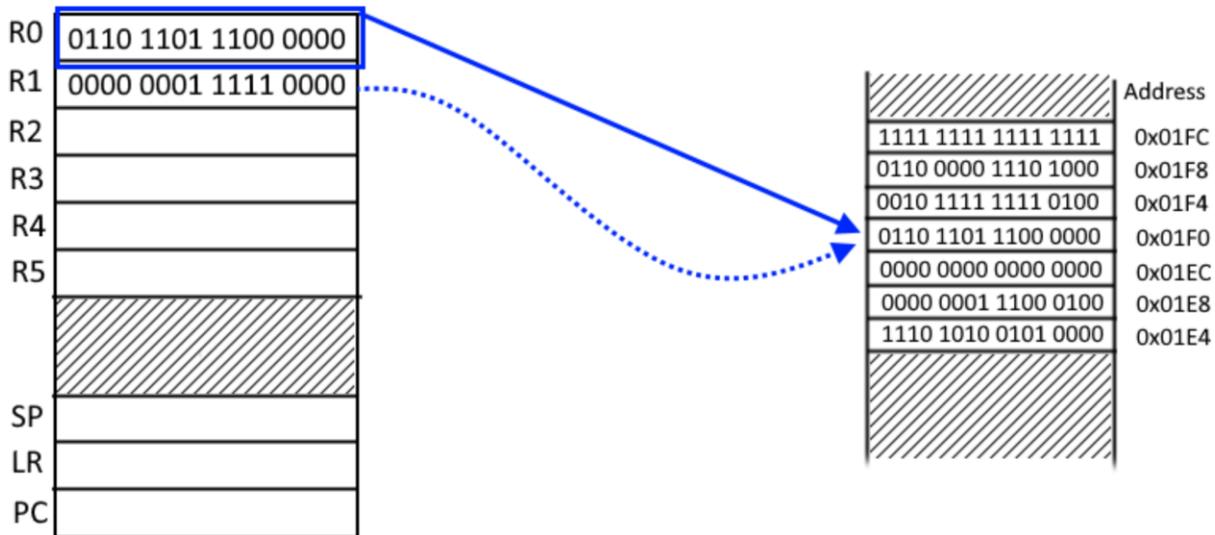


L'istruzione **STR** (**Store Register**) esegue esattamente l'operazione inversa di LDR; carica in memoria il contenuto di un registro **sorgente** all'indirizzo definito dal **puntatore**.

STR{condition} source, [pointer,offset]

STR R0, [R1]

