Tutorato di Informatica

Foglio 2 - QtSpim e Assembly

Matematici I Anno

11 Marzo 2026

"What do you call a group of computer scientists? An Assembly."

Codifica delle Istruzioni e Formati MIPS

Esercizio 01

Testo: Si consideri l'istruzione `j xx`. Occupa più spazio in memoria la sua rappresentazione ASCII o l'istruzione macchina corrispondente dopo che è stata assemblata?

Spiegazione: La rappresentazione ASCII è lunga 4 caratteri, quindi occupa 4B, cioè 32 bit, che è esattamente pari alla lunghezza di una qualsiasi istruzione macchina in MIPS32. Dunque, entrambe le rappresentazioni occupano **lo stesso spazio**.

Esercizio 02

Testo: Interpretare il codice esadecimale `0x0232502A` come istruzione MIPS, ed indicare:

- La sua rappresentazione binaria
- L'opcode ed il formato (R-type, I-type o J-type)
- La rappresentazione simbolica dell'istruzione, completa del valore dei suoi parametri

Nota: usare l'appendice del libro a pagina A50.

Spiegazione:

- In rappresentazione binaria l'istruzione diventa: **`0000 0010 0011 0010 0101 0000 0010 1010`**
- L'opcode è dato dai 6 bit più significativi, ovvero 000000. A questo opcode corrispondono più operazioni, che si differenziano a seconda del valore dei 6 bit meno significativi (`funct`) rispettando dunque il formato **R-type**: `[opcode:6][rs:5][rt:5][rd:5][shamt:5][funct:6]`
- Innanzitutto ricaviamo i valori dei campi, separando la rappresentazione binaria come opportuno per il formato R-Type:
  - opcode: $000000 = 0_{10}$
  - rs: $10001 = 17_{10}$
  - rt: $10010 = 18_{10}$
  - rd: $01010 = 10_{10}$
  - shamt: $00000 = 0_{10}$

  - funct: $101010 = 42_{10}$

- L'istruzione corrispondente è identificata da opcode (0) e funct ($101010_2 = 42_{10}$), che corrispondono all'operazione `slt`, ovvero "set less than", che scrive in `rd` il risultato booleano del confronto `rs` < `rt`.
- La rappresentazione simbolica dell'operazione è `slt rd, rs, rt`, che nel nostro caso diventa **`slt $10, $17, $18`** (ovvero **`slt $t2, $s1, $s2`** utilizzando i nomi standard della convenzione hardware MIPS).

Esercizio 03

Testo: Indicare il formato dell'istruzione per effettuare la differenza tra due registri, ed esprimere esplicitamente l'operazione che effettua una sottrazione tra il valore nel registro $3 e quello nel registro $4, e deposita il risultato nel registro $2:

- in forma simbolica
- in forma binaria
- in forma esadecimale

Spiegazione:

- L'istruzione MIPS che effettua la differenza tra i contenuti di due registri è `sub rd, rs, rt` (vedi Appendice A, pag.A-56), che è di formato **R-type**.
- In forma simbolica: **`sub $2, $3, $4`**
- In forma binaria: **`000000 00011 00100 00010 00000 100010`**
- In forma esadecimale: **`0x00641022`**

Esercizio 04

Testo: Stiamo eseguendo il seguente programma:

Indirizzo	Istruzione (parziale)
`0x00400000`	`add $9, ...`
`0x00400004`	`sub $14, ...`
`0x00400008`	`lw $8, ...`
`0x0040000C`	`add $7, ...`

e ad un certo punto il Program Counter (PC) contiene il valore `0000 0000 0011 1111 1111 1111 1010 0100`. Inoltre sta per essere eseguita un'istruzione con opcode `000010`. Come devono essere i bit rimanenti di questa istruzione, affinché dopo venga eseguita l'istruzione `lw $8, ...`?

Spiegazione:

- L'opcode `000010` identifica un'istruzione di salto incondizionato (`j`), che appartiene al formato **J-type**.
- **Il problema dimensionale:** In MIPS, ogni istruzione è lunga esattamente 32 bit. Avendo utilizzato 6 bit per l'opcode, rimangono a disposizione solo **26 bit** per specificare l'indirizzo di destinazione. Tuttavia, gli indirizzi di memoria MIPS sono lunghi **32 bit**. Non potendo inserire 32 bit in uno spazio di 26, il processore ricostruisce l'indirizzo completo tramite una tecnica chiamata *Indirizzamento Pseudo-Diretto*, applicando due regole hardware.

- **Step 1 (I bit meno significativi):** Poiché in MIPS ogni istruzione è lunga 4 byte, tutte le istruzioni sono *allineate alla word*. Ciò significa che ogni indirizzo di istruzione valido è un multiplo di 4, e pertanto i suoi ultimi due bit in binario sono sempre `00`. L'hardware "risparmia" spazio omettendo questi due bit nell'istruzione e li aggiunge implicitamente a runtime eseguendo uno shift a sinistra di 2 posizioni (`« 2`). In questo modo, dai 26 bit se ne ottengono 28.

- **Step 2 (I bit più significativi):** Per ottenere i restanti 4 bit necessari a completare l'indirizzo a 32 bit, l'hardware "prende in prestito" i **4 bit più significativi dell'attuale Program Counter (PC+4)**. Matematicamente, dividendo lo spazio di memoria totale di 4 GB (2^{32}) per i 16 possibili valori dei primi 4 bit (2^4), si ottengono "quartieri" di memoria da 256 MB (2^{28} byte). Questa regola impone che un'istruzione `j` possa saltare ovunque, ma *solo all'interno dello stesso blocco da 256 MB* in cui si trova attualmente in esecuzione.

- **Calcolo pratico:** Vogliamo saltare all'indirizzo `0x00400008`. Scriviamolo in binario: `0000 0000 0100 0000 0000 0000 0000 1000`

  Per codificarlo nell'istruzione a 26 bit, eseguiamo il processo inverso che farà la CPU:

  1. Rimuoviamo i 2 bit meno significativi (`00`) derivanti dall'allineamento.
  2. Rimuoviamo i 4 bit più significativi (`0000`) che la CPU copierà automaticamente dal PC.

  Il risultato da inserire nell'istruzione è la parte centrale di 26 bit:
  `0000 0100 0000 0000 0000 0000 10`.

