Prova Finale di Reti Logiche A.A. 2023-2024

Prof. William Fornaciari

Leonardo Chiaretti 10720295
James Enrico Busato 10771958

Corso di Laurea Triennale in Ingegneria Informatica

Milano Leonardo

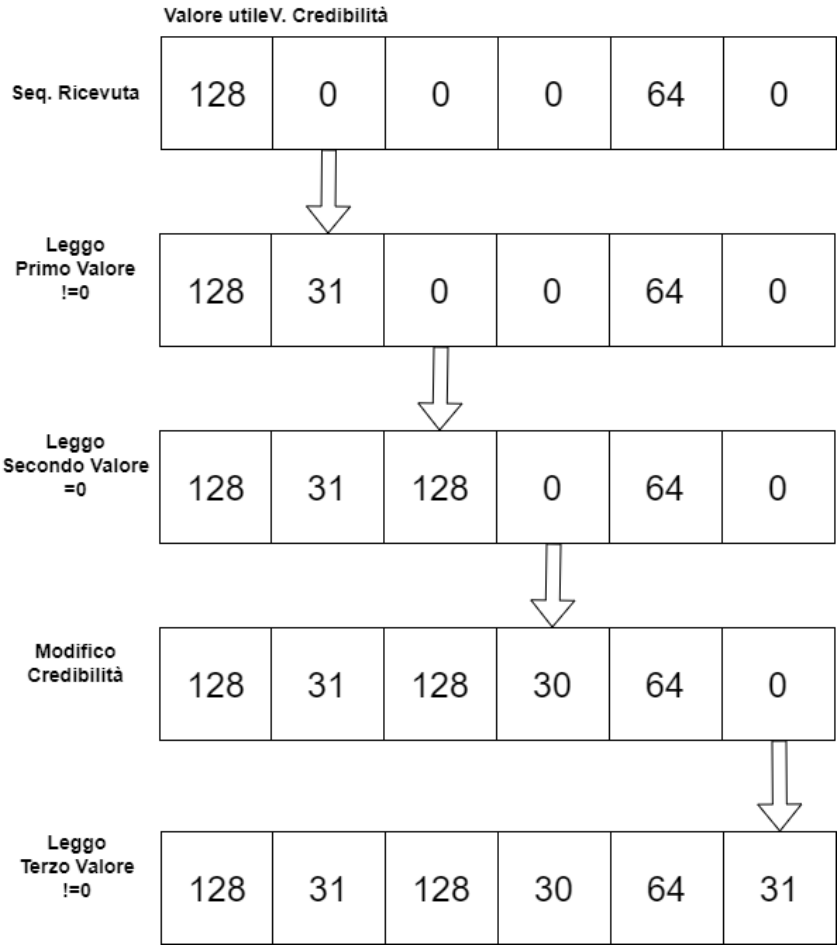
Introduzione

Lo scopo del progetto è di realizzare un componente che effettua la lettura di una **sequenza** presente nella RAM e ne modifica alcuni valori secondo la specifica fornita.

Più in particolare, vengono forniti il numero di parole **K** e l'indirizzo della memoria (**ADD**) da cui inizia la sequenza da leggere; all'interno della sequenza, una parola può essere considerata come l'unione, in due indirizzi di memoria separati ma contigui, di un **valore utile** e di un **valore di credibilità**.

Il valore utile può consistere di un numero diverso o uguale a 0; nel caso in cui sia uguale a 0 viene sovrascritto dall'**ultimo valore utile letto**.
Il valore di credibilità consiste invece inizialmente di soli 0, e la specifica si occupa di modificarlo in base al suo valore utile associato.
Se infatti, la parola ha un valore utile diverso da 0, il valore di credibilità sarà sempre uguale a **31**; se invece, è presente uno o **N** valori utili pari a zero, sequenzialmente, il valore di credibilità è pari alla formula **31-N** dove N rappresenta il numero di zeri incontrati, e viene aggiornato per ogni valore utile=0 fino a fermarsi a 31.