Carico all'indirizzo puntato da R1 la parola contenuta nel registro R0

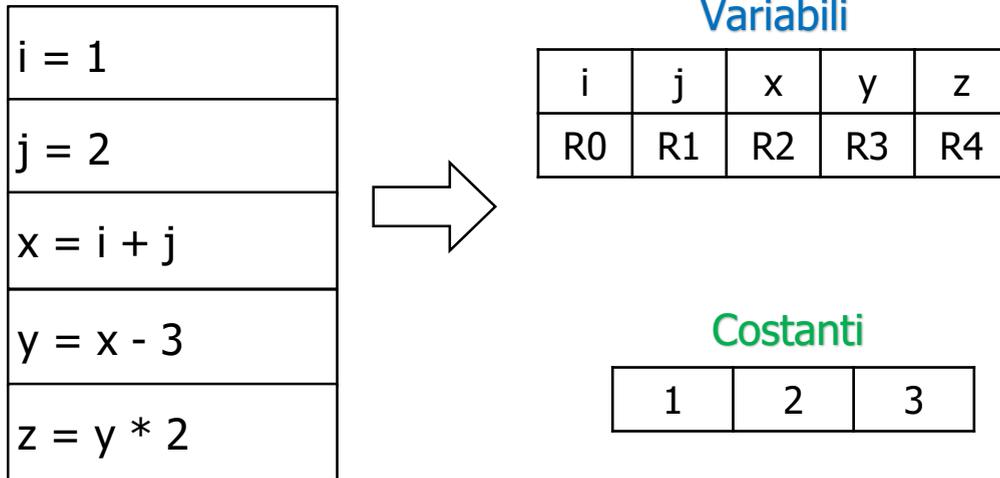


0x01F0 = 0000 0001 1111 0000

Traduzione dei blocchi DNS in linguaggio Assembly ARM

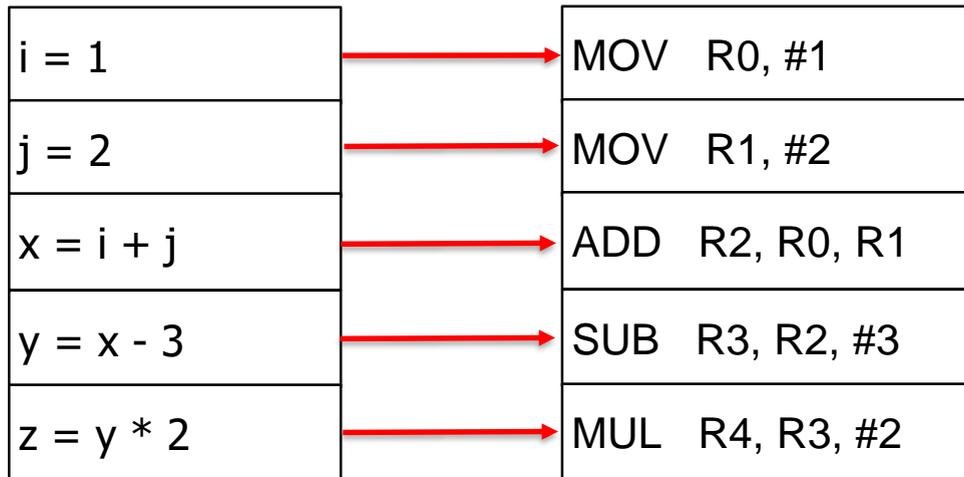
Sequenza di blocchi

Inizialmente si cercano tutte le variabili e costanti che compaiono ne DNS. Si assegna ogni variabile ai registri disponibili, se il loro numero è tale da contenere tutte le variabili (ed il numero di byte tali da poter essere memorizzati in un registro), altrimenti occorre memorizzarle in memoria.

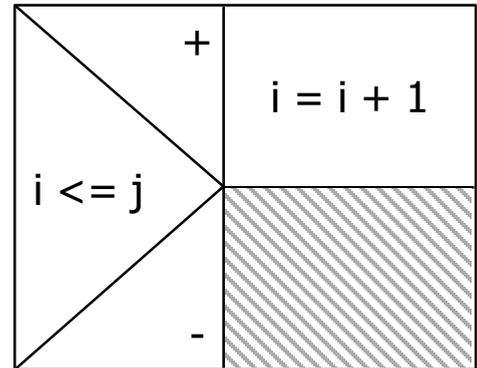
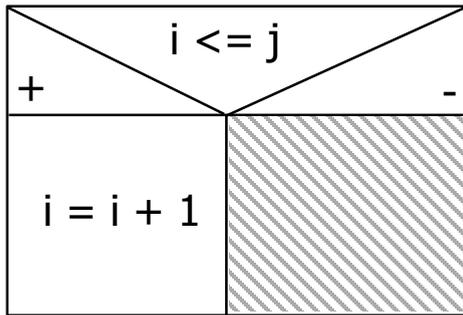


Sequenza di blocchi

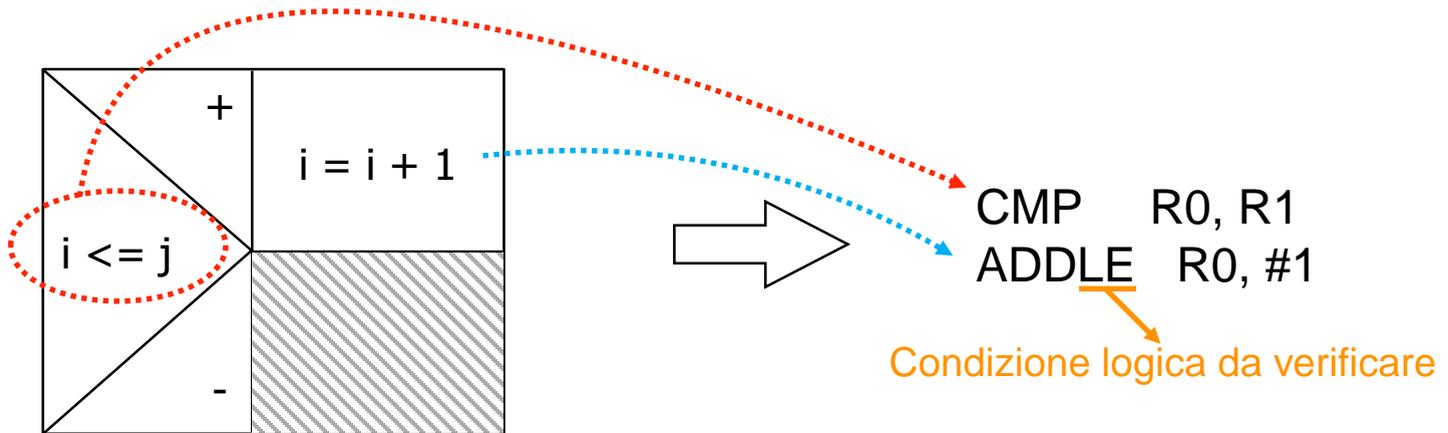
Se il blocco contiene una istruzione semplice, si traduce con la corrispondente istruzione assembly, altrimenti si segue la procedura che segue per ciascuna tipologia di blocco:



Selezione singola



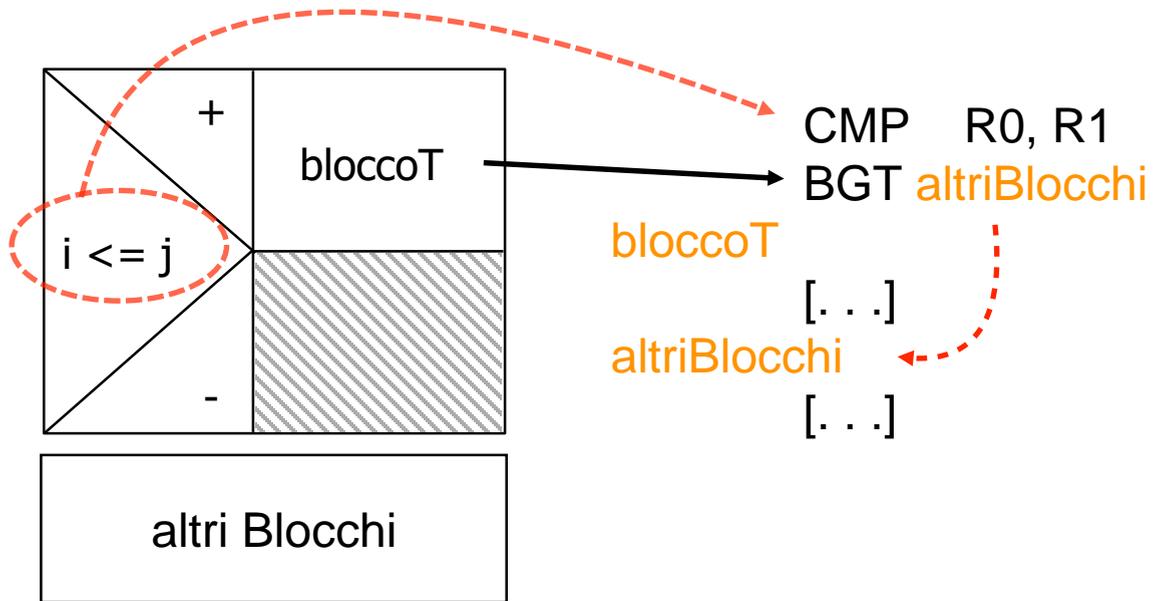
Selezione singola



Se la condizione logica è verificata viene eseguita l'istruzione ADD, altrimenti il flusso di esecuzione proseguirà il normale corso.

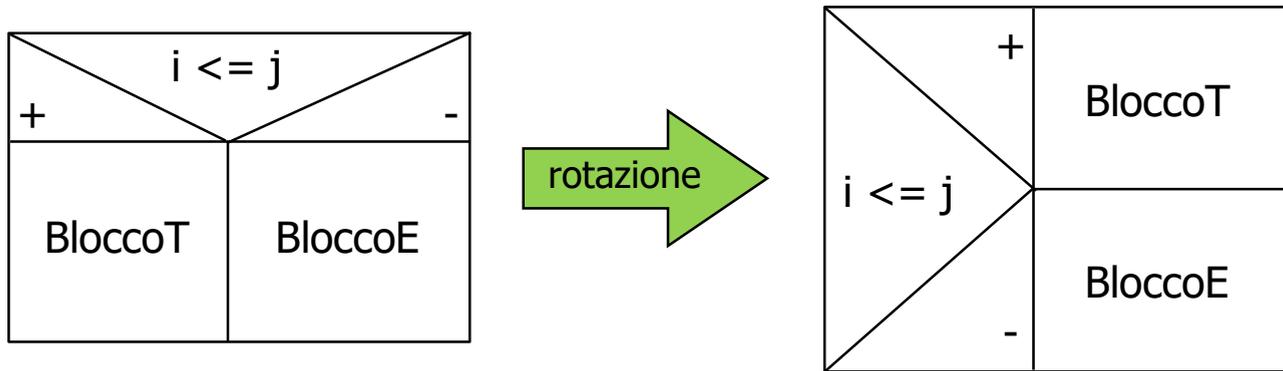
Selezione singola con branch

La selezione singola con esecuzione condizionata funziona nel caso ci sia una sola istruzione nel blocco, altrimenti è necessario utilizzare l'istruzione di branch.

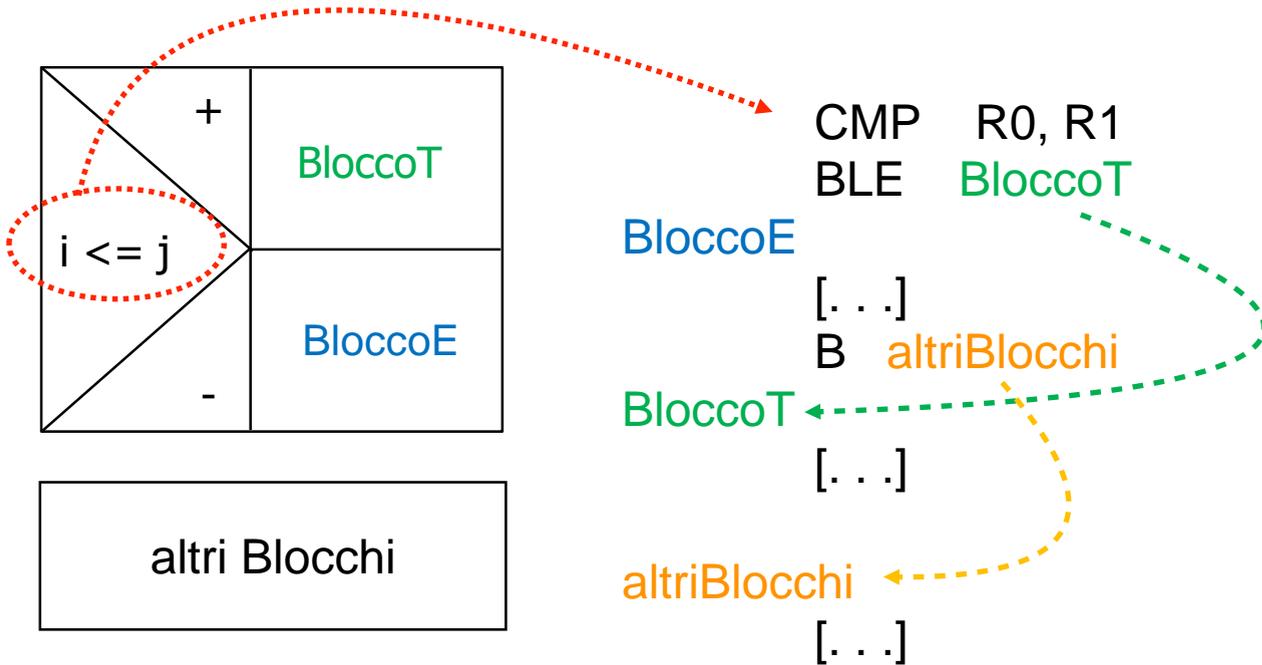


Selezione binaria

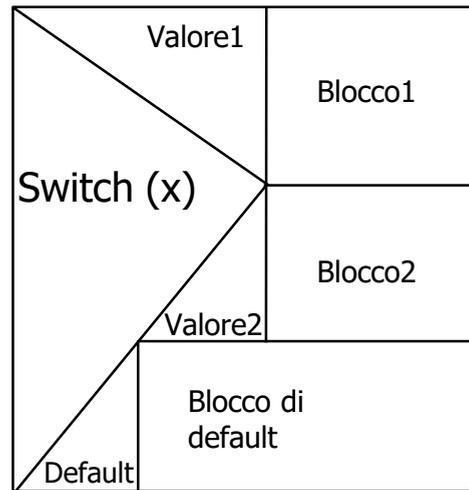
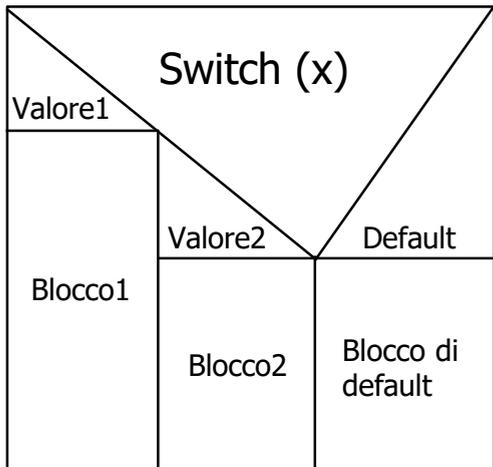
La selezione binaria verso altri blocchi richiedono una particolare attenzione, in quanto bisogna garantire il corretto proseguimento del flusso di istruzioni del programma.



Selezione binaria



Selezione multipla

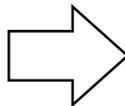
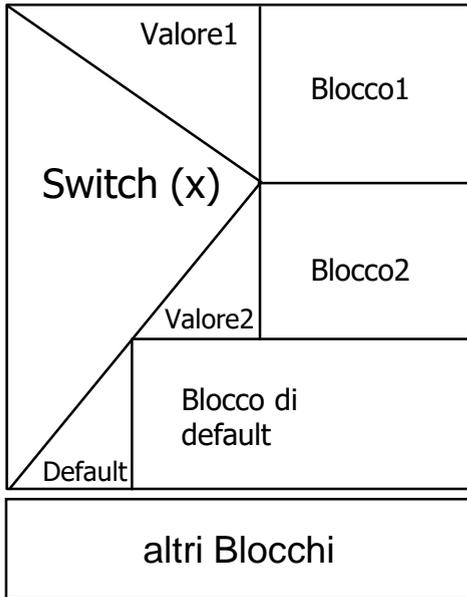


Variabili

x	Valore1	Valore2
R0	R1	R2

Selezione multipla

$x = \text{Valore1} ? \longrightarrow \text{CMP } R0, R1$
 $\text{BEQ } \text{Blocco1}$
 $x = \text{Valore2} ? \longrightarrow \text{CMP } R0, R2$
 $\text{BEQ } \text{Blocco2}$
 $\text{Default} \longrightarrow \text{B } \text{Default}$



Blocco1

[. . .]

B altriBlocchi

Blocco2

[. . .]

B altriBlocchi

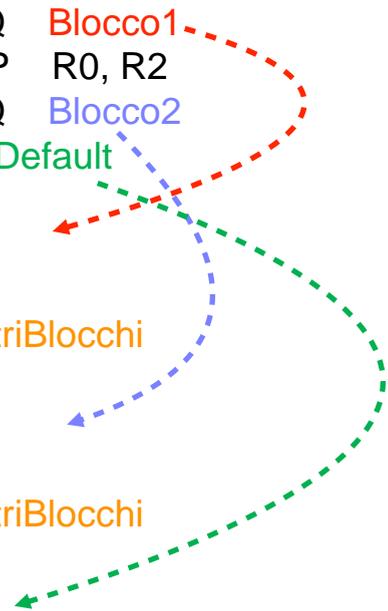
Default

[. . .]

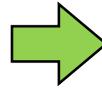
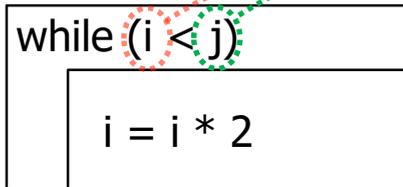
B altriBlocchi

altriBlocchi

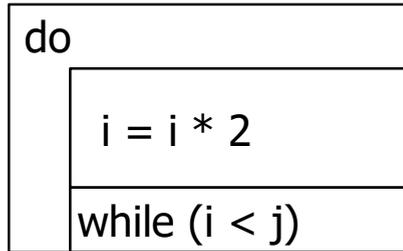
[. . .]



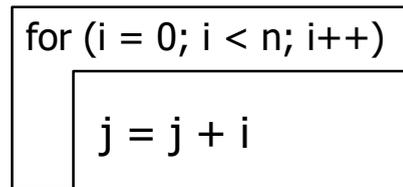
Blocchi iterativi



```
ciclo
    CMP    R0, R1
    BGE   end
    MUL   R0, R0, #2
    B     ciclo
```



```
end
ciclo
    MUL   R0, R0, #2
    CMP   R0, R1
    BLT   ciclo
```



```
MOV    R0, #0
ciclo
    CMP   R0, Rn
    BGE   end
    ADD   R1, R1, R0
    ADD   R0, R0, #1
    B     ciclo
end
```

Esempio

Massimo tra tre numeri interi

Esercizio

Calcolare il massimo tra tre numeri interi $x, y, z \in \mathbb{Z}$

Soluzione

Prima di tutto definiamo prototipo, pre-condizioni e post-condizioni che la funzione deve rispettare:

1) prototipo

int max (X, Y, Z)

2) pre-condizioni

X, Y e Z sono interi

3) post-condizione

max (X, Y, Z) restituisce il massimo tra X, Y e Z

Massimo tra tre numeri interi

Ora proviamo a scrivere in linguaggio naturale l'algoritmo: un modo per trovare il massimo è quello di confrontare i valori a due a due fino a trovare il più grande.

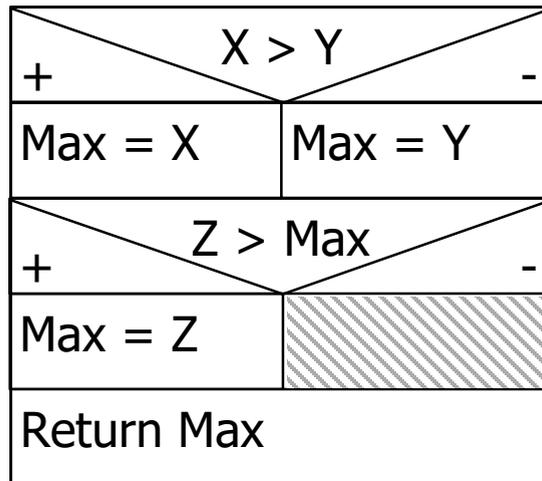
Formalizzando l'algoritmo i passaggi sono:

1. Confronta X e Y e calcola il massimo temporaneo;
2. Confronta il massimo temporaneo precedentemente calcolato con Z per e calcola il massimo definitivo.

DNS

Il corrispondente diagramma DNS è:

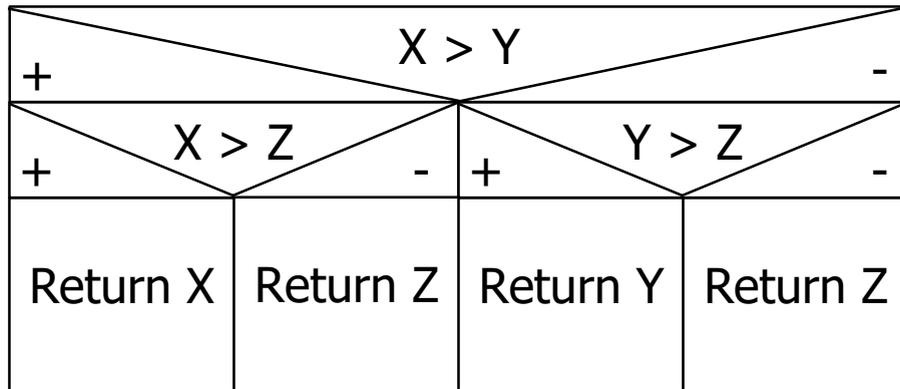
int max(X , Y , Z)



Una soluzione alternativa

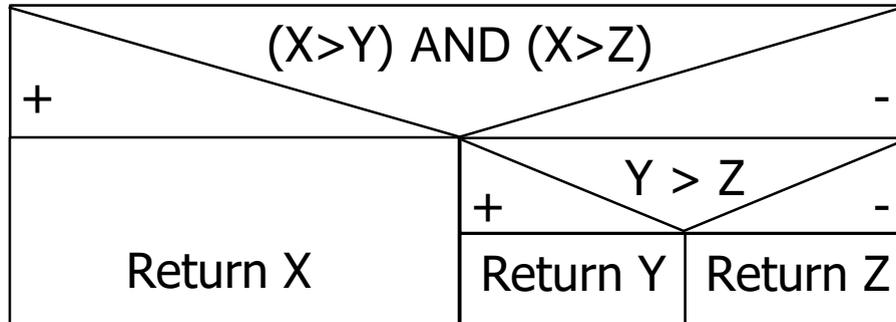
Tuttavia per ogni problema si possono trovare più soluzioni con diverse caratteristiche in termini di efficienza, complessità, riusabilità, ...

```
int max( X , Y , Z )
```



Terza soluzione

```
int max( X , Y , Z )
```



La fase di traduzione del DNS in assembly

Con il metodo che abbiamo visto, trasformiamo il primo diagramma DNS nel corrispondente codice Assembly ARM.

Inizialmente per studiare se la soluzione è corretta assegniamo ad X, Y e Z dei valori costanti

```
varX EQU #X  
varY EQU #Y  
varZ EQU #Z
```

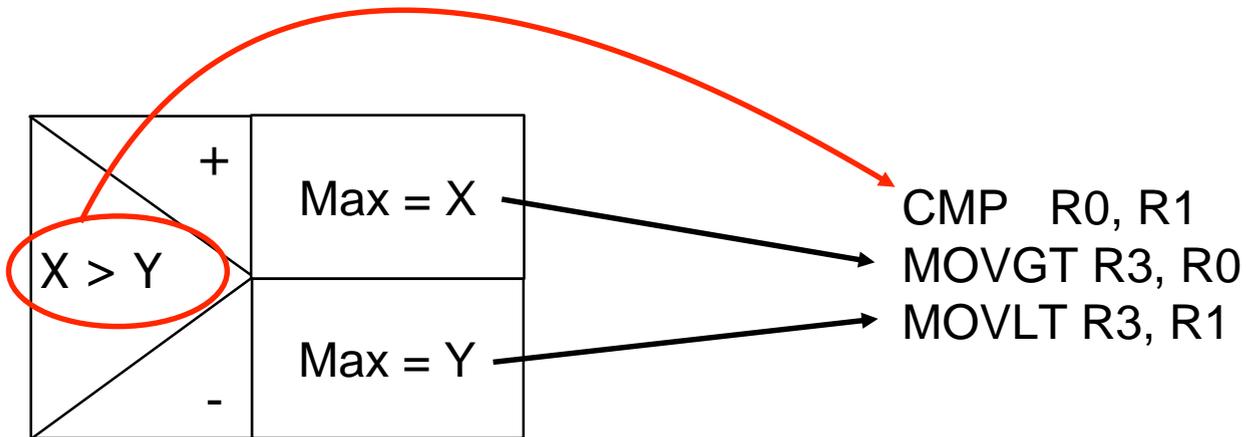
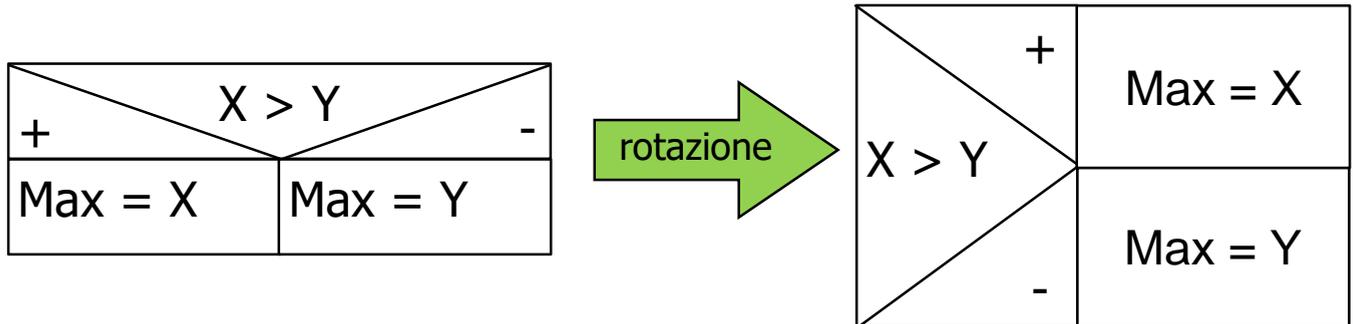
La fase di traduzione del DNS in assembly