Esercizio 05

Testo: Qual è l'indirizzo della più lontana cella di memoria in cui è possibile saltare (in avanti) con una istruzione `bne` se PC = `0010 0010 1100 0011 0110 0010 0100 0000`?

Spiegazione: A differenza dell'istruzione `j` che specifica coordinate assolute, le istruzioni di salto condizionato (`bne`, `beq`) utilizzano l'*Indirizzamento Relativo al PC*. Esse non indicano "dove" andare, ma indicano una **distanza** rispetto alla posizione attuale. La CPU calcola l'indirizzo di arrivo utilizzando la formula:

$$\text{Target} = (PC + 4) + (\text{offset} \times 4)$$

Analizziamo i tre passaggi hardware per trovare il salto in avanti massimo consentito:

1. **Il punto di partenza (PC + 4):** Durante la fase di esecuzione, il PC è già stato incrementato per puntare all'istruzione successiva. Dato il PC = `0x22C36240` (versione esadecimale della stringa fornita), il nostro punto di partenza reale è:

   $$PC + 4 = \texttt{0x22C36244}$$

2. **L'Offset massimo in istruzioni:** Il formato I-type riserva **16 bit** per l'offset, interpretato come numero con segno in Complemento a Due (CA2). Il numero positivo più grande rappresentabile con 16 bit si ottiene con il bit di segno a `0` e gli altri 15 bit a `1` (`0111 1111 1111 1111`). In decimale, questo valore corrisponde a $2^{15} - 1 = \mathbf{32.767}$ istruzioni.

3. **La conversione da Istruzioni a Byte (Shift Left 2):** L'offset indica il numero di *istruzioni* da saltare, ma gli indirizzi di memoria si misurano in *byte*. Poiché ogni istruzione in MIPS occupa 4 byte, l'hardware moltiplica l'offset per 4 eseguendo uno shift logico a sinistra di 2 posizioni (`« 2`).

A livello binario, questo equivale semplicemente ad aggiungere due zeri (`00`) in coda alla sequenza. Partendo dal valore binario massimo calcolato al punto precedente:

`0111 1111 1111 1111`

Applicando lo shift a sinistra di 2 posizioni otteniamo la distanza esatta in byte:

`01 1111 1111 1111 1100`

Per convertire questo valore in esadecimale in modo semplice e diretto, raggruppiamo i bit a gruppi di 4 partendo da destra (aggiungendo zeri iniziali se necessario, poiché 4 bit = 1 cifra esadecimale):

$$\underbrace{0001}_{1}\ \underbrace{1111}_{F}\ \underbrace{1111}_{F}\ \underbrace{1111}_{F}\ \underbrace{1100}_{C}$$

La distanza massima convertita risulta quindi essere in modo diretto **`0x1FFFC`** *(che in decimale corrisponde esattamente a 131.068 byte)*.

Il Calcolo dell'Indirizzo Finale: Sommiamo ora la distanza massima al $(PC + 4)$:

`0x22C36244` + `0x0001FFFC`

Eseguendo l'addizione esadecimale cifra per cifra (ricordando di riportare l'unità al superamento del 16_{10}):

- `4 + C` (12) = 16_{10} $\implies$ scrivo **0**, riporto 1.
- `4 + F` (15) `+ 1` (riporto) = 20_{10} $\implies$ scrivo **4**, riporto 1 (poiché $20 - 16 = 4$).
- `2 + F` (15) `+ 1` (riporto) = 18_{10} $\implies$ scrivo **2**, riporto 1 (poiché $18 - 16 = 2$).
- `6 + F` (15) `+ 1` (riporto) = 22_{10} $\implies$ scrivo **6**, riporto 1 (poiché $22 - 16 = 6$).
- `3 + 1 + 1` (riporto) = **5**.
- I restanti digit `22C` rimangono inalterati.

L'indirizzo finale risulta essere **`0x22C56240`**. Espanso in binario, che è la risposta finale richiesta:

```
  0010 0010 1100 0011 0110 0010 0100 0100   (PC + 4)
+ 0000 0000 0000 0001 1111 1111 1111 1100   (Offset shifted)
-------------------------------------------
  0010 0010 1100 0101 0110 0010 0100 0000   (Target)
```

Esercizio 06

Testo: Determinare a quale istruzione macchina MIPS corrisponde la sequenza binaria:
`00000000011010010010000000100000`

Spiegazione:

- Dividiamo la sequenza di 32 bit nei campi previsti dal formato delle istruzioni MIPS. I primi 6 bit rappresentano l'opcode: **`000000`**.
- Un opcode pari a 0 indica un'istruzione aritmetico-logica di tipo **R-type**. Il formato da seguire è quindi `[opcode:6][rs:5][rt:5][rd:5][shamt:5][funct:6]`.
- Suddividiamo i rimanenti bit in blocchi:

  - opcode: $000000 = 0_{10}$
  - rs: $00011 = 3_{10}$ (registro `$3` o `$v1`)
  - rt: $01001 = 9_{10}$ (registro `$9` o `$t1`)
  - rd: $00100 = 4_{10}$ (registro `$4` o `$a0`)
  - shamt: $00000 = 0_{10}$ (nessuno scorrimento)
  - funct: $100000 = 32_{10}$

- Il campo `funct` con valore decimale 32 corrisponde all'istruzione di addizione (`add`).
- Ricordando che la sintassi in assembly prevede l'ordine `add rd, rs, rt`, l'istruzione finale è: **`add $4, $3, $9`** (oppure utilizzando la convenzione dei nomi **`add $a0, $v1, $t1`**).

Esercizio 07

Testo: Determinare a quale istruzione macchina MIPS corrisponde la sequenza binaria: `00100010111010110000000001100000`

Spiegazione:

- Isoliamo i primi 6 bit per ottenere l'opcode: **`001000`**.
- Questo opcode in binario corrisponde al valore decimale 8, che identifica l'istruzione `addi` (add immediate). Trattandosi di un'istruzione che utilizza una costante, il formato è **I-type**: `[opcode:6][rs:5][rt:5][imm:16]`.
- Suddividiamo i bit secondo il formato I-type:
  - opcode: $001000 = 8_{10}$ (`addi`)
  - rs: $10111 = 23_{10}$ (registro `$23` o `$s7`)
  - rt: $01011 = 11_{10}$ (registro `$11` o `$t3`)
  - imm: $0000000001100000 = 96_{10}$
- La sintassi dell'istruzione assembly richiede l'ordine `addi rt, rs, imm`.
- L'istruzione finale è quindi: **`addi $11, $23, 96`** (oppure **`addi $t3, $s7, 96`**).

Esercizio 08

Testo: A quale istruzione macchina MIPS corrisponde il codice esadecimale: `0x8fa40000`

Spiegazione:

- Convertiamo innanzitutto il valore esadecimale espandendolo in binario (ogni cifra esadecimale corrisponde a 4 bit):
  `8fa40000` → **`1000 1111 1010 0100 0000 0000 0000 0000`**
- Isoliamo i primi 6 bit (opcode): **`100011`**. Questo valore binario corrisponde a 35_{10}, che identifica l'istruzione `lw` (load word). Anch'essa appartiene al formato **I-type**.
- Suddividiamo la sequenza come fatto in precedenza:
  - opcode: $100011 = 35_{10}$ (`lw`)
  - rs: $11101 = 29_{10}$ (registro base, `$29` o `$sp`)

- rt: $00100 = 4_{10}$ (registro destinazione, `$4` o `$a0`)
- imm: $0000000000000000 = 0_{10}$ (offset)

- La sintassi per l'istruzione di caricamento dalla memoria è `lw rt, offset(rs)`.
- L'istruzione finale è quindi: **`lw $4, 0($29)`** (oppure **`lw $a0, 0($sp)`**).

Programmazione Assembly

Esercizio 09

Testo: Scrivere un programma che calcoli la somma del valore che è memorizzato nel registro $17, e del valore memorizzato nella locazione di memoria che si trova 14 word più avanti dell'indirizzo specificato al registro $16, e memorizzare il risultato 3 word di memoria più avanti rispetto alla locazione attuale del secondo operando. Nota: Vedi le istruzioni `lw` e `sw`, e ricorda che puoi usare la sintassi $n(x)$.

Spiegazione: Analogamente a quanto visto nell'esercizio precedente, in MIPS le distanze (offset) relative alla memoria devono sempre essere convertite in byte. Poiché una "word" (parola) in MIPS è composta da 4 byte, per spostarci di un certo numero di celle dobbiamo moltiplicare il valore per 4:

1. Sommare il contenuto del registro $17 al valore in memoria che sta 14 word ($14 \times 4 = 56$ byte) dopo l'indirizzo indicato in $16.
2. Salvare il risultato in memoria ad un indirizzo che sta 3 word ($3 \times 4 = 12$ byte) dopo quello del secondo addendo. Essendo il secondo addendo all'offset 56, la posizione finale rispetto all'indirizzo base originale in $16 sarà: $56 + 12 = 68$ byte.

Una possibile soluzione è:

```
lw   $8, 56($16)    # copio in $8 il valore a 56B (14 word) da base $16
add  $9, $17, $8    # sommo $8 con $17 e scrivo il risultato in $9
sw   $9, 68($16)    # salvo risultato a 68B (14+3 = 17 word) da base $16
```

Esercizio 10

Testo: I valori relativi alle variabili della dimensione di una word a, b, c, d sono memorizzati di seguito in memoria a partire dall'indirizzo specificato dal contenuto del registro $10. Scrivere la sequenza di istruzioni assembly che aggiunge la costante 10 alle variabili a, b, c, d e salva i nuovi valori delle variabili in memoria.

Spiegazione: Come esposto, i dati contigui in memoria necessitano di salti di 4 byte (una word) per passare da un elemento al successivo. L'approccio con ciclo ("pointer arithmetic") è di fondamentale importanza perché incrementando l'indirizzo base di `4` byte ad ogni iterazione, la CPU punta fisicamente al dato adiacente.
Soluzione sequenziale:

```
lw   $8, 0($10)
addi $8, $8, 10
sw   $8, 0($10)
lw   $8, 4($10)
addi $8, $8, 10
```

```
sw   $8, 4($10)
lw   $8, 8($10)
addi $8, $8, 10
sw   $8, 8($10)
lw   $8, 12($10)
addi $8, $8, 10
sw   $8, 12($10)
```

Soluzione con ciclo:

```
add  $11, $zero, $zero  # inizializzo a 0 il contatore di cicli in $11
addi $9,  $zero, 4      # numero di cicli da eseguire in $9
ciclo:
lw   $8,  0($10)
addi $8,  $8, 10
sw   $8,  0($10)
addi $10, $10, 4        # pointer arithmetic: passo alla word successiva (+4 byte)
addi $11, $11, 1        # incremento il contatore iterazioni
bne  $11, $9, ciclo     # se il contatore non è 4, salta all'etichetta 'ciclo'
```

Esercizio 11

Testo: In QtSpim, scrivere (manualmente) in memoria, nella sezione Data, la stringa seguente: `"Hello world!"`. La stringa deve apparire leggibile nel giusto ordine nell'interfaccia del programma. NOTA: Usare la tabella ASCII (extended) per la codifica dei caratteri.

Spiegazione: Per scrivere la stringa sono necessari 12 caratteri, ciascuno da 1 byte, che raggruppati formano 3 word da 4 byte ciascuna. Apriamo la sezione Data e inseriamo i valori ASCII rispettando l'endianness. I sistemi come QtSpim (spesso eseguiti su architetture x86) utilizzano il formato **Little Endian**, dove il byte meno significativo viene memorizzato all'indirizzo più basso. Per questo motivo, inserendo un'intera "word" alla volta, i caratteri all'interno della singola word dovranno essere digitati in ordine inverso:

1. `"Hell"` → `(l)(l)(e)(H)` = **`6C 6C 65 48`**
2. `"o wo"` → `(o)(w)( )(o)` = **`6F 77 20 6F`**
3. `"rld!"` → `(!)(d)(l)(r)` = **`21 64 6C 72`**

Esercizio 12

Testo: In QtSpim, inserire manualmente due valori nei registri $t0 e $t1. Poi scrivere un programma assembly che calcoli la somma dei due numeri presenti nei registri $t0 e $t1 e metta il risultato in $t2.

Spiegazione:

- Per cambiare i valori basta cliccare col destro sui registri e inserire i numeri indicati (modello di calcolo *register-register* tipico del *datapath* MIPS).
- Il programma consiste in un'unica operazione aritmetica che coinvolge l'ALU.
- Utilizziamo la direttiva `.text` per indicare la sezione di codice e `.globl main` per rendere l'entry point visibile al simulatore.

Spiegazione:

```
.text
.globl main

main:
    # Calcola $t2 = $t0 + $t1
    add $t2, $t0, $t1

    # Terminazione standard (opzionale in QtSpim ma buona pratica)
    li $v0, 10
    syscall
```

Nota: Caricando questo file in QtSpim, assicurati di aver inserito i valori in $t0 e $t1 prima di premere "Run" o di eseguire il programma passo-passo (Step).

Esercizi su Emulatore QtSpim

Esercizio 13: La mia prima somma assembly

Scaricare il programma `la_mia_prima_somma_Assembly.asm` da Moodle, caricarlo ed eseguirlo passo passo in QtSpim. Annotare, ad ogni passaggio (istruzione), il valore del PC e dei registri usati, ed inoltre monitorare cosa succede nelle sezioni `.text` e `.data`.

Spiegazione: Un programma Assembly MIPS ben strutturato è diviso in sezioni tramite direttive (parole che iniziano con il punto, come `.text`). La sezione `.text` indica all'assemblatore che quello che segue è codice eseguibile. Per interagire con l'emulatore (es. stampare a video o terminare il programma) usiamo le **syscall** (System Call). Il funzionamento delle syscall richiede di inserire un codice operativo specifico nel registro `$v0` e gli eventuali argomenti nel registro `$a0`. Inoltre, usiamo `li` (Load Immediate), che è una *pseudo-istruzione*: non esiste fisicamente nell'hardware MIPS, ma l'assemblatore QtSpim la traduce in automatico in istruzioni base (come `ori` o `addiu`) per semplificarci la vita.

```
.text           # Direttiva: indica l'inizio della sezione del codice eseguibile
.globl main     # Rende l'etichetta 'main' visibile globalmente a QtSpim

main:
    # 1. Fase di Inizializzazione
    # Usiamo la pseudo-istruzione 'li' (Load Immediate) per caricare costanti.
    # L'hardware reale richiederebbe passaggi più lunghi per caricare un numero a 32 bit,
    # ma QtSpim traduce 'li' in automatico.
    li $t0, 7        # Carica il valore decimale 7 nel registro temporaneo $t0
    li $t1, 5        # Carica il valore decimale 5 nel registro temporaneo $t1

    # 2. Fase di Calcolo
    # L'istruzione 'add' richiede sempre 3 registri: destinazione, sorgente1, sorgente2.
    add $t2, $t0, $t1 # Somma $t0 e $t1. Il risultato (12) viene salvato in $t2.

    # 3. Fase di Output (Stampa a video)
    # Per dire a QtSpim di stampare un intero, dobbiamo seguire le convenzioni:
    # - Il codice per la syscall "print_int" è 1. Lo carichiamo in $v0.
    # - Il numero da stampare DEVE trovarsi in $a0 (registro per gli argomenti).
    li $v0, 1        # Preparo la syscall 1 (print_int)
    move $a0, $t2    # Copio il risultato della somma da $t2 al registro argomenti $a0
    syscall          # "Sveglio" il sistema operativo simulato per eseguire l'azione

    # 4. Fase di Chiusura
    # Se non diciamo esplicitamente al programma di fermarsi, QtSpim cercherà di
    # eseguire la memoria successiva, causando errori. Il codice di exit è 10.
    li $v0, 10       # Preparo la syscall 10 (exit)
```

```
    syscall          # Termina l'esecuzione in modo pulito
```

Esercizio 14: Ricerca all'interno di un array

Completare il programma `ricerca_in_array.asm` che cerca il valore 5 all'interno di un array di 10 numeri interi. Farlo girare su QtSpim, passo passo e verificarne il funzionamento. Per verificare il funzionamento dell'intero programma, modificare il vettore perché contenga, o meno, il valore che si sta cercando.

Spiegazione: Qui introduciamo la direttiva `.data`, che serve a riservare spazio nella memoria dati (RAM) per le nostre variabili statiche (come array e stringhe). Per scorrere l'array, usiamo l'**aritmetica dei puntatori** vista negli esercizi 6 e 7: partiamo dall'indirizzo di base e aggiungiamo 4 byte ad ogni iterazione, poiché ogni numero è salvato come una `.word` (32 bit = 4 byte). Utilizziamo `la` (Load Address), un'altra pseudo-istruzione fondamentale, che calcola l'indirizzo di memoria di un'etichetta e lo salva in un registro.

```
.data           # Direttiva: inizio della sezione dei dati in memoria (RAM)
# Definisco l'array. '.word' alloca 4 byte per ogni numero elencato.
array:          .word 3, 7, 1, 4, 9, 6, 8, 2, 0, 5
# Stringhe per l'output. '.asciiz' salva la stringa e aggiunge un terminatore NULL (0x00)
str_trovato:    .asciiz "Trovato\n"
str_non_trovato: .asciiz "Non trovato\n"

.text
.globl main

main:
    # 1. Preparazione dei registri di controllo
    li $t0, 0          # Contatore del ciclo (indice i = 0)
    li $t1, 10         # Lunghezza dell'array (condizione di uscita)
    li $t2, 5          # Target: il numero che stiamo cercando

    # 'la' (Load Address) copia l'indirizzo di memoria in cui inizia 'array' in $t3.
    # Questo è il nostro puntatore base.
    la $t3, array

ciclo:
    # 2. Condizione di terminazione del ciclo
    # Se abbiamo controllato 10 elementi (i == 10), abbiamo finito l'array senza
    # trovare il numero. Saltiamo all'etichetta 'non_trovato'.
    beq $t0, $t1, non_trovato

    # 3. Lettura dalla memoria
    # lw (Load Word) carica il dato dalla RAM al registro.
    # 0($t3) significa: prendi l'indirizzo in $t3, aggiungi 0, e leggi la word.
    lw $t4, 0($t3)     # Ora $t4 contiene l'i-esimo elemento dell'array

    # 4. Verifica della condizione di ricerca
    # Se il valore letto ($t4) è uguale al nostro target ($t2), abbiamo successo!
    beq $t4, $t2, trovato

    # 5. Aggiornamento dei puntatori e iterazione
    # Poichè ogni numero occupa 4 byte in memoria, DEVO aggiungere 4 all'indirizzo
    # base per far puntare $t3 alla cella successiva.
    addi $t3, $t3, 4   # Sposta il puntatore avanti di 4 byte (1 word)
    addi $t0, $t0, 1   # Incrementa il contatore delle iterazioni (i++)
    j ciclo            # Salta incondizionatamente all'inizio del ciclo

trovato:
    # Il codice 4 di syscall stampa una stringa terminata da NULL.
```

```
    # Richiede che l'indirizzo della stringa sia in $a0.
    li $v0, 4
    la $a0, str_trovato
    syscall
    j fine              # Salto alla fine per evitare di eseguire 'non_trovato'

non_trovato:
    li $v0, 4
    la $a0, str_non_trovato
    syscall

fine:
    li $v0, 10          # Syscall 10: termina programma
    syscall
```

Esercizio 15: Interazione utente e Salto Condizionato

Scrivere un programma in assembly che esegua le seguenti operazioni:

1. Leggere due numeri interi (num1 e num2) usando l'apposita syscall
2. Confrontare i due valori
3. Stampare a schermo il più piccolo dei due

dopodiché caricarlo su QtSpim e testarlo.
Consiglio: puoi partire dal file dell'esercizio precedente e prendere spunto.

Spiegazione: Per leggere l'input da tastiera usiamo la syscall `5` (`read_int`). Una cosa fondamentale da capire è che **tutte le syscall che restituiscono un valore, lo mettono in $v0**. Pertanto, appena leggiamo il primo numero, dobbiamo subito "metterlo al sicuro" (spostandolo in un registro come `$t0`) prima di fare un'altra syscall, altrimenti il nuovo input sovrascriverà il precedente in `$v0`. Per il confronto usiamo `blt` (Branch on Less Than), una pseudo-istruzione che si traduce in un blocco logico 'slt' + 'bne'.

```
.data
prompt1:    .asciiz "Inserisci il primo numero: "
prompt2:    .asciiz "Inserisci il secondo numero: "
resultMsg:  .asciiz "Il minimo è: "

.text
.globl main

main:
    # 1. Richiesta del primo numero (Stampa prompt)
    li $v0, 4             # Syscall 4: print_string
    la $a0, prompt1       # Carico l'indirizzo della prima stringa in $a0
    syscall

    # 2. Lettura del primo numero
    li $v0, 5             # Syscall 5: read_int (legge da console)
    syscall               # Dopo questa istruzione, l'input dell'utente è in $v0

    # SALVATAGGIO: Sposto subito il valore da $v0 a $t0.
    # Se non lo facessi, la prossima syscall sovrascriverebbe il mio input!
    move $t0, $v0

    # 3. Richiesta del secondo numero (Stampa prompt)
    li $v0, 4
    la $a0, prompt2
    syscall
```

```
27
28      # 4. Lettura del secondo numero
29      li $v0, 5
30      syscall
31      move $t1, $v0         # Metto al sicuro il secondo numero in $t1
32
33      # 5. Logica decisionale (Confronto)
34      # blt (Branch Less Than) valuta se il primo argomento è minore del secondo.
35      # Se $t0 < $t1, il minimo è già in $t0. Quindi salto direttamente alla fase di stampa.
36      blt $t0, $t1, stampa_minimo
37
38      # Se arrivo a questa riga, significa che $t0 >= $t1, quindi il minimo è $t1.
39      # Copio il valore di $t1 in $t0. In questo modo, garantisco che alla fine
40      # di questo blocco logico, il valore minimo si trovi SEMPRE nel registro $t0.
41      move $t0, $t1
42
43  stampa_minimo:
44      # 6. Output del risultato testuale
45      li $v0, 4
46      la $a0, resultMsg
47      syscall
48
49      # 7. Output del numero
50      # Ora stampiamo il valore calcolato. La syscall 1 (print_int) richiede
51      # il numero da stampare in $a0. Dato che abbiamo assicurato che il minimo
52      # si trovi in $t0, lo spostiamo in $a0.
53      li $v0, 1
54      move $a0, $t0
55      syscall
56
57      # 8. Chiusura programma
58      li $v0, 10            # Syscall 10: exit
59      syscall
```

Esercizio 16: Somma agli estremi di un array

Testo: Scrivere un programma in assembly MIPS che allochi in memoria un array chiamato `vet` contenente i seguenti 5 numeri interi: 2, 4, 6, 8, 10. Il programma deve poi:

1. Caricare l'indirizzo base dell'array in un registro temporaneo.
2. Leggere dalla memoria il primo elemento e l'ultimo elemento, salvandoli in due registri distinti.
3. Calcolare la somma di questi due valori e memorizzare il risultato finale in un terzo registro.

Dopodiché, salvare il codice in un file `.asm`, caricarlo su QtSpim ed eseguirlo passo passo per verificare che il valore finale nel registro di destinazione sia effettivamente 12.

Spiegazione: Per risolvere questo esercizio è fondamentale padroneggiare l'aritmetica dei puntatori in MIPS. Poiché ogni numero intero definito con la direttiva `.word` occupa esattamente 4 byte in memoria, per accedere all'elemento in posizione i (partendo da 0) dobbiamo spostarci dall'indirizzo base di $i \times 4$ byte.

- Il primo elemento (indice 0, valore 2) si trova all'offset $0 \times 4 = \mathbf{0}$.
- Il quinto elemento (indice 4, valore 10) si trova all'offset $4 \times 4 = \mathbf{16}$.

Errore comune: È facile confondersi e pensare che per prendere l'ultimo elemento basti usare la sua posizione "assoluta" o il suo indice puro. Ricorda sempre di moltiplicare per 4!

```assembly
.data
vet: .word 2, 4, 6, 8, 10  # Vettore di 5 interi (ogni intero = 4 byte)

.text
.globl main

main:
    la $t0, vet       # Carica in $t0 l'indirizzo base del vettore
    lw $t1, 0($t0)    # Carica il primo elemento (vet[0] = 2) in $t1
    lw $t2, 16($t0)   # Carica il quinto elemento (vet[4] = 10) in $t2
    add $t3, $t1, $t2 # Somma $t1 + $t2, il risultato finale (12) va in $t3

    # Terminazione standard del programma
    li $v0, 10
    syscall
```

Esercizi Extra Per i Più Coraggiosi (Lab 2)

01. L'Isomorfismo tra Dati e Istruzioni (Il Codice Automodificante)

Testo: Abbiamo visto che in MIPS ogni istruzione è codificata come una sequenza di 32 bit, esattamente come un numero intero (una `.word`). Dal punto di vista algebrico, c'è qualche differenza tra un'istruzione e un dato? Sarebbe teoricamente possibile per un programma MIPS calcolare un numero, salvarlo in memoria, e poi "saltare" su quel numero ed eseguirlo come se fosse codice? Prova a costruire un esempio.

Spiegazione (L'Architettura di Von Neumann e i Numeri di Gödel): Nel 1931, Kurt Gödel distrusse il sogno formalista di Hilbert dimostrando i Teoremi di Incompletezza. Per farlo, inventò la *Gödelizzazione*: una funzione iniettiva che mappa ogni formula della logica formale in un numero naturale. In questo modo, le formule logiche possono "parlare" di altri numeri, e quindi di altre formule logiche.

I computer moderni (macchine di Von Neumann) sono l'implementazione fisica di questa idea. La memoria non fa distinzione ontologica tra "Dati" e "Istruzioni". Entrambi sono stringhe binarie (numeri di Gödel fisici). In Assembly, l'istruzione `add $t0, $t1, $t2` viene assemblata nell'esadecimale `0x012A4020`. Ma `0x012A4020` è anche il numero decimale 19.546.144.

La Dimostrazione Pratica (Self-Modifying Code): Se il sistema operativo non blocca l'area di memoria (per motivi di sicurezza, tramite meccanismi come il bit *NX - No-eXecute* che impedisce di eseguire codice residente nel segmento dati), un programma Assembly può usare un'istruzione `sw` (Store Word) per sovrascrivere le istruzioni successive direttamente nel segmento `.text`. Il programma può letteralmente calcolare la sua prossima istruzione eseguendo operazioni aritmetiche, salvarla in memoria, e quando il Program Counter la raggiunge, la CPU la eseguirà alla cieca. L'hardware è un isomorfismo perfetto: legge numeri e li interpreta come azioni, rendendo la distinzione tra "Dato" e "Codice" una pura illusione logica.

Esperimento in QtSpim (La "Mutazione"): Il seguente programma nasconde l'istruzione di addizione (`0x01095020`) sotto forma di numero grezzo. Dopo averla eseguita la prima volta, la CPU rilegge quel numero, applica una banale addizione algebrica (+2) per cambiarne i bit, e sovrascrive se stessa. Il risultato matematico è che l'opcode si trasforma da `add` a `sub`.

```assembly
.data
# Teniamo in .data SOLO le stringhe innocue
msg_add: .asciiz "Esecuzione 1 (Addizione): "
msg_sub: .asciiz "\nEsecuzione 2 (Sottrazione): "

```

```
.text
.globl main

main:
    # 1. Prepariamo gli operandi nei registri temporanei
    li $t0, 10      # Primo numero: 10
    li $t1, 4       # Secondo numero: 4

    # 2. ESECUZIONE 1 (Addizione)
    # Chiamiamo la funzione nascosta. La CPU troverà il numero
    # 0x01095020 e lo interpreterà alla cieca come 'add $t2, $t0, $t1'
    jal codice_mutante

    # Stampiamo il risultato della prima esecuzione (14)
    li $v0, 4
    la $a0, msg_add
    syscall
    li $v0, 1
    move $a0, $t2
    syscall

    # ============================================================
    # 3. LA MUTAZIONE (Il cuore del Self-Modifying Code)
    # ============================================================
    # Calcoliamo l'indirizzo in RAM della nostra finta istruzione
    la $t3, codice_mutante
    # Leggiamo l'istruzione in $t4 come se fosse un normale numero intero
    lw $t4, 0($t3)

    # Modifica algebrica del codice:
    # Il campo 'funct' dell'istruzione ADD è 0x20.
    # Il campo 'funct' dell'istruzione SUB è 0x22.
    # Aggiungiamo 2 per trasformare 0x01095020 (add) in 0x01095022 (sub)
    addi $t4, $t4, 2

    # SOVRASCRIVIAMO L'ISTRUZIONE: il codice altera se stesso in RAM!
    sw $t4, 0($t3)
    # ============================================================

    # 4. ESECUZIONE 2 (Sottrazione)
    # Saltiamo di nuovo nello stesso identico indirizzo di prima,
    # ma ora l'hardware leggerà 0x01095022 e lo eseguirà come sottrazione!
    jal codice_mutante

    # Stampiamo il nuovo risultato (6)
    li $v0, 4
    la $a0, msg_sub
    syscall
    li $v0, 1
    move $a0, $t2
    syscall

    # 5. Uscita sicura
    li $v0, 10
    syscall

# --- LA ZONA MUTANTE ---
# Inseriamo i dati "crudi" direttamente all'interno della sezione .text
# per aggirare le protezioni di sicurezza (NX bit) della sezione dati.
codice_mutante:
    # 0x01095020 è l'esatta traduzione binaria di: add $t2, $t0, $t1
    .word 0x01095020

```

```
    # 0x03e00008 è l'esatta traduzione binaria di: jr $ra (ritorno al main)
    .word 0x03e00008
```

02. La Sfida di Flavio Giuseppe e la Magia dello Shift Binario

Testo: Lo storico Flavio Giuseppe si salvò da un assedio disponendosi nel punto esatto di un cerchio di $n = 41$ persone, in cui, a turno, veniva eliminata una persona sì e una no. In quale posizione si mise per essere l'ultimo superstite?

Suggerimento: Prima di fare calcoli complessi, prova a disegnare cosa succede se nel cerchio ci sono 2, 4, 8 o 16 persone. Trova la regola matematica generale e poi spiega il sorprendente legame con il sistema binario: come si ottiene la soluzione in un millisecondo facendo un semplice *Left Circular Shift* sul numero n (ovvero prendendo il suo bit più a sinistra e spostandolo in fondo a destra)?

Soluzione (Passo 1: L'intuizione matematica): Analizzando i numeri piccoli (es. $n = 8$), notiamo un pattern: dopo un giro completo, tutti quelli in posizione pari muoiono. Si ricomincia esattamente dalla persona numero 1, ma con la metà dei partecipanti ($n = 4$). Andando avanti, vincerà sempre la persona numero 1. Quindi: **se il numero di persone è una potenza esatta di 2 ($n = 2^m$), il sopravvissuto è sempre il numero 1.**

Passo 2: Il trucco per un numero qualsiasi: Cosa succede per $n = 41$? Qualsiasi numero può essere scritto come la somma della più grande potenza di 2 che lo precede, più un resto che chiameremo l. Nel nostro caso: $41 = 32 + 9$ (quindi $2^5 + 9$). Il trucco è semplice: dobbiamo far svolgere il gioco finché non eliminiamo esattamente 9 persone. A quel punto, rimarranno nel cerchio $41 - 9 = 32$ persone. E quando restano 32 persone, sappiamo che chi ha il turno in quel momento vincerà l'intera partita! Poiché a ogni eliminazione salta una persona (muoiono i pari: 2, 4, 6...), per far fuori 9 persone dovrà morire il soldato in posizione 18. Il prossimo a colpire, che dà il via al "cerchio perfetto" di 32 persone, è il **numero 19**. Flavio Giuseppe si era messo lì! La formula matematica generale per il vincitore è quindi: $J(n) = 2l + 1$.

Passo 3: Il Segreto Informatico (Lo Shift Circolare): È qui che la matematica si trasforma in magia binaria. Scriviamo la nostra formula $n = 2^m + l$ in Base 2. Il termine 2^m è semplicemente un **1** seguito da tanti zeri. Il resto l occuperà i bit a destra. Quindi, la codifica binaria di n ha questa forma:

$$n = \mathbf{1}\, b_{m-1}\, b_{m-2} \ldots b_1\, b_0$$

Dove tutta la stringa dopo l'1 iniziale ($b_{m-1} \ldots b_0$) rappresenta esattamente il nostro resto l.

Ora applichiamo la formula $2l + 1$ direttamente in binario:

- Moltiplicare l per 2 in binario significa aggiungere uno zero in fondo (questa operazione si chiama *Shift Logico a Sinistra*): $b_{m-1} \ldots b_0\, \mathbf{0}$.
- Aggiungere 1 significa semplicemente accendere quell'ultimo zero facendolo diventare un uno: $b_{m-1} \ldots b_0\, \mathbf{1}$.

Abbiamo appena dimostrato una proprietà bellissima: **calcolare il sopravvissuto equivale a staccare il bit '1' più a sinistra di n e incollarlo all'estrema destra!** In architettura dei calcolatori, questa mossa fulminea prende il nome di *Left Circular Shift*.

Verifichiamo al volo con $n = 41$:

1. 41_{10} in binario è $\mathbf{1}01001_2$
2. Stacco l'1 iniziale e lo sposto in coda: $01001\mathbf{1}_2$
3. Converto in decimale: $16 + 2 + 1 = \mathbf{19}$. La risposta è corretta!

03. Il Minimalismo Estremo (Turing-Completezza con 1 sola istruzione)

Testo: Abbiamo visto molte istruzioni MIPS, ma qual è il numero *minimo* di comandi necessari a un processore per eseguire qualsiasi algoritmo possibile?

Dimostra che ne basta **uno solo**: usando l'istruzione teorica `subleq A, B, C` (che significa "sottrai A da B e salva in B; se il risultato è ≤ 0 salta all'indirizzo C"), fai vedere in che modo potresti azzerare una variabile (cioè fare $X = 0$) e come potresti simulare un salto incondizionato (l'equivalente di `j`).

La Risposta: Ne basta UNA sola.

Il Concetto Matematico (OISC): L'architettura MIPS ha decine di istruzioni. Ma la teoria della calcolabilità dimostra che possiamo ridurre il set a un singolo elemento. Esistono architetture teoriche (chiamate OISC - One Instruction Set Computer) basate unicamente su **subleq** (Subtract and Branch if Less than or Equal to zero).

Sintassi: `subleq A, B, C` significa "Sottrai il valore all'indirizzo A dal valore all'indirizzo B, salva il risultato in B. Se il nuovo B è ≤ 0, salta all'indirizzo C, altrimenti vai alla riga successiva".

Dimostrazione Costruttiva (Perché una sola basta per fare tutto?): Un sistema è "Turing-completo" (può calcolare qualsiasi algoritmo esistente) se possiede tre capacità matematiche fondamentali: scrivere dati in memoria, fare aritmetica e fare salti condizionati (la base dei cicli `while` e degli `if`). Il salto condizionato è già integrato nativamente. Vediamo come costruire il resto dal nulla:

- **1. Svuotare una variabile ($X = 0$):**
  Basta sottrarre una variabile a sé stessa!
  `subleq X, X, next` $\implies X \leftarrow X - X = 0$.

- **2. Salto Incondizionato (JUMP a C):**
  Sfruttiamo l'azzeramento. Se svuotiamo una variabile fittizia Z e la sottraiamo a sé stessa, il risultato è esattamente 0. Poiché il comando salta se il risultato è ≤ 0, il salto a C scatterà *sempre*.

- **3. Addizione senza addizione ($Z = X + Y$):**
  Poiché $X + Y = X - (-Y)$, usiamo una variabile temporanea T.

  1. `subleq T, T, next` (Azzero T)
  2. `subleq X, T, next` (T diventa $0 - X = -X$)
  3. `subleq T, Z, next` (Z diventa $Z - (-X) = Z + X$)

- **4. Copiare un dato, ovvero l'Assegnamento ($Y = X$):**
  Cosa succede se vogliamo assegnare a Y il valore di X, distruggendo il vecchio valore di Y?

  1. `subleq Y, Y, next` (Azzero Y)
  2. `subleq T, T, next` (Azzero T)
  3. `subleq X, T, next` (T diventa $-X$)
  4. `subleq T, Y, next` (Y diventa $0 - (-X) = X$)

Conclusione della Dimostrazione: Avendo dimostrato costruttivamente che con `subleq` possiamo azzerare variabili, copiare dati, sommare, sottrarre e creare salti incondizionati o legati a una condizione, abbiamo ottenuto l'esatta capacità espressiva di un linguaggio come il C o Python. Qualsiasi altra cosa (moltiplicazioni, array, funzioni) è matematicamente solo una combinazione di questi mattoncini di base!

04. La Sfida: Piegare il 2D in una RAM 1D

Testo: La memoria RAM di un computer è come un lunghissimo nastro monodimensionale (1D) di cellette numerate in fila. Ma in matematica e nei videogiochi, lavoriamo spesso con griglie o immagini bidimensionali (2D). Se volessimo salvare un'immagine nella RAM, avremmo bisogno di una funzione matematica che trasformi ogni coppia di coordinate (X, Y) in un singolo indirizzo Z (una biiezione da $\mathbb{N}^2$ a $\mathbb{N}$).

La sfida è questa: come possiamo mappare questa griglia in modo "intelligente"? Se usassimo un metodo banale (come contare riga per riga), due pixel vicinissimi nell'immagine 2D potrebbero finire salvati in indirizzi di memoria 1D lontanissimi tra loro, rallentando terribilmente il computer. Riesci a immaginare un modo per assegnare un numero Z a ogni cella in modo da preservare il più possibile la vicinanza spaziale?

Soluzione (L'Intreccio dei Bit): In matematica esistono diverse soluzioni eleganti a questo problema, come le famose curve di Peano o di Hilbert, che riempiono il piano mantenendo i punti vicini tra loro. In informatica, una delle soluzioni più brillanti ed efficienti è il **Codice di Morton** (o curva *Z-order*). La logica alla base è di una semplicità disarmante, eppure genera una bellissima struttura frattale.

Date le coordinate di un pixel (X, Y), le convertiamo in binario. Per trovare il suo indirizzo univoco nella RAM monodimensionale (Z), ci basta **intrecciare i bit di X e Y** esattamente come i denti di una cerniera lampo!

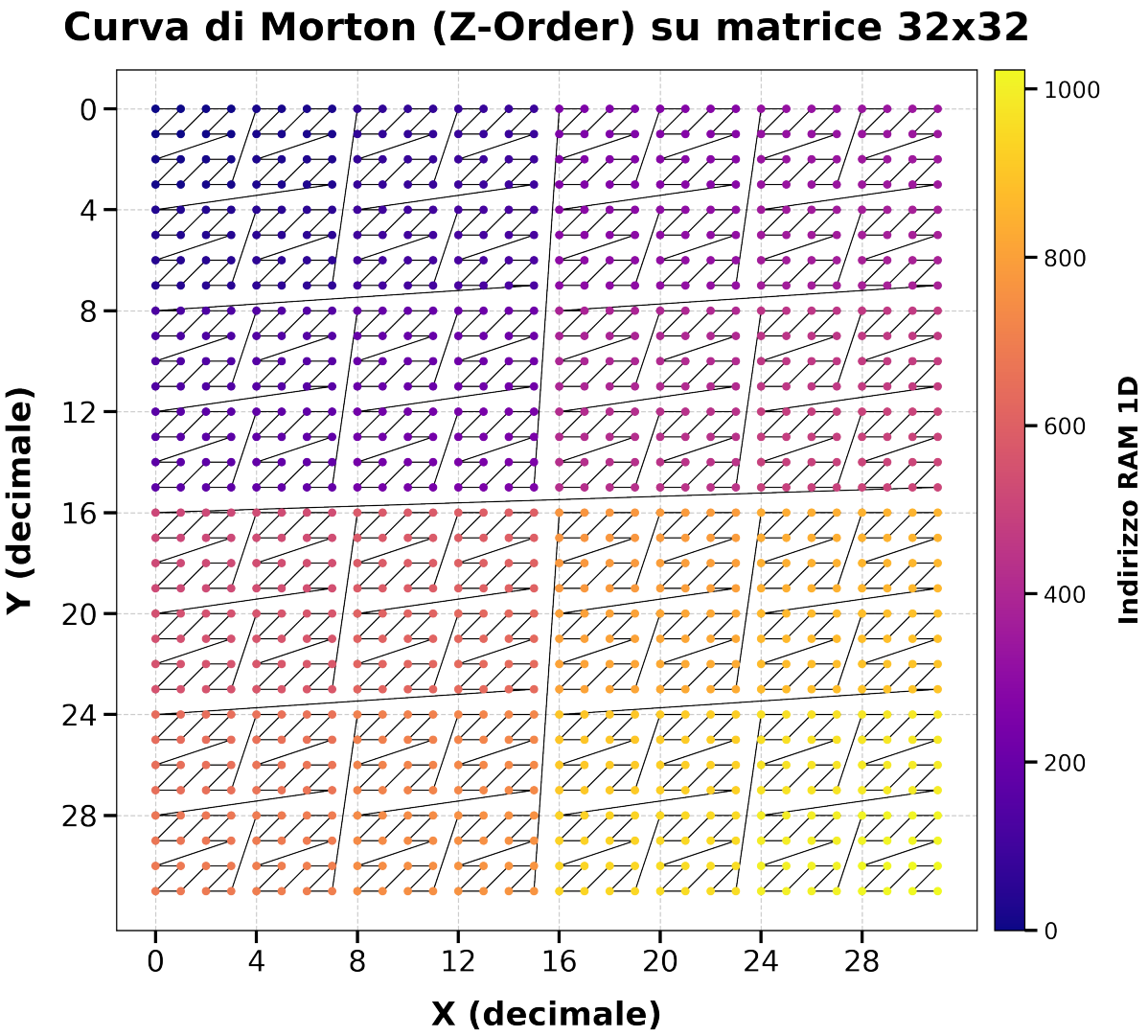
Esempio Pratico: Vogliamo trovare l'indirizzo Z del pixel alle coordinate $X = 3$ e $Y = 5$.

1. Scriviamo in binario: $X = \mathbf{011}_2$ e $Y = \mathit{101}_2$.
2. Intrecciamo i bit alternandoli (prima un bit di Y, poi uno di X, e così via):

$$Z = \mathit{1}\mathbf{0}\mathit{0}\mathbf{1}\mathit{1}\mathbf{1}_2$$

3. Convertiamo di nuovo in decimale: $100111_2 = \mathbf{39}$.

Il pixel di coordinate $(3, 5)$ verrà salvato all'indirizzo di memoria 39. Se disegniamo il percorso che unisce gli indirizzi da 0 in poi sulla nostra griglia 2D, non otteniamo delle noiose righe dritte, ma una curva a forma di "Z" che si ripete all'infinito (una **Curva Frattale di Lebesgue**). Questa magia permette al computer di caricare blocchi di memoria vicini in modo incredibilmente veloce.


Curva di Morton (Z-Order) su matrice 32x32
Y (decimale)
0
4
8
12
16
20
24
28
X (decimale)
0
4
8
12
16
20
24
28
Indirizzo RAM 1D
0
200
400
600
800
1000