Ad esempio, per una sequenza di lunghezza K=3, si ha la seguente lettura e riscrittura:

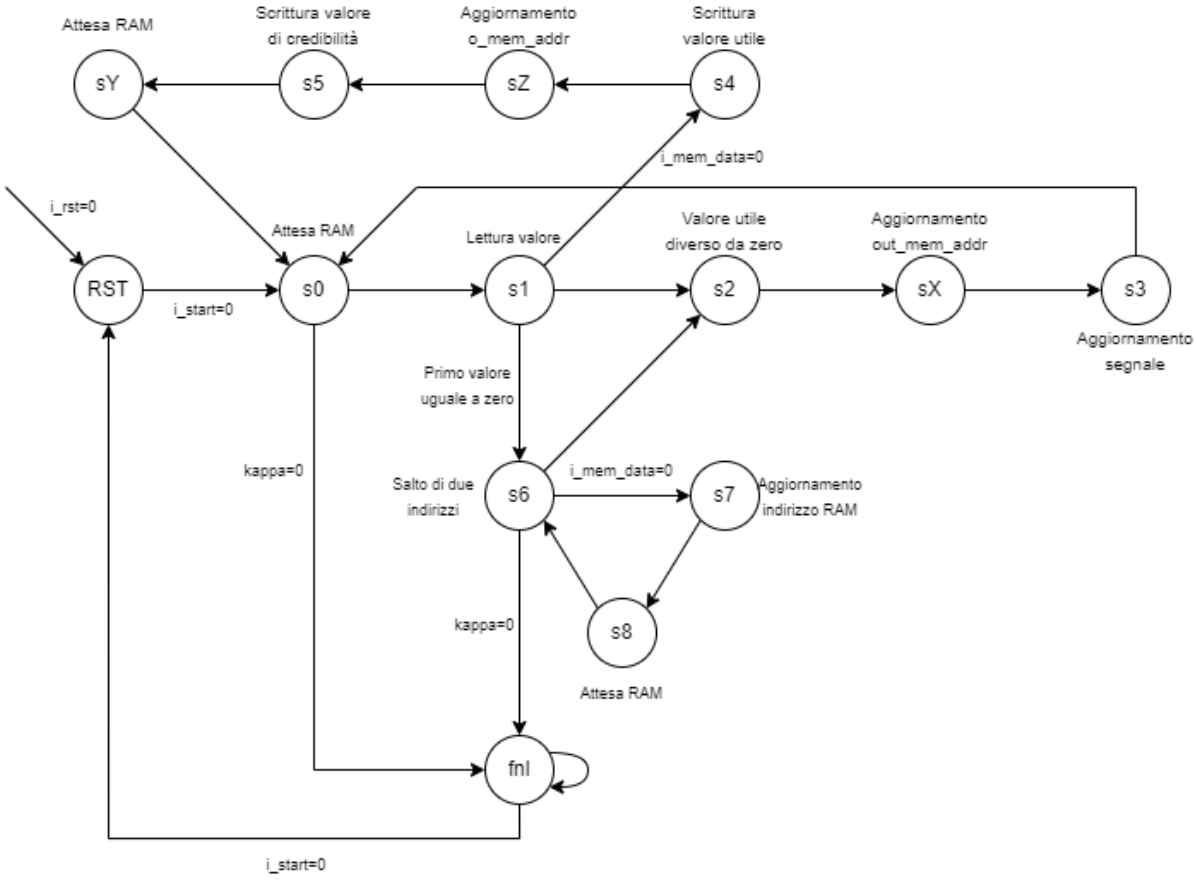


Architettura

Per rendere l'entità il più compatta possibile e non dovere mappare multipli componenti all'interno dello stesso file, abbiamo optato per una **FSM**, particolarmente adatta anche per la semplicità di confronto e scrittura dei valori che fornisce.

Tale macchina è composta da **14 stati**, un numero dettato principalmente dalla "lentezza" con cui VHDL permette di leggere i valori provenienti dalla RAM, operazione che necessita di due stati, e dalla necessità di aggiornare correttamente i valori utili e di credibilità.

