Relazione Progetto Reti Logiche

Alberto Francesco Visconti

a.a 2023/24
CodicePersona: 10763965
Docente: Gianluca Palermo

POLITECNICO
MILANO 1863

Contents

1 Introduzione

1.1 Specifica

1.1.1 Descrizione

Il progetto consiste nella descrizione di un modulo hardware, utilizzando come linguaggio VHDL, che si interfacci con una memoria RAM.
Al modulo viene indicato un indirizzo iniziale di memoria e una lunghezza di parole da leggere; il modulo quindi deve procedere alla lettura, elaborazione e riscrittura dei dati a partire dall'indirizzo di memoria indicato fino a quando viene raggiunto il numero di parole o viene alzato il segnale di **i_rst**. In entrambi i casi il modulo deve essere pronto per un' altra possibile sequenza di dati da leggere.

1.1.2 Funzionamento

Il modulo viene inizializzato ponendo il segnale di **o_done** e gli altri segnali di output a '0', la situazione rimane invariata fino a quando il segnale di **i_start** viene posto a '1'; a questo punto vengono memorizzati i valori di **i_add** e **i_k** rispettivamente come l'indirizzo di memoria da cui iniziare a leggere e il numero di parole che devono essere lette.
Dopo la memorizzazione dei parametri il circuito svolge in ordine queste operazioni: legge la parola W all'indirizzo attuale e la sostituisce con una nuova se necessario, scrive nel byte subito dopo la credibilità associata con la parola, incrementa il valore delle parole già lette e controlla se si è raggiunto il limite: in caso negativo ricomincia la sequenza altrimenti alza il segnale di **o_done** e aspetta fino a quando **i_start** venga abