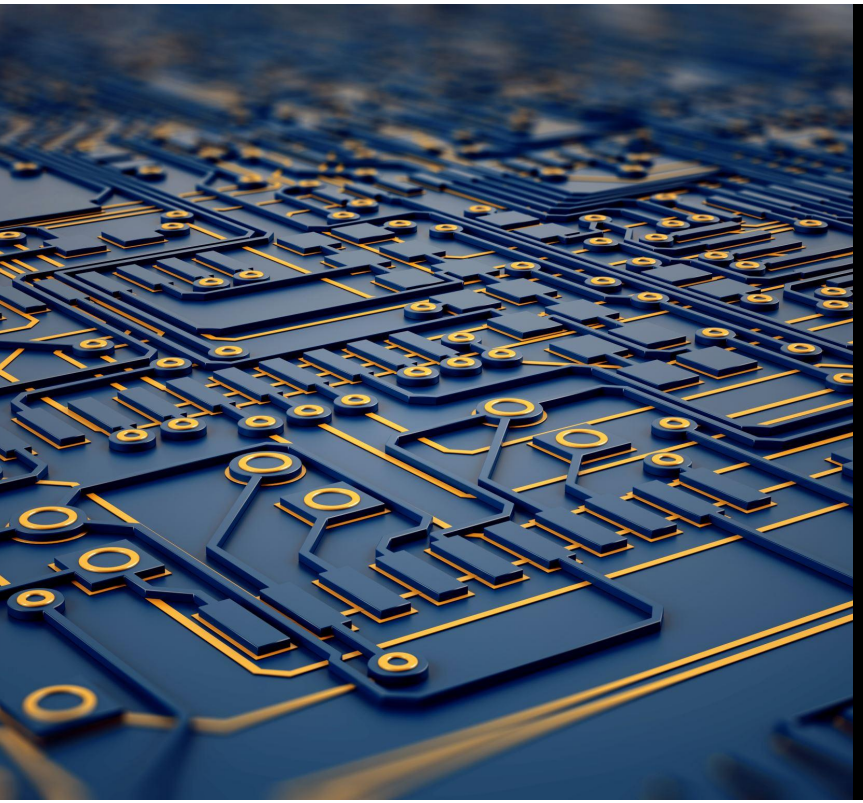


INFORMATICA

Intelligenza Artificiale & Data Analytics

Precorsi a.a. 2023/24

Docente: Gloria Pietropolli

6. BASI DI PROGRAMMAZIONE

Istruzioni matematiche e operatori aritmetici

IL NOSTRO PRIMO PROGRAMMA CON COSTRUTTO IF

Iniziamo a introdurre il costrutto if.
Vogliamo un programma che accetti in input due numeri interi x e y.
Se x è minore di y, voglio che i due numeri vengano **sommati**, altrimenti **moltiplicati**.
Voglio ottenere in output il risultato di questa operazione.

1) dobbiamo solamente verificare se x è minore di y

```
x ← input()
y ← input()
a ← x < y
```

```
IF a:
    z ← x + y
    output(z)
ELSE:
    z ← x * y
    output(z)
```

Se a è vero Esegui le istruzioni qui dentro

Altrimenti (Se a è falso) Esegui le istruzioni qui dentro

RECAP

Dalla lezione 2

ISTRUZIONI E OPERATORI LOGICI

Nella sua accezione più di base, il programma è costituito da istruzioni che possono essere:

- Di tipo matematico (es. somma due numeri, fai la radice di questo numero…)
- Di tipo logico (es. si verificano questo E quest'altro, NON si verifica questo…)
- Di tipo imperativo (es. SE si verifica qualcosa ALLORA fai questo)
- …

CHE COSA FAREMO OGGI

Inizieremo a costruire un linguaggio, più simile possibile a quello naturale, per poter definire un procedimento algoritmico.
Lo chiameremo ***pseudo-codice***

Impareremo a *compilare* lo pseudo-codice e a produrre un set di istruzioni per eseguire il programma sul modello semplificato.

PREMESSA

Per semplicità, ci limiteremo a considerare soli algoritmi per effettuare **calcoli numerici interi**.

La parte riguardante **numeri a virgola mobile** e **stringhe** a cui abbiamo accennato nelle lezioni precedenti è invece propedeutica a seguire i corsi di informatica del I e II semestre.

ISTRUZIONI DI TIPO MATEMATICO (I)

Il nostro computer deve svolgere **operazioni fra numeri interi**

In informatica, si possono categorizzare gli **operatori** in:

- **unari**
- **binari**
- **ternari**...

in base al numero di argomenti che accettano.

*****

È un operatore **binario**, in quanto moltiplichiamo due numeri
2 * 5
Il 2 e il 5 sono gli **argomenti** della moltiplicazione

-

Può essere sia **binario** che **unario**. Infatti, in -5 lo usiamo ad indicare il numero 5 negativo, mentre in 3 - 2 lo usiamo per indicare la sottrazione

ISTRUZIONI DI TIPO MATEMATICO (II)

Vogliamo poter utilizzare i seguenti operatori matematici:

Operatori binari

- `+` Addizione
- `-` Sottrazione
- `*` Moltiplicazione
- `/` Divisione intera 
- `%` Resto

Operatori unari

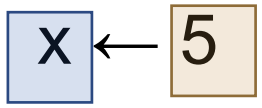
- `-` Cambio segno

VARIABILE E ASSEGNAZIONE

La **variabile** è un **nome** che diamo ad un **numero intero** nell'ambito del nostro programma

Vedremo anche variabili riferite ad altri tipi di dato, ma ora rimaniamo solamente ai numeri interi

Possiamo **definire** una variabile **assegndoci** un numero intero.

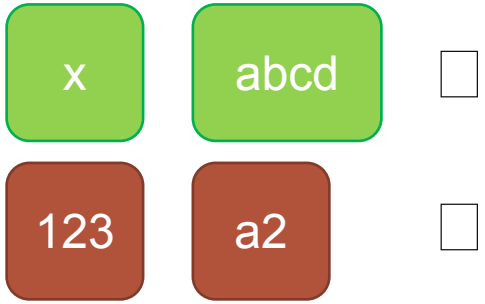


Definiamo una nuova variabile chiamata «x»

E vi assegniamo il numero intero 5

CHIARIMENTI SU VARIABILI E ASSEGNAZIONE

Le variabili possono avere nomi contenenti solo **caratteri testuali**



L'assegnazione è un operatore binario NON matematico, che richiede due argomenti:

- Una variabile a sinistra
- Un intero a destra

x ← 5

5 ← x

5 ← 3

NB: un'assegnazione di questo tipo è invece valida
x ← 5
y ← x
La vediamo come: prendi il valore di x ed assegnalo ad y

OUTPUT

Potremmo usare questo linguaggio per programmi semplici
→ per fare i conti come una calcolatrice molto grezza.

Dobbiamo far conoscere il risultato all'utente

→ i calcoli vengono tutti fatti dalla CPU e i risultati conservati in memoria.

Abbiamo bisogno di un operatore **output()** che trasferisca uno o più risultati dalla **memoria** ad un **dispositivo di output** (schermo, stampante...)

IL NOSTRO PRIMISSIMO PROGRAMMA

Vogliamo creare un programma che risolva la seguente espressione aritmetica e che ne restituisca il risultato

3 * 5 - 12

Iniziamo assegnando i numeri interi a delle variabili

x ← 3

y ← 5

z ← 12

Ora possiamo risolvere l'espressione

w ← x * y

risultato ← w - 12

output(risultato)

Siccome output non è un simbolo, decidiamo, per chiarezza, di indicarne l'argomento fra parentesi

Nota: siccome dobbiamo effettuare due operazioni nella nostra espressione, decidiamo di salvare il risultato intermedio della moltiplicazione in una quarta variabile

INPUT

Se pensiamo alle calcolatrici, è l'utente a definire i valori dei numeri da inserire in una espressione.

Se vogliamo moltiplicare due numeri int x e y definiti dall'utente al momento dell'esecuzione del programma.

In *runtime*

Operatore **input()**

- non accetta argomenti
- indica che il valore di quella variabile deve essere definito dall'utente in runtime.

IL PROGRAMMA DI PRIMA, MA CON INPUT VARIABILE

x * y - z

Abbiamo sostituito 3, 5 e 13 con delle variabili x, y e z di cui non conosciamo ancora il valore (sono un *input variabile*)

```
x ← input()
y ← input()
z ← input()
```

La seconda parte del programma resta invariata

```
w ← x * y
risultato ← w - z
output(risultato)
```

6. BASI DI PROGRAMMAZIONE

Istruzioni imperative e dati booleani, costrutto if

ISTRUZIONI IMPERATIVE

In questo momento, il nostro computer non si differenzia da una calcolatrice da tavolo.
Può fare conti, accettare input e restituire il risultati in output.

Ricordiamo che un algoritmo si compone anche di istruzioni imperative, che sono del tipo

SE si verifica qualcosa, **ALLORA** fai questo,
ALTRIMENTI fai qualcos'altro

In informatica, chiamiamo questa istruzione un COSTRUTTO IF

UN PROBLEMA

Il nostro linguaggio non consente di eseguire la prima parte dell'istruzione imperativa

SE `si verifica qualcosa`

Come facciamo a controllare se si verifica qualcosa se abbiamo a disposizione solo numeri interi e possiamo fare solo operazioni aritmetiche?

VERO/FALSO

La frase «SE si verifica qualcosa» richiede che questo «*qualcosa*» possa essere:

- ★ VERO
- ★ FALSO

In altre parole, possiamo riscrivere il costrutto imperativo in questo modo

SE qualcosa è **vero**, **ALLORA** fai questo, **ALTRIMENTI** (ovvero se qualcosa **è falso**) fai qualcos'altro

VARIABILI BOOLEANE

Formalizziamo introducendo un nuovo tipo di dato

Il **dato booleano** può assumere solo due valori

- ★ vero
- ★ falso

posso definire una variabile booleana in questo modo

a ← vero oppure a ← falso

OPERATORI DI CONFRONTO (I)

Introduciamo un nuovo tipo di operatori fra numeri interi: **gli operatori di confronto**

Gli operatori di confronto:

- accettano due interi
- **restituiscono un dato booleano**

`3 + 5`

Operatore aritmetico:
+ accetta due interi e restituisce un altro intero (8 nell'esempio sopra)

`6 > 5`

Operatore di confronto:
> accetta due interi e restituisce un valore booleano (vero nel caso sopra)

OPERATORI DI CONFRONTO (II)

All’interno del nostro linguaggio, riconosciamo sei operatori di confronto fra interi.
Tutti questi accettano due interi e restituiscono un valore booleano.

$>$	Maggiore	$\geq$	Maggiore-uguale
$<$	Minore	$\leq$	Minore-uguale
$=$	Uguale	$\neq$	Diverso

IL NOSTRO PRIMO PROGRAMMA CON OPERATORI DI CONFRONTO

Vogliamo costruire un programma che effettui la sottrazione di due numeri interi e che ci dica se questo numero è positivo

```
x ← input()
y ← input()
z ← x – y
a ← z > 0
output(a)
```

Nota, a adesso è un booleano (VERO/FALSO)

SALTIAMO UN PASSAGGIO...

Se vogliamo essere audaci, possiamo anche saltare l'assegnazione di x – y a z e scrivere direttamente così il programma

```
x ← input()
y ← input()
a ← x – y > 0
output(a)
```

PARITÀ DI UN NUMERO

Proviamo ad aumentare la complessità del programma.

Abbiamo un numero x in input, vogliamo determinare se il numero è pari oppure dispari. Ammettiamo per semplicità che 0 sia pari.

Abbiamo a disposizione solamente operatori aritmetici e di confronto.
Come facciamo a determinare se x è pari?

Prendiamo i primi 3 numeri pari e dispari.
Proviamo a dividere questi numeri per 2 (nota: abbiamo a disposizione solo la divisione intera!)
Notate qualche pattern?

```
x ← input()
y ← x % 2
a ← y = 0
output(a)
```

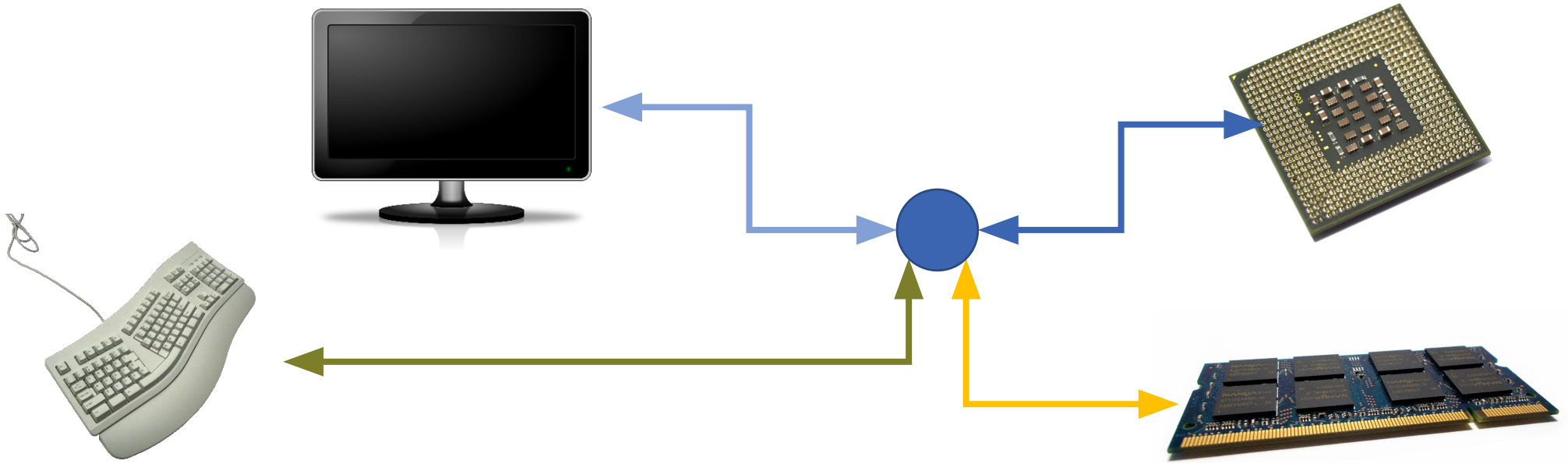
6. BASI DI PROGRAMMAZIONE

Il programma fra memoria ed elaboratore, ovvero, come l'elaboratore gestisce l'esecuzione del programma

MA... COM'È CHE IL COMPUTER FA FUNZIONARE IL PROGRAMMA?

Facciamo un recap rapido della prima lezione.

Prendiamo un computer con l'hardware minimo affinché possa funzionare (omettiamo la scheda madre per semplicità):



CARICAMENTO DEL PROGRAMMA

Ricordiamo, il programma è una sequenza di istruzioni

Le istruzioni vengono spacchettate e caricate in memoria

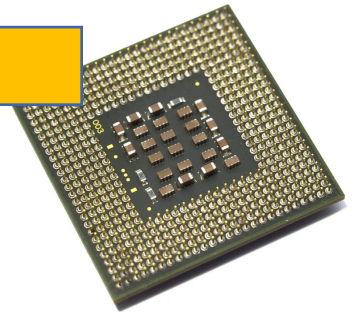
```
x ← input()
y ← input()
z ← x – y
a ← z > 0
output(a)
```

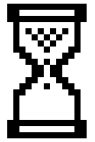
PASSAGGIO ALLA CPU

E una ad una vengono passate all'elaboratore ed eseguite

x ← input()
y ← input()
z ← x – y
a ← z > 0
output(a)

x ← input()

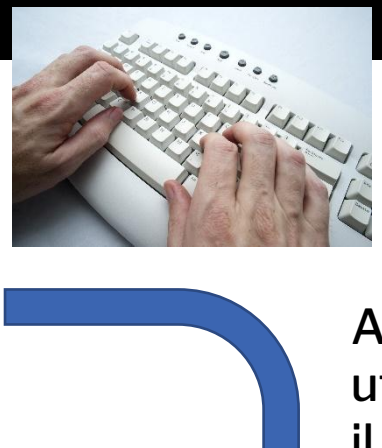




RESTITUZIONE DEL RISULTATO INTERMEDIO

```
x ← input()
y ← input()
z ← x – y
a ← z > 0
output(a)
```

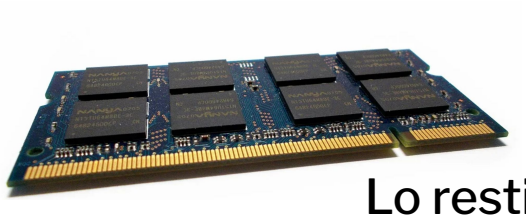
x ← input()



A seguito dell'input utente, la CPU restituisce il risultato (ovvero il numero che l'utente ha digitato sulla tastiera)

Ora, cosa se ne fa la CPU di questo risultato?

15

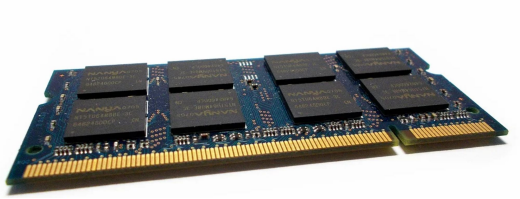


Lo restituisce alla memoria!

SALVATAGGIO DEL RISULTATO INTERMEDIO

Lo salva al suo interno!

Ma come?

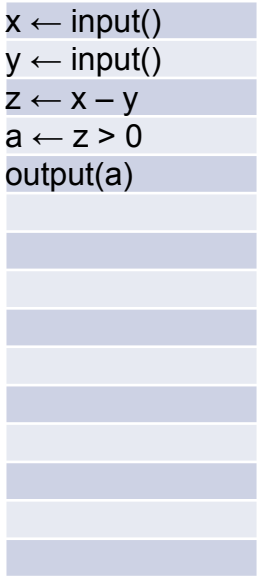


15

E la memoria cosa se ne fa del risultato intermedio?

La memoria è composta da tanti slot di memoria, che chiameremo **celle.**

In realtà, alcuni slot sono già occupati dalle istruzioni del programma in esecuzione



La memoria alloca una di queste celle per salvare il numero 15

CORRISPONDENZA CELLE MEMORIA - VARIABILI

Inoltre, la memoria aggiunge un'etichetta x vicino alla cella con il 15, ad indicare che quello spazio di memoria è allocato alla variabile che, nel programma, si chiama x

	x ← input()	0
	y ← input()	1
	z ← x – y	2
	a ← z > 0	3
	output(a)	4
		5
		6
		7
		8
		9
		10
		11
		12
		13
x	15	14

Alla fine, per il computer, il nome di una variabile è uno ***pseudonimo*** (***alias***) dello spazio di memoria in cui è custodito quel valore

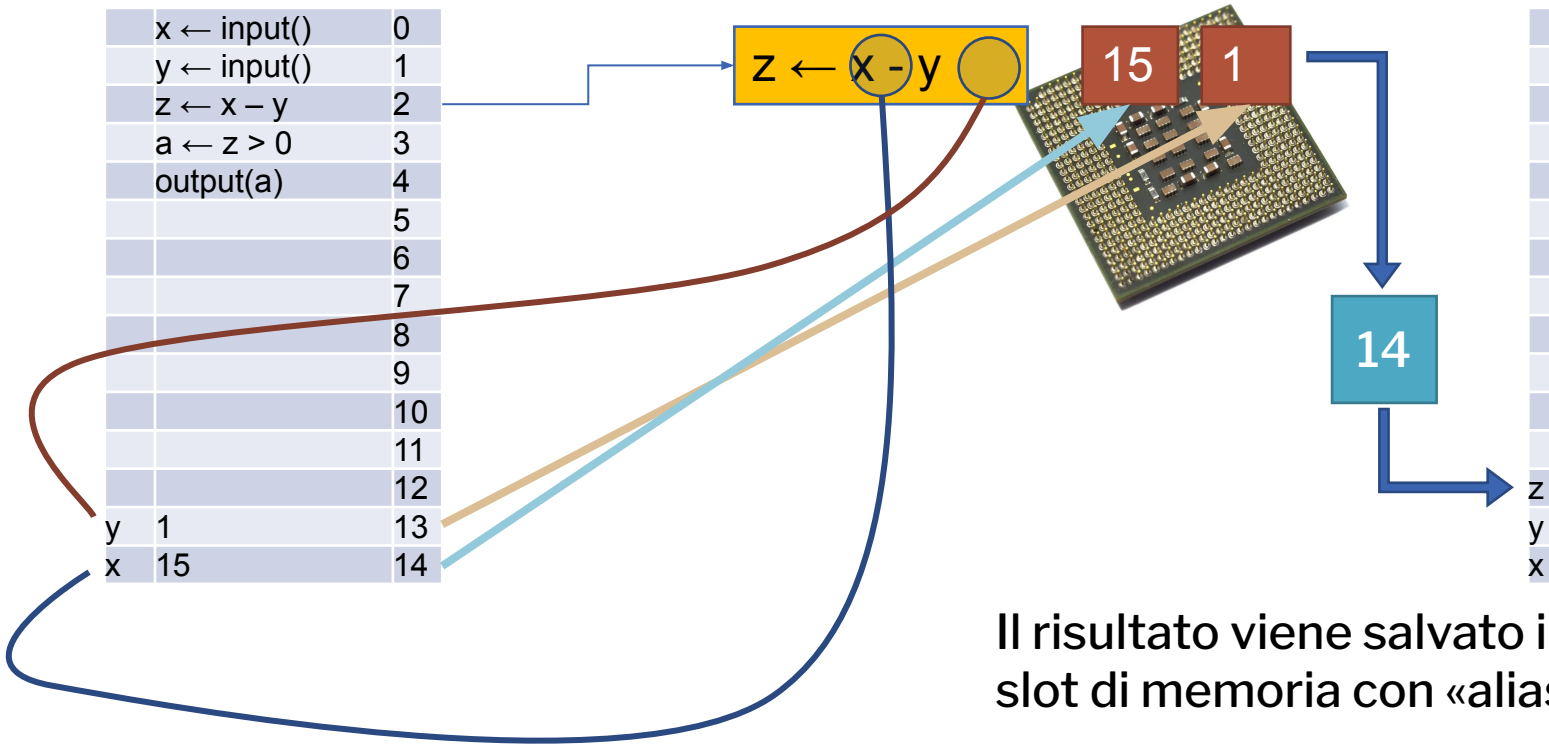
Normalmente, nella memoria ogni cella è identificata da un valore numerico

Vediamo come l'istruzione x ← input() si trovi nella cella 0

Il valore di x si trova in cella 14. Per la memoria, cella 14 o «x» hanno ora il medesimo significato

ANDIAMO AVANTI CON L'ESECUZIONE

La parte dell'input per y è uguale, vediamo cosa succede dopo



	x ← input()	0
	y ← input()	1
	z ← x – y	2
	a ← z > 0	3
	output(a)	4
		5
		6
		7
		8
		9
		10
		11
z	14	12
y	1	13
x	15	14

...

	x ← input()	0
	y ← input()	1
	z ← x – y	2
	a ← z > 0	3
	output(a)	4
		5
		6
		7
		8
		9
		10
a	vero	11
z	14	12
y	1	13
x	15	14

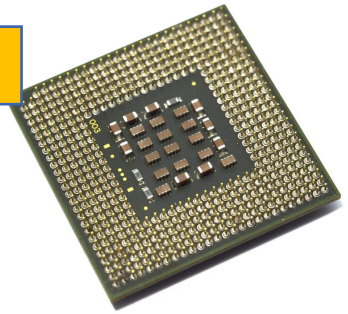
Il risultato viene salvato in uno slot di memoria con «alias» z

idem per l'istruzione successiva

IN CONCLUSIONE...

	x ← input()	0
	y ← input()	1
	z ← x – y	2
	a ← z > 0	3
	output(a)	4
		5
		6
		7
		8
		9
		10
		11
		12
y	1	13
x	15	14

output(a)



La CPU carica il contenuto di a e dà segnale al monitor di stamparlo

6. BASI DI PROGRAMMAZIONE

Accumulatore e registro istruzioni, compilazione, Assembly

VERSO IL MODELLO DI COMPUTER

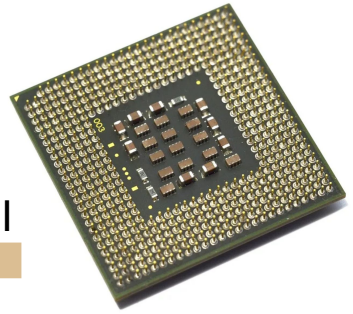
Come funziona l'esecuzione di un'istruzione all'interno dell'elaboratore

L'elaboratore è dotato di una minuscola memoria chiamata REGISTRO. Nel nostro caso, supponiamo che il registro possa memorizzare al massimo un intero o booleano. Questo registro serve a memorizzare il risultato prima di essere trasferito in memoria

Inoltre, supponiamo che esista un ulteriore registro, chiamato REGISTRO DELLE ISTRUZIONI, per memorizzare l'istruzione corrente che la CPU deve eseguire

REGISTRO ISTRUZIONI

0



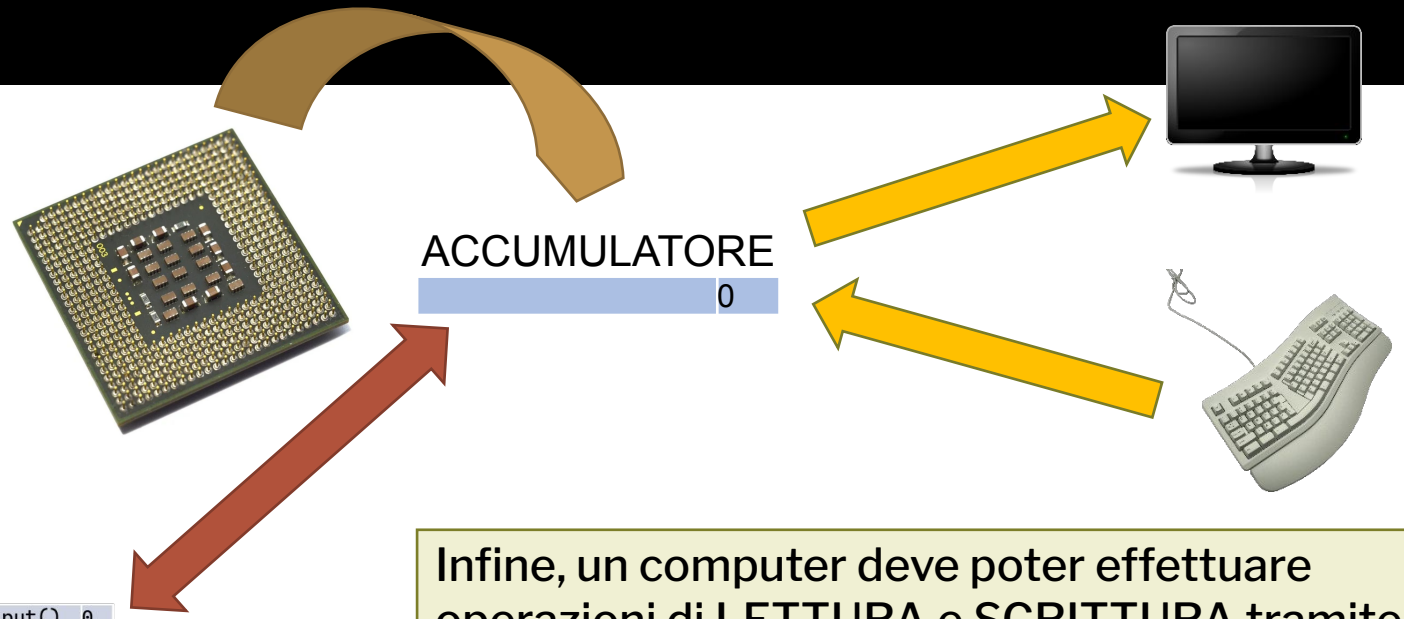
ACCUMULATORE

~~REGISTRO~~

0

Per non confondere i due registri, chiameremo il primo ACCUMULATORE

L'ACCUMULATORE



L'elaboratore usa l'accumulatore per salvare i risultati immediati dei suoi calcoli

Inoltre, l'accumulatore viene utilizzato dall'elaboratore per caricare dalla memoria i dati che devono venir elaborati.

	x ← input()	0
	y ← input()	1
	z ← x − y	2
	a ← z > 0	3
	output(a)	4
		5
		6
		7
		8
		9
		10
		11
		12
y	1	13
x	15	14

Infine, un computer deve poter effettuare operazioni di LETTURA e SCRITTURA tramite le periferiche I/O.

I dati ottenuti tramite lettura vengono caricati in accumulatore.

I dati in scrittura vengono caricati nelle periferiche di output a partire dall'accumulatore.

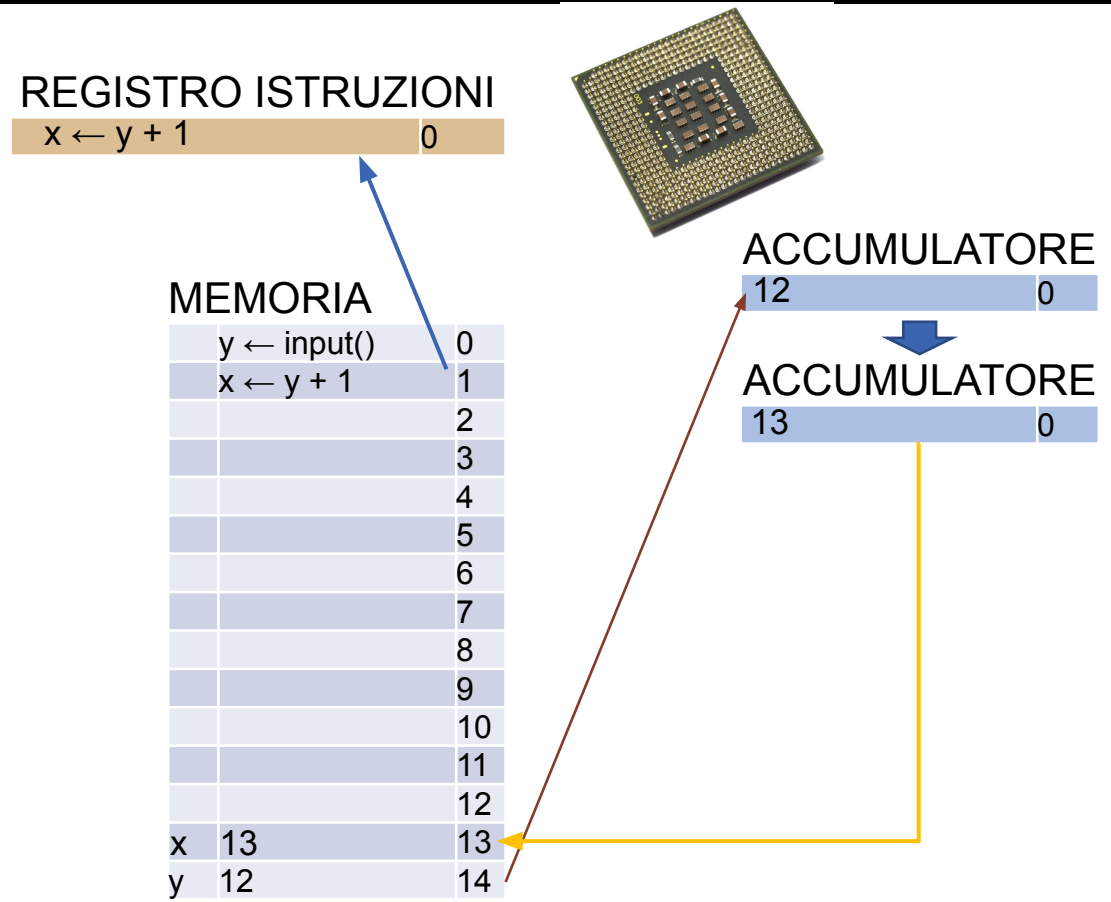
ISTRUZIONI, VISTE IN MANIERA UN PO' PIÙ FORMALE

Supponiamo di avere un'istruzione del genere

$x \leftarrow y + 1$

La CPU deve:

1. Leggere il valore di y dalla memoria (es. cella 14) caricarlo in accumulatore
2. Sommarci 1 e salvare il risultato nell'accumulatore
3. Spostare il risultato dall'accumulatore ad una cella vuota della memoria (es. la 13)



FORMALIZZIAMO I PASSAGGI

$x \leftarrow y + 1$

La CPU deve:

1. Leggere il valore di y dalla memoria (es. cella 14) caricarlo in accumulatore
2. Sommarci 1 e salvare il risultato nell'accumulatore
3. Spostare il risultato dall'accumulatore ad una cella vuota della memoria (es. la 13)

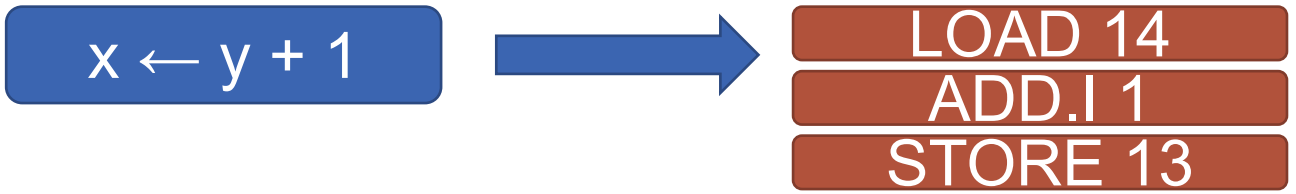
Questa è la rappresentazione dell'istruzione nel linguaggio del computer

LOAD 14	LOAD = carica in memoria
ADD.I 1	ADD.I = somma costante numerica
STORE 13	STORE = salva in memoria

COMPILAZIONE

Vediamo quindi come, una semplice istruzione come x ← y + 1 racchiuda in realtà 3 passaggi che devono essere svolti da un computer.

Il processo di conversione dell'istruzione



è un esempio di **COMPILAZIONE (o interpretazione, ad essere più precisi)**: il nostro codice è stato **convertito in una sequenza di istruzioni comprensibili dalla macchina**

ASSEMBLY

Nel nostro modello, questo linguaggio è assimilato al linguaggio macchina

In informatica, vi è ancora uno step successivo, che prevede la conversione in codice binario (che è la vera e propria **compliazione**)

Questo tipo di linguaggio «quasi macchina» è un esempio di

Ridurre le istruzioni in un linguaggio atomico semplice: una serie di comandi che siano direttamente comprensibili ed eseguibili da un elaboratore

FORMALIZZIAMO ANCORA DI PIÙ

Prima avevamo una situazione della memoria di questo tipo

	MEMORIA	
	y ← input()	0
	x ← y + 1	1
		2
		3
		4
		5
		6
		7
		8
		9
		10
		11
		12
		13
y	12	14

In realtà, in memoria dovremmo rappresentare l'istruzione in una sequenza di procedimenti lato macchina

Questa è una rappresentazione più realistica della memoria

	MEMORIA	
	y ← input()	0
	LOAD 14	1
	ADD.I 1	2
	STORE 13	3
		4
		5
		6
		7
		8
		9
		10
		11
		12
		13
y	12	14

UN PROBLEMA UN PO' PIÙ COMPLICATO

Ammettiamo che l'istruzione sia invece

MEMORIA

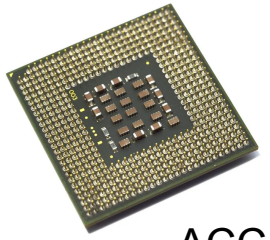
	y ← input()	0
	z ← input()	1
	??	2
		3
		4
		5
		6
		7
		8
		9
		10
		11
		12
z	30	13
y	12	14

x ← y + z

Come possiamo eseguire questa istruzione?

REGISTRO ISTRUZIONI

0



ACCUMULATORE

0

FORMALMENTE...

LOAD 14

ADD 13

STORE 12

Aggiungi il valore in accumulatore al **valore della cella di memoria 13**

VS

ADD.I 13

Aggiungi il **valore numerico** 13 all'accumulatore

6. BASI DI PROGRAMMAZIONE

Lo schema «definitivo» di computer

IL NOSTRO SCHEMA DI COMPUTER

Siamo ora pronti ad introdurre lo schema semplificato di computer

MEMORIA

	0
	1
	2
	3
	4
	5
	6
	7
	8
	9
	10
	11
	12
	13
	14

Abbiamo una memoria con un certo numero di celle

REGISTRO ISTRUZIONI

Abbiamo un registro istruzioni con spazio per un'istruzione singola

ACCUMULATORE

Abbiamo un accumulatore con spazio per un **naturale** singolo

NASTRO INPUT

Un «nastro» tramite cui viene passato l'input

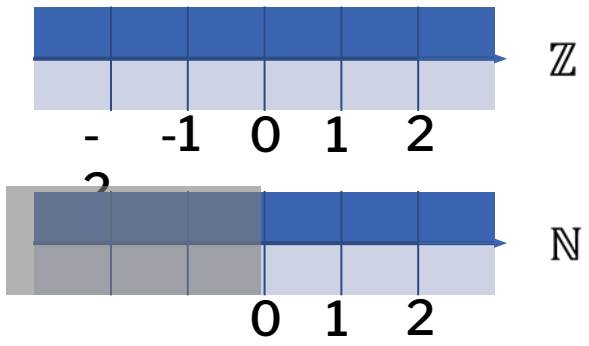
NASTRO OUTPUT

Un «nastro» su cui viene «stampato» l'output

CHE COSA COMPORTA IL FATTO DI LAVORARE CON NUMERI NATURALI ANZICHÉ INTERI?

Finora abbiamo visto paradigmi in cui si lavorava con numeri interi e booleani

- **Il nostro schema di computer non accetta booleani, né come istruzione, né come accumulatore/input/output**
- **Posso utilizzare solo numeri naturali ($\mathbb{N} = \{0,1,2, \dots\}$)**



Considerando $\mathbb{Z}$, posso sempre assicurarmi che qualsiasi sottrazione avrà sempre un risultato all'interno dello stesso insieme ($\mathbb{Z}$ è un insieme chiuso rispetto alla sottrazione)

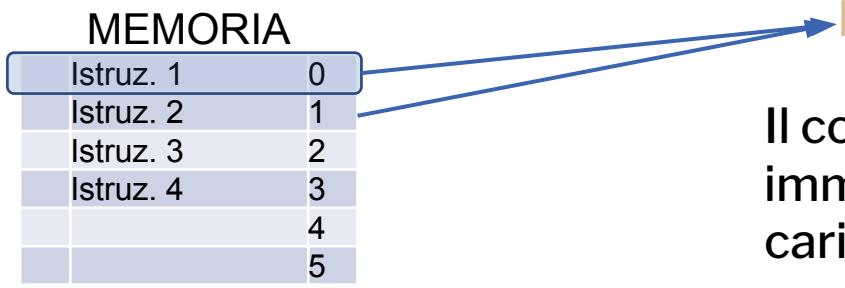
$\mathbb{N}$, invece, non è chiuso rispetto alla sottrazione. Che cosa succede se eseguo un'operazione del tipo $1 - 5$?

Non si causa alcun errore, restituisco 0

IL REGISTRO ISTRUZIONI

Abbiamo già visto memoria e l'accumulatore, vediamo il registro istruzioni

Le istruzioni sono contenute nella memoria.
La prima istruzione è in cella 0



Il computer carica la istruzione nel registro e la esegue

Il computer passa all'istruzione immediatamente successiva, la carica e la esegue

Il processo continua finché non eseguo l'ultima istruzione. Questa è un'istruzione speciale che chiamo

HALT

INPUT

Il nastro input è composto da un certo numero di celle

Ogni cella può contenere un intero

Ricordiamo le istruzioni che avevamo prima: x ← input()

Queste diventano

IN

STORE xx

MEMORIA	
Istruz. 1	0
Istruz. 2	1
IN	2
STORE 05	3
Istruz. 5	4
2	5

Il nastro è dotato di un puntatore, posizionato sulla prima cella

Quando il registro istruzioni ottiene un'istruzione di input (IN), la CPU carica in accumulatore il numero contenuto nella cella (2) e lo salva (STORE) nella cella di memoria 5

Subito dopo, avanza il puntatore alla cella successiva.

La prossima istruzione di input causerà la lettura del numero 10.

OUTPUT

L'output viene stampato su un nastro il quale contiene lo spazio per un solo intero

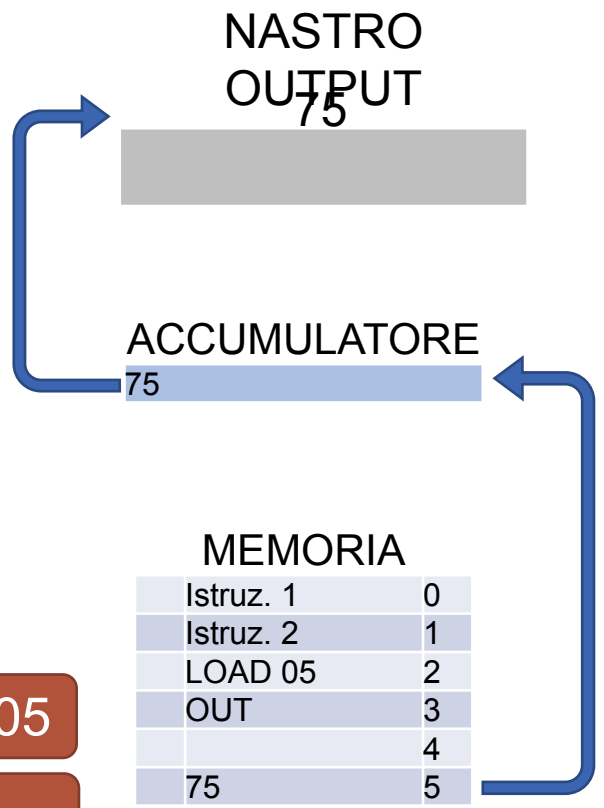
Prima avevamo istruzioni del tipo output(x)

Queste diventano LOAD yy OUT

Dove yy rappresenta la cella di memoria dove risiede la variabile x

E OUT stampa sul nastro il contenuto dell'accumulatore

Es. LOAD 05
OUT

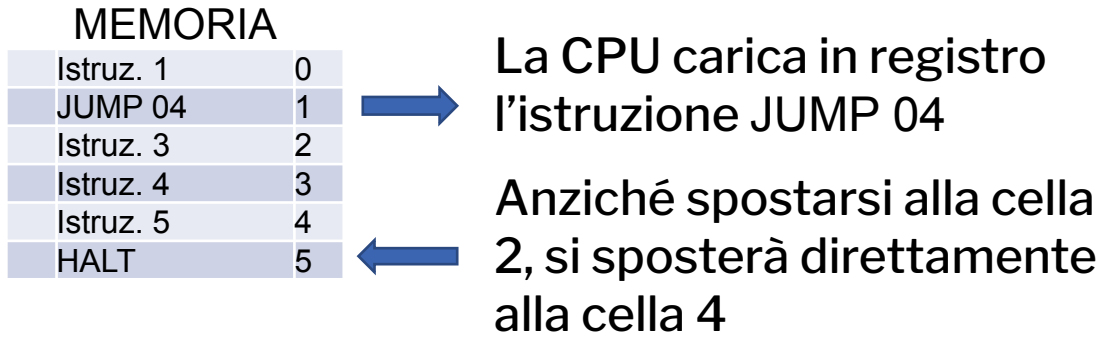


ULTERIORI ISTRUZIONI: SALTI

Per conlcudere, aggiungiamo ancora due istruzioni, che ci serviranno per esprimere i costrutti if e altri costrutti che impareremo più tardi

JUMP xx — Salta a istruzione in cella xx

JEZ xx — Salta a istruzione in cella xx se il valore nell'accumulatore è 0



OPERAZIONI MATEMATICHE

Abbiamo visto prima queste due istruzioni

ADD xx

ADD.I z

Abbiamo anche i corrispettivi per la sottrazione

SUB xx

SUB.I z

Sottrai all'accumulatore il valore contenuto nella cella xx

Sottrai all'accumulatore il numero naturale z

E MOLTIPLICAZIONE O DIVISIONE?

Come accadeva per le prime calcolatrici meccaniche, anche il nostro computer non è in grado di effettuare nativamente moltiplicazioni e divisioni in maniera diretta. Come possiamo implementare queste operazioni avendo a disposizione solamente addizione e sottrazione?

MOLTIPLICAZIONE

Iniziamo con la moltiplicazione.

Idee?

Qual è il primo concetto che si impara alle elementari per fare le moltiplicazioni?

Le tabelline

5 ×	
1	5
2	10
3	15
4	20
5	25
6	30
7	35
8	40
9	45
10	50

La moltiplicazione viene fatta secondo **addizione ripetuta**:

5×7

può essere pensato come «somma 5 per 7 volte»

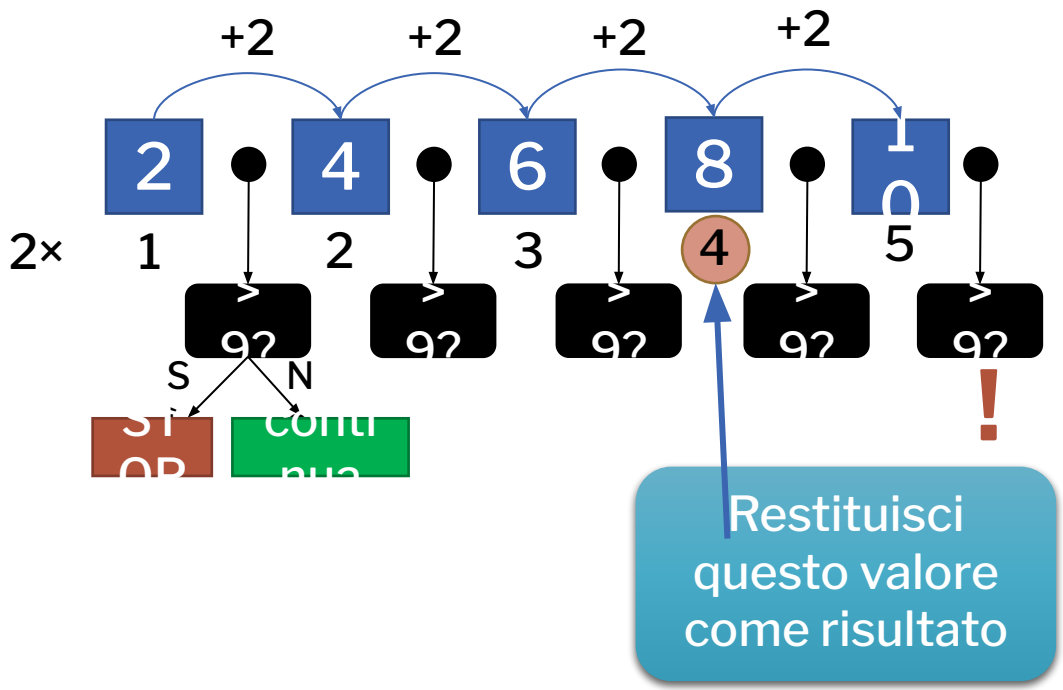
o

«somma 7 per 5 volte»

DIVISIONE

Possiamo pensare alla divisione in maniera analoga

Come implementiamo $9/2$ con sole somme e sottrazioni?
Nota: stiamo parlando di divisione intera, quindi il risultato deve essere 4 (con resto di 1)



CICLI

Gli esempi di moltplicazione e divisione sono propedeutici all'introduzione della nozione di ciclo, un costrutto che consiste nel ripetere un'azione per un certo numero di iterazioni

5×3

«Somma 5 **per** 3 iterazioni»

$9/2$

«Somma 2 **finché** il numero che ottieni non è maggiore di 9»

Ciclo FOR

Nr. iterazioni fissato

Ciclo WHILE

Nr. iterazioni condizonato ad uno specifico vincolo

MOLTIPLICAZIONE IN PSEUDOCODICE

```
moltiplicando ← 5
moltiplicatore ← 3
prodotto ← 0
FOR i ∈ {1,…,moltiplicando}:
  prodotto ← prodotto + moltiplicatore
output(prodotto)
```

Inizializzo il prodotto a 0

i = indice dell'iterazione
Assume tutti i valori nell'insieme indicato, in ordine sequenziale

Incremento il valore del prodotto del valore del moltiplicatore

Arrivati a questo punto, i viene incrementato di 1 e questa parte viene ri-eseguita

Quando i diventa maggiore di moltiplicatore, usciamo dal ciclo e eseguiamo l'istruzione successiva

MOLTIPLICAZIONE IN LINGUAGGIO «MACCHINA» (I)

	MEMORIA	
	IN	0
	STORE 100	1
	IN	2
	STORE 99	3
		4
		5
		6
		7
		8
		9
		10
		11
		12
		13
	…	
		98
moltiplicatore	3	99
moltiplicando	5	100

NASTRO INPUT

5	3			

In input abbiamo moltiplicando e moltiplicatore

1 Lettura input

Leggiamo i due input in sequenza e li conserviamo nelle celle finali della memoria

IN
STORE 100
IN
STORE 99

MOLTIPLICAZIONE IN LINGUAGGIO «MACCHINA» (II)

MEMORIA

	IN	0
	STORE 100	1
	IN	2
	STORE 99	3
	SUB 99	4
	STORE 98	5
		6
		7
		8
		9
		10
		11
		12
		13
	…	
prodotto	0	98
moltiplicatore	3	99
moltiplicando	5	100

2 Inizializzazione prodotto a 0

Non abbiamo alcuna istruzione per assegnare un valore arbitrario alla memoria.
L'unica cosa che possiamo fare è leggere un valore

- da input
- da memoria

Domanda: **cosa possiamo fare per dare un valore 0 ad una cella di memoria?**

Leggere un valore da input/memoria e sottrarlo con se stesso

ACCUMULATORE

~~3~~ 0

SUB 99

STORE 98

In accumulatore abbiamo attualmente il moltiplicatore letto da input

Ci sottraiamo il moltiplicatore stesso dalla memoria

Salviamo lo 0 nello spazio memoria del prodotto (cella 98)

MOLTIPLICAZIONE IN LINGUAGGIO «MACCHINA» (III)

MEMORIA		
	IN	0
	STORE 100	1
	IN	2
	STORE 99	3
	SUB 99	4
	STORE 98	5
		6
		7
		8
		9
		10
		11
		12
		13
…		
prodotto	0	98
moltiplicatore	3	99
moltiplicando	5	100

3 Inizio ciclo

Abbiamo ora bisogno di iniziare il ciclo. Ricordiamo: deve partire da 1 e arrivare a 5.

Domanda: **abbiamo un'istruzione che ci permette di controllare quando una variabile è arrivata ad un certo punto?**

NO, almeno se pensiamo al ciclo in questo modo

Qual è l'unica istruzione che ci permette di fare operazioni di comparazione?

JEZ
xx

Come possiamo ripensare il ciclo in modo che l'ultimo valore sia lo 0?

Invertiamo la successione:

1	2	3	4	5	6

5	4	3	2	1	0

MOLTIPLICAZIONE IN LINGUAGGIO «MACCHINA» (IV)

MEMORIA

	IN	0
	STORE 100	1
	IN	2
	STORE 99	3
	SUB 99	4
	STORE 98	5
	LOAD 100	6
		7
		8
		9
		10
		11
		12
		13
	…	
prodotto	0	98
moltiplicatore	3	99
moltiplicando	5	100

3 Inizio ciclo

Se vediamo il problema in questo modo (iterazione da 3 a 1), non abbiamo effettivamente bisogno di una variabile i per contare gli step del ciclo

Quale variabile utilizziamo al posto di i?

moltiplicando

Carichiamolo in accumulatore LOAD 100

Ora inizia la parte nuova: le istruzioni del ciclo devono essere ripetute fintantoché il moltiplicatore sarà diverso da 0

Arrivati alla fine delle istruzioni, torniamo indietro all'inizio del ciclo

JEZ yyy	xxx
JUMP xxx	
	yyy

Codice del ciclo

➡ Prima istruzione fuori dal ciclo

MOLTIPLICAZIONE IN LINGUAGGIO «MACCHINA» (V)

	MEMORIA	
	IN	0
	STORE 100	1
	IN	2
	STORE 99	3
	SUB 99	4
	STORE 98	5
	LOAD 100	6
Inizio ciclo	JEZ 15	7
Incr. prodotto	LOAD 98	8
	ADD 99	9
	STORE 98	10
Decr. indice	LOAD 100	11
	SUB.I 1	12
	STORE 100	13
Fine ciclo	JUMP 7	14
		15
		16
		17
	…	
prodotto	0	98
moltiplicatore	3	99
moltiplicando	5	100

3 Inizio ciclo

4 Codice interno al ciclo

5 Fine ciclo

Iniziamo con introdurre un JEZ in posizione 7 con riferimento non ancora identificato (ce l'avremo quando finiamo di scriv JEZ …)

prodotto ← prodotto + moltiplicatore

Lo possiamo esprimere cos LOAD 98 ADD 99 STORE 98

Abbiamo anche bisogno di aggiornare l'indice moltiplicando: LOAD 100 SUB.I 1 STORE 100

Dobbiamo tornare all'inizio del cic JUMP 7

Ora possiamo anche aggiornare il riferimento di JEZ. Ci posizioniamo sulla cella immediatamente successiva alla fine c JEZ 15

MOLTIPLICAZIONE IN LINGUAGGIO «MACCHINA» (VI)

	MEMORIA	
	IN	0
	STORE 100	1
	IN	2
	STORE 99	3
	SUB 99	4
	STORE 98	5
	LOAD 100	6
Inizio ciclo	JEZ 15	7
Incr. prodotto	LOAD 98	8
	ADD 99	9
	STORE 98	10
Decr. indice	LOAD 100	11
	SUB.I 1	12
	STORE 100	13
Fine ciclo	JUMP 7	14
	LOAD 98	15
	OU	16
	HAL	17
	T …	
prodotto	0	98
moltiplicatore	3	99
moltiplicando	5	100

6 Output del prodotto

Ora dobbiamo restituire in output il prodotto

Possiamo scrivere subito OUT?

No, perché in accumulatore abbiamo ancora il moltiplicando

Quindi: LOAD 98 OUT

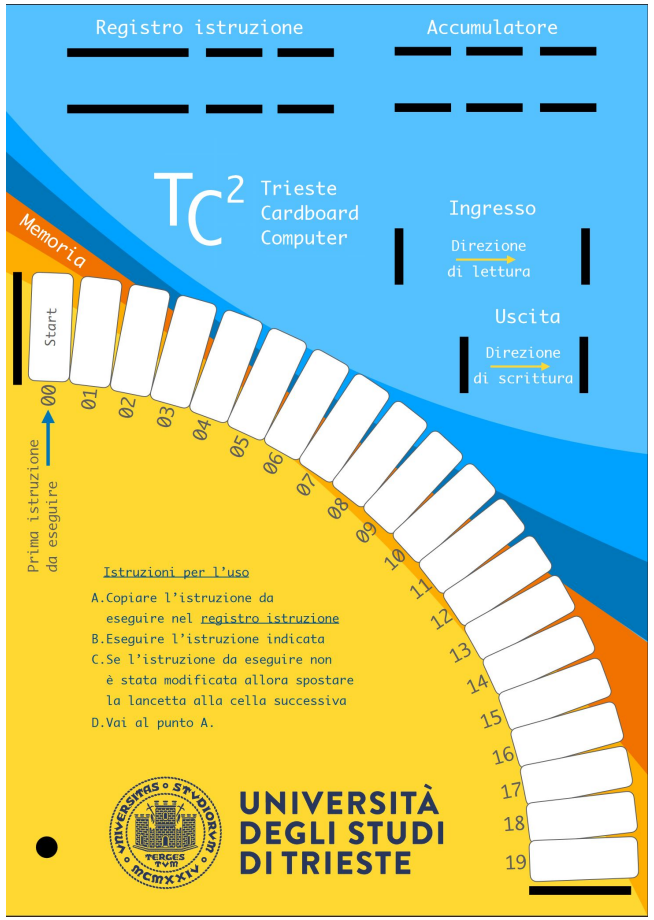
7 Terminazione programma

Manca un'ultimissima istruzione per concludere l'esecuzione de ma

HALT

ANDIAMO A PROVARE IL NOSTRO PROGRAMMA DAL VIVO!

https://cutt.ly/kEOHy0P



Il TC^2 (Trieste Cardboard Computer) è un'implementazione fisica del nostro schema di computer

È un adattamento di CARDIAC, sviluppato dalla Bell nel 1968 come strumento di apprendimento dell'informatica

La CPU è il nostro cervello, che esegue le somme e le sottrazioni richieste e si occupa di «spostare» i dati da memoria ai vari registri e nastri e viceversa.