Eccone la composizione:



Più in dettaglio, la macchina è divisa in **cinque zone** principali di funzionamento:

- **La zona iniziale**, comprendente gli stati di RST, s0, s1: Tale zona è la prima visitata all'avvio del codice e si occupa di inizializzare il codice e operare la scelta di indirizzamento del flusso nelle tre aree.
    a. Lo stato **RST** si occupa di settare correttamente i valori in uscita (**o_mem_en, o_mem_we, o_done**) e di attendere l'arrivo del segnale di **i_start.**
    All'arrivo di tale segnale, si occupa di inizializzare correttamente i nostri signals con gli ingressi ricevuti dal testbench, quali **i_add** e **i_k.**

b. Lo stato **s0** attende dalla RAM preparata nello stato precedente l'arrivo del valore da leggere; a questo stato ritornano anche i primi due flussi (valore utile diverso da zero o uguale a zero ma non primo valore letto).
c. Lo stato **s1** è infine il nodo centrale che si occupa di leggere il valore utile e di scegliere a quale delle tre zone indirizzarsi.
Conserva su un segnale apposito chiamato **valore_utile,** l'ultimo valore utile diverso da zero incontrato, per poterlo scrivere nell'evenienza siano incontrati successivamente valori utili uguali a zero.

Questi tre stati sono quindi i più frequentati e consentono sia di eseguire due specifiche sequenzialmente (cioè ricevere due segnali di **i_start**), sia di ricominciare a lettura in corso in seguito alla ricezione di un improvviso **i_rst**.

- La zona in cui si è letto un **valore utile diverso da zero**, comprendente gli stati s2, sX, s3**:** tale zona risulta la più semplice in quanto la lettura di un valore utile diverso da zero richiede solamente di spostarsi di un indirizzo di memoria e di scrivere il valore 31.
    a. Lo stato **s2** quindi si occupa dell'effettiva scrittura del dato sul segnale **o_mem_data** che comunica direttamente con la RAM.
    b. Lo stato **sX** aggiorna il segnale di **out_mem_add**r.
    c. Lo stato **s3** aggiorna effettivamente il segnale **o_mem_addr** e si reindirizza nel normale flusso tornando allo stato s0.
- La zona in cui si è letto un **valore utile uguale a zero ma ho già letto dei valori utili diversi da zero**, comprendente gli stati s4, sZ, s5, sY: in questa zona abbiamo la presenza di uno stato in più dettata dalla necessità di dovere scrivere anche sull'indirizzo contente il valore utile.
    a. Lo stato **s4** si occupa pertanto di scrivere effettivamente l'ultimo valore utile salvato presente su **valore_utile**, aggiorna il segnale **out_mem_addr** e lo passa a sZ.
    b. Lo stato **sZ** si occupa di scriverlo effettivamente su **o_mem_addr.**
    c. Lo stato **s5** si occupa quindi di calcolare il valore da scrivere, tramite un signal chiamato **decounter**, che viene aggiornato (+1) per ogni valore utile uguale a zero e resettato se viene letto un valore utile diverso.
    d. Lo stato **sY** si occupa di ulteriore attesa per la lettura RAM, e riporta il flusso alla zona iniziale.
- La zona in cui si è letto un **valore utile uguale a zero ed è il primo** della sequenza fornita, comprendente gli stai s6, s7, s8: questa zona nasce dal fatto che, qualora il primo valore utile letto risulti zero, si ha la possibilità di saltare direttamente al prossimo valore utile, saltando lettura e scrittura del valore di credibilità (che equivale ad un salto di due indirizzi di memoria).
    a. Lo stato **s6** si occupa, similmente allo stato s1, di leggere il valore ed eventualmente di reindirizzarsi al flusso classico, o di continuare a saltare due indirizzi.
    b. Lo stato **s7** si occupa di aggiornare il segnale **o_mem_addr**.
    c. Lo stato **s8** si occupa di attendere la ricezione del dato.

- La zona finale, comprendente solo lo stato **fnl**, si occupa di controllare il corretto funzionamento dei segnali **o_done**, anche in risposta al segnale di **i_start**, ed eventualmente si riporta allo stato di RST per iniziare una nuova computazione.

Inoltre, ai già descritti segnali presenti nella specifica, abbiamo aggiunto per necessità progettuali i seguenti segnali:

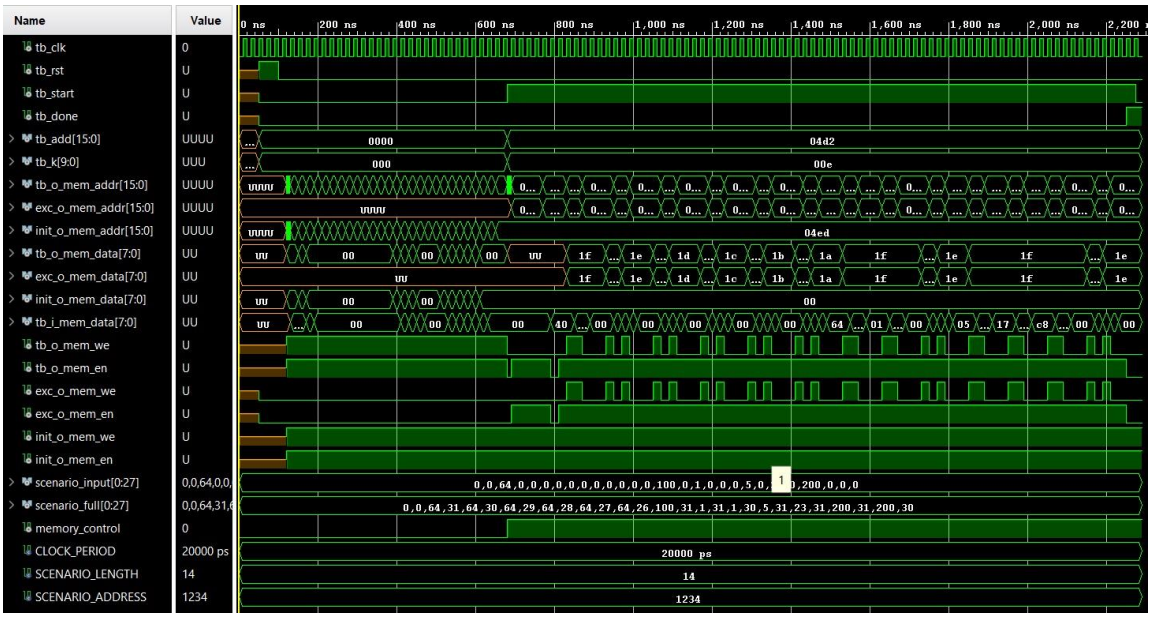
1. signal **decounter**: integer range 0 to 31;
   -- per la gestione della scrittura nei valori di credibilità preceduti da valori utili uguali a zero, viene settato a zero e incrementato secondo specifica; eventualmente viene reinizializzato a zero quando si incontra un valore proveniente da i_mem_data diverso da zero.
2. signal **kappa**: integer range 0 to 511;
   -- per il conteggio delle parole da leggere; il segnale viene direttamente copiato da i_k e decrementato ogni due indirizzi di memoria, ovvero una parola.
3. signal **out_mem_addr**: std_logic_vector(15 downto 0) := (others => '0');
   -- per una gestione ottimale della memoria e l'impossibilità di riscrivere direttamente i segnali di output; all'inizio riceve (come per o_mem_addr) il valore di i_add e viene poi incrementato di 1 o di 2 a seconda delle casistiche.
4. signal **valore_utile**: std_logic_vector(7 downto 0) := (others => '0');
   -- per salvare l'ultimo valore utile incontrato ed eventualmente riscriverlo.

Risultati sperimentali

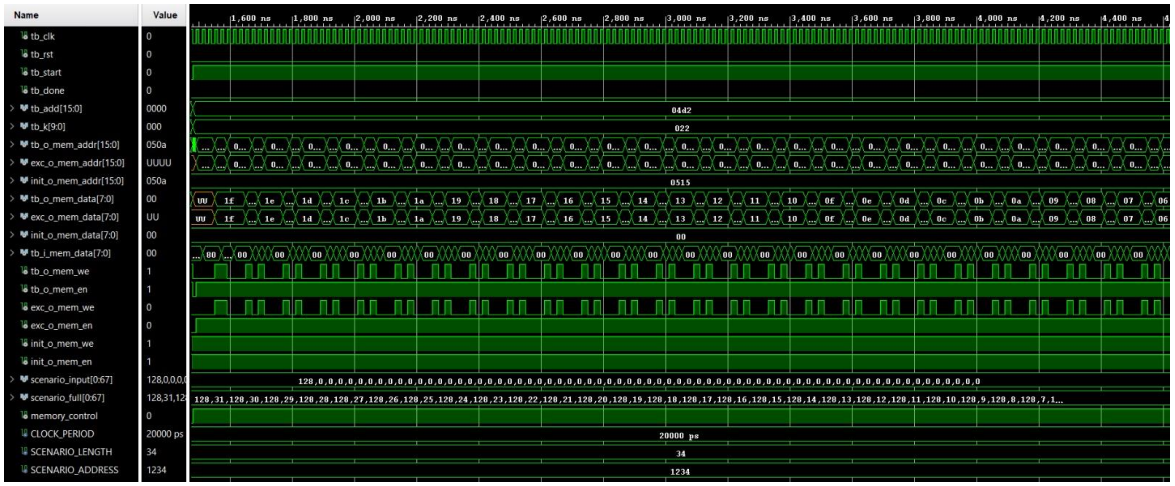
Testbench

Al fine di verificare innanzitutto che la macchina funzioni in tutti e tre le zone distinte sopra, oltre a quello fornito, abbiamo provato due testbench che:

1. Prevedano **l'inizio della sequenza con degli zeri**, nel nostro caso uno solo, come si evince da scenario_input.

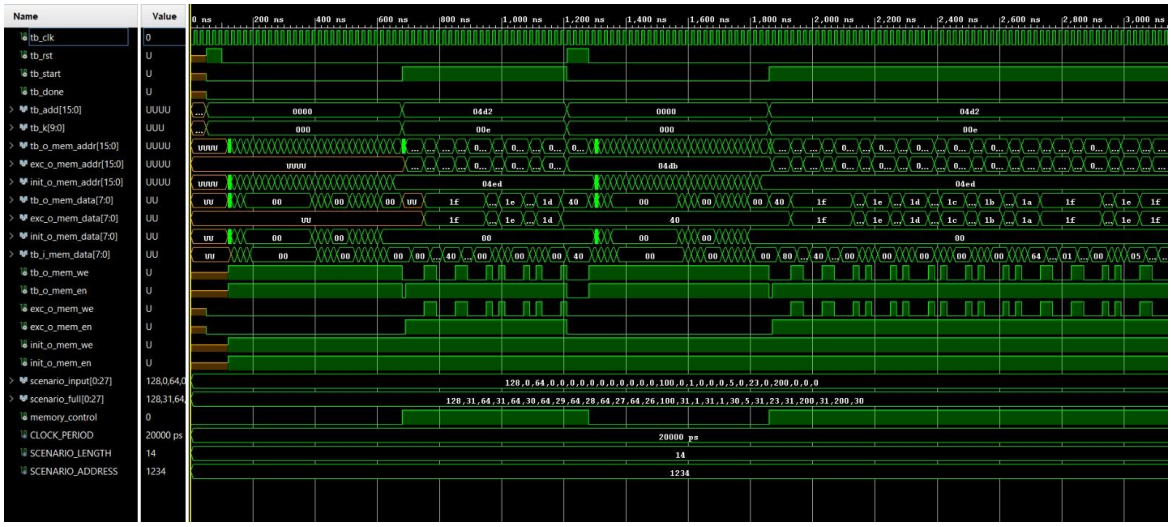


2. Verifichino che il decounter venga **correttamente fermato a 31** e non scriva valori negativi, ed in questo modo abbiamo verificato anche una sequenza numericamente più corposa (i_k=34).

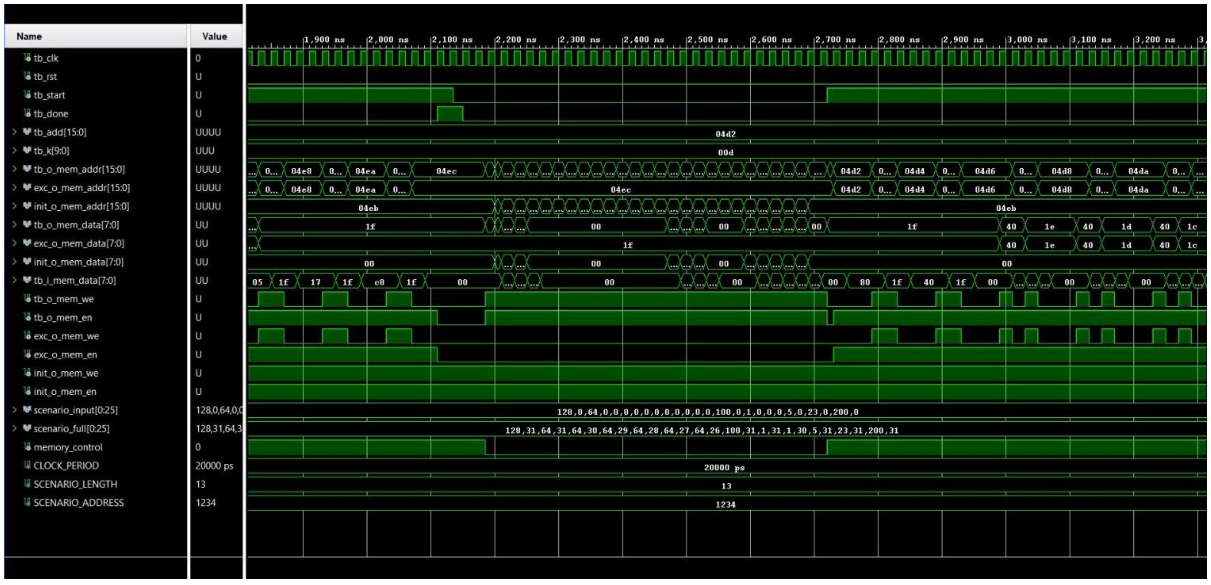


Dopodiché abbiamo verificato che:

1. La macchina risponda correttamente alla ricezione di un **segnale i_rst casuale** riportandosi subito allo stato di RST



2. La macchina sia in grado di **compiere una successiva elaborazione** dopo che il segnale i_start sia nuovamente portato a 1



Report di sintesi

Dal **report_utilization** si evince la **non presenza**, voluta, **di Latch**, che ci conferma quindi una corretta scrittura degli stati della FSM, ed ovviamente un uso modestissimo (**0,03%** riguardo ai registri **Flip Flop** e **0,06%** alle **LookUp Tables**) della componentistica effettivamente fornita dalla FPGA.

```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |   82 |     0 |    134600 |  0.06 |
|   LUT as Logic          |   82 |     0 |    134600 |  0.06 |
|   LUT as Memory         |    0 |     0 |     46200 |  0.00 |
| Slice Registers         |   69 |     0 |    269200 |  0.03 |
|   Register as Flip Flop |   69 |     0 |    269200 |  0.03 |
|   Register as Latch     |    0 |     0 |    269200 |  0.00 |
| F7 Muxes                |    0 |     0 |     67300 |  0.00 |
| F8 Muxes                |    0 |     0 |     33650 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Dal **report_timing_summary** invece, si nota come la FSM sia abbastanza veloce, impiegando circa **4,251 ns** per completare il testbench base, rendendola quindi anche utilizzabile con clock con frequenza più alta.

```
------------------------------------------------------------------------------------
Slack (MET) :             15.749ns  (required time - arrival time)
  Source:                 kappa_reg[4]/C
                            (rising edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            out_mem_addr_reg[0]/CE
                            (rising edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            20.000ns  (clock rise@20.000ns - clock rise@0.000ns)
  Data Path Delay:        3.869ns  (logic 1.145ns (29.594%)  route 2.724ns (70.406%))
```

Conclusioni

La specifica richiesta concentrava la propria attenzione principalmente sull'interazione con la memoria **RAM** e sulla gestione delle **operazioni di lettura e scrittura** tramite i segnali forniti.

L'utilizzo di una FSM ha permesso di compartimentare qualsiasi operazione e renderla atomica, in modo da potere gestire correttamente lo sfasamento fra la richiesta di lettura/scrittura e l'effettiva operazione in termini di clock, oltre che la gestione dei vari segnali ricevuti da input.

L'FSM, inoltre, risulta molto robusta rispetto a tutti i comportamenti sottoposti tramite testbench e può sicuramente gestire volumi di dati più alti di quelli ipotizzati nei nostri casi.