Carichiamo convenzionalmente le costanti corrispondenti registri:

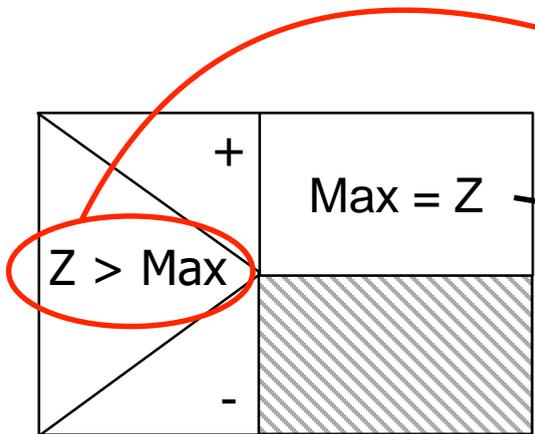
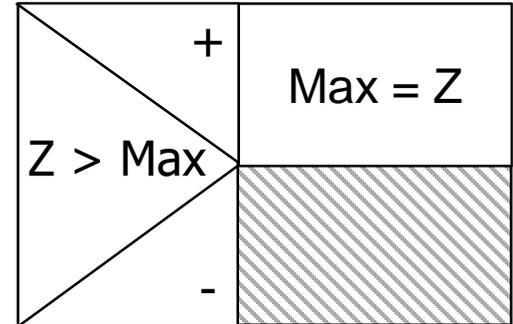
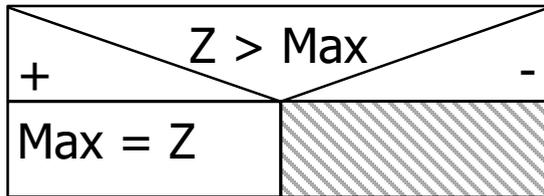
```
varX EQU #X  
varY EQU #Y  
varZ EQU #Z
```

```
MOV R0, #varX  
MOV R1, #varY  
MOV R2, #varZ
```

La fase di traduzione del DNS in assembly



La fase di traduzione del DNS in assembly

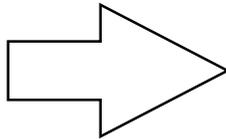


```
CMP R2, R3  
MOVGT R3, R2
```

La fase di traduzione del DNS in assembly

In fine supponendo di voler salvare il risultato della computazione nello Stack basterà eseguire una istruzione di Store Register all'indirizzo puntato dallo Stack Pointer.

Return Max



STR R3, [SP]

Codice assembly ARM finale

```
1 varX    EQU    1
2 varY    EQU    2
3 varZ    EQU    3
4
5     MOV     R0, #varX ; leggo X
6     MOV     R1, #varY ; leggo Y
7     MOV     R2, #varZ ; leggo Z
8
9     CMP     R0, R1    ; X > Y ?
10    MOVGT   R3, R0    ; TRUE ----> Max = X
11    MOVLT   R3, R1    ; FALSE ----> Max = Y
12
13    CMP     R2, R3    ; Z > Max ?
14    MOVGT   R3, R2    ; TRUE ----> Max = Z
15
16    STR     R3, [sp]  ; Return Max
17    END
18
```

Massimo tra tre numeri interi

Esercizio

Calcolare la moltiplicazione tra due numeri interi positivi $x, y \in \mathbb{N}$ senza utilizzare l'istruzione MUL.

Soluzione

Prima di tutto definiamo prototipo, pre-condizioni e post-condizioni che la funzione deve rispettare:

1) prototipo

int molt (X, Y)

2) pre-condizioni

X, Y sono interi e positivi

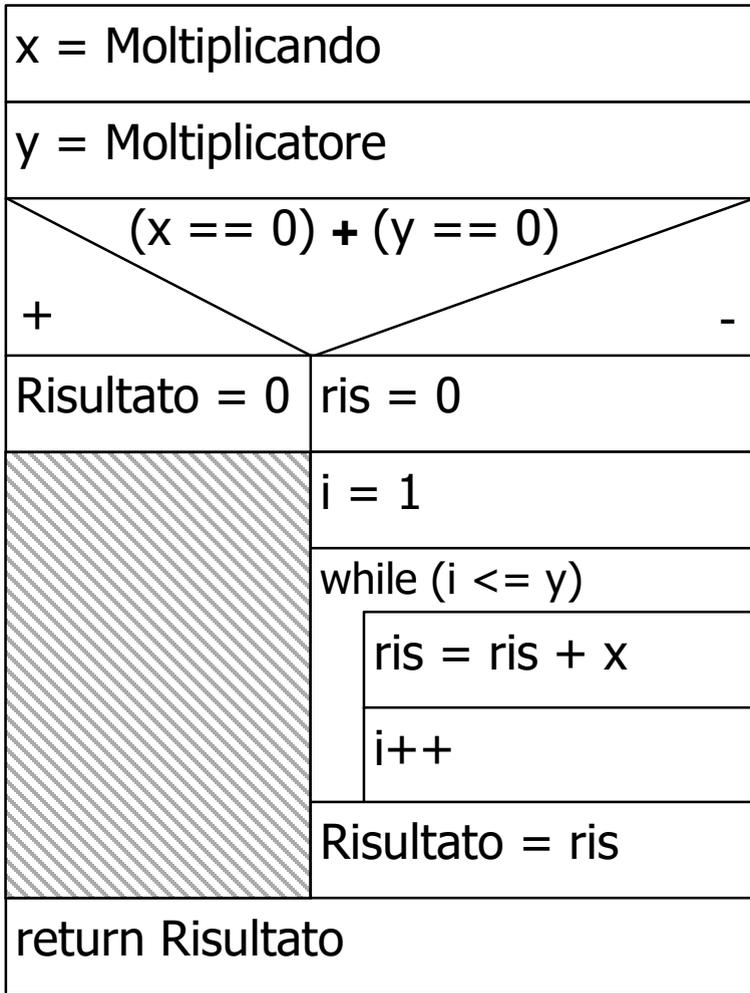
3) post-condizione

molt (X, Y) restituisce il massimo tra $X * Y$

Moltiplicazione per addizioni successive

È possibile implementare la moltiplicazione tra due numeri interi positivi $x, y \in \mathbb{N}$ mediante il solo utilizzo di addizioni. Infatti è possibile definire la moltiplicazione tra numeri interi sotto forma di serie numerica.

$$x \times y := \begin{cases} 0 & \text{se } x = 0 \text{ o } y = 0 \\ \sum_{i=1}^y x = \sum_{i=1}^x y & \text{se } x > 0 \text{ e } y > 0 \end{cases}$$



```

1 Moltiplicando EQU 7 ; x = Moltiplicando
2 Moltiplicatore EQU 8 ; y = Moltiplicatore
3 Risultato DCD 0
4
5 start
6 ADR R10, Risultato ; Carico in R10 l'indirizzo di memoria del Risultato
7
8 SUBS R0, R0, #Moltiplicando ; controllo se il Moltiplicando è uguale a 0
9 BPL Blocco1 ; TRUE
10 SUBS R1, R1, #Moltiplicatore ; controllo se il Moltiplicatore è uguale a 0
11 BPL Blocco1 ; TRUE
12 B Blocco2 ; FALSE
13
14
15 ; ===== TURE (x == 0) OR (y == 0) =====
16 Blocco1
17 MOV R0, #0 ; ris = 0
18 B fine
19
20
21 ; ===== FALSE (x == 0) OR (y == 0) =====
22 Blocco2
23 MOV R0, #0 ; ris = 0
24 MOV R1, #1 ; i = 1
25
26 CMP R1, #Moltiplicatore ; confronto i e y
27 ciclo
28 ADDLE R0, R0, #Moltiplicando ; ris = ris + x
29 ADDLE R1, R1, #1 ; i ++
30 CMP R1, #Moltiplicatore ; confronto i e y
31 BLE ciclo ; i <= y?
32
33 B fine
34
35 ; ===== Continuazione dell' algoritmo =====
36 fine
37 STR R0, [R10] ; Risultato <-- ris
38 END
39

```