DIZIONARIO DELLE ISTRUZIONI

TC² ragiona a numeri naturali. **Qualsiasi cosa presente in memoria**, registri, nastri, **è un numero intero**

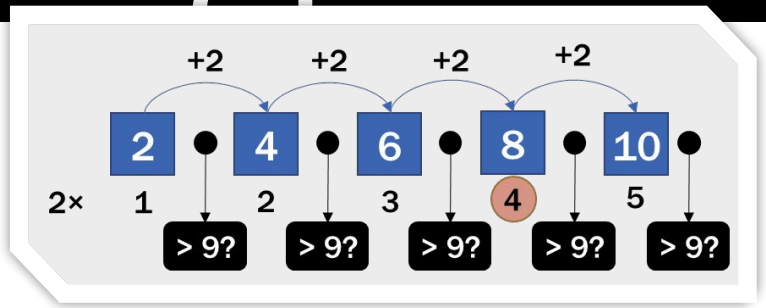
Conseguentemente, **dobbiamo convertire le istruzioni da testo in numero**

NB: siccome usiamo 3 numeri per la codifica, ADD.I e SUB.I ammettono al massimo 2 cifre, ne consegue che **TC² supporta solo naturali da 0**

Istruzione	Codifica	Significato
LOAD xy	0xy	Carica valore cella xy in accumulatore
STORE xy	1xy	Salva valore accumulatore in cella xy
ADD xy	2xy	Somma all'accumulatore il valore in cella xy
SUB xy	3xy	Sottrai all'accumulatore il valore in cella xy
ADD.I zw	4zw	Somma all'accumulatore il valore zw
SUB.I zw	5zw	Sottrai all'accumulatore il valore zw
IN	600	Leggi input e sposta la testina nastro input a destra
OUT	601	Scrivi valore accumulatore in nastro output
JEZ xy	7xy	Salta a cella xy se il valore dell'accumulatore è 0
JUMP xy	8xy	Salta a cella xy
HALT	900	Interrompi esecuzione programma

DIVISIONE INTERA IN PSEUDOCODICE

Il vostro compito ora è quello di progettare lo pseudocodice della divisione intera utilizzando l'idea che avevamo prima (somma ripetuta).
Dobbiamo ricalibrare il problema considerando che possiamo uscire dal ciclo solamente quando una data variabile è uguale a 0




```
dividendo ← 9
divisore ← 2
quoziente ← 0
WHILE dividendo > 0:
  dividendo ← dividendo - divisore
  quoziente ← quoziente + 1
quoziente ← quoziente - 1
output(quoziente)
```

Condizione di uscita dal ciclo: quando la condizione non è più rispettata (dividendo > 0 restituisce falso), esco dal ciclo e continuo con l'esecuzione normale del codice

Sottraggo il divisore dal dividendo

Siccome, all'uscita del ciclo, il quoziente è più grande del quoziente reale di 1 unità, lo devo decrementare di 1

DIVISIONE INTERA IN LINGUAGGIO «MACCHINA»

Ora, a voi la palla.

Dobbiamo convertire questo algoritmo in linguaggio macchina:

```
dividendo ← 9
divisore ← 2
quoziente ← 0
WHILE dividendo > 0:
  dividendo ← dividendo - divisore
  quoziente ← quoziente + 1
quoziente ← quoziente - 1
output(quoziente)
```

DIVISIONE INTERA – ALGORITMO COMPLETO

SOMMA DEI PRIMI N NUMERI NATURALI

Vogliamo costruire un programma che, dato un input *n*, restituisca la somma dei primi *n* numeri naturali

IF IN LINGUAGGIO MACCHINA

Ricordiamo questo codice da prima (con piccola modifica)

```
x ← input()
y ← input()
a ← x < y
IF a:
  z ← x + y
  output(z)
ELSE:
  z ← x - y
  output(z)
```

Come implementiamo questa istruzione?

$x < y \leftrightarrow x - y < 0 \leftrightarrow y - x > 0$

IN	0
STORE 19	1
IN	2
STORE 18	3
SUB 19	4
JEZ 13	5
LOAD 19	6
ADD 18	7
OUT	8
JUMP 13	9
LOAD 19	10
SUB 18	11
OUT	12
HALT	13
y	18
x	19

Nel TC^2 y – x mi consente di verificare se y è maggiore di x ed applicare il costrutto IF tramite un salto condizionale (JEZ):

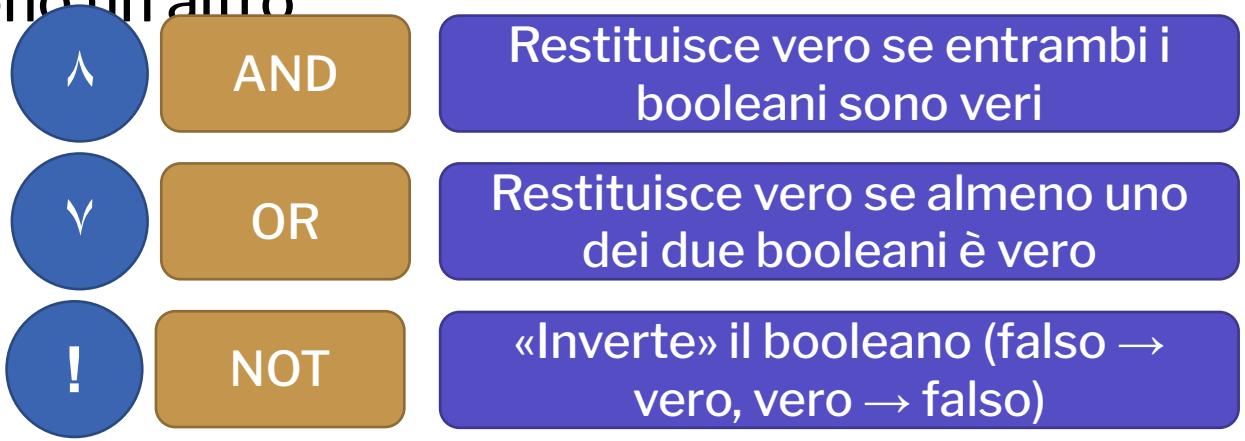
- Se y è maggiore di x, il valore di y-x (in accumulatore) è positivo; il JEZ dunque non si attiva ed eseguo il primo blocco
- Se x ≥ y, y-x è zero e attiva il JEZ

ISTRUZIONI LOGICHE

Abbiamo visto le istruzioni matematiche e imperative, compresi gli operatori aritmetici e di confronto.

A volte è necessario introdurre degli operatori che agiscano anche sui booleani.

Questi operatori sono i c.d. operatori logici. Accettano uno o due booleani e ne restituiscono un altro



QUANDO PUÒ SERVIRE UN OPERATORE LOGICO?

Lavoriamo per l'ufficio contabilità di un'azienda

Abbiamo un programma che invia un'email al supervisore quando i guadagni mensili superano i 10.000€

Ma SOLO SE tutte le entrate mensili sono state validate dal contabile

```
IF guagagni_mensili > 10000 AND entrate_validate:
  … invia email
```

L'ufficio marketing di un'assicurazione ci chiede di sviluppare un programma che invii una proposta di polizza a tutti i clienti che abbiano almeno altre due polizze contratte con l'assicurazione, oppure che non abbiano riportato sinistri

```
IF numero_polizze > 1 OR num_sinistri = 0:
  … invia proposta marketing
```

IMPLEMENTARE AND SUL TC^2

Esempio classico: $x \in \{2,3,4\}$

Come diventa utilizzando operatori logici e di confronto?

$$x \geq 2 \wedge x \leq 4$$

IF x ≥ 2 AND x ≤ 4:

JUMP se x < 2 o x > 4

Supponiamo x risieda in cella 100

x < 2:

(Voglio ottenere 0 se x<2)

LOAD 100	0
ADD.I 1	1
SUB.I 2	2
STORE 99	3

x > 4 :

(Voglio ottenere 0 se x>4)

SUB 99	4
ADD.I 4	5
ADD.I 1	6
SUB 100	7

AND:

ADD 99	8

IF:

JEZ xx	9
…	

IMPLEMENTARE OR SUL TC^2

Ora tocca a voi, implementate OR con la seguente istruzione

$$x \leq 10 \vee x \geq 15$$

```
IF x ≤ 10 OR x ≥ 15:
```

```
…do something
```

Soluzione a pagina seguente.

Se volete provare a risolvere in autonomia, notiamo una cosa:

- Mentre l'AND richiede che ENTRAMBE le condizioni si verifichino (quindi, devo andare a fare la «*SOMMA*» delle condizioni)
- L'OR richiede che si verifichi ALMENO una delle condizioni (quindi, non è effettivamente necessario fare la «somma» delle condizioni: come possiamo implementare ciò nel TC^2?)

IMPLEMENTAZIONE DI OR

Procediamo come prima, invertendo la condizione: $x \leq 10 \vee x \geq 15$ **diventa** $x > 10 \wedge x < 15$
Supponiamo sempre x risieda in cella 100 della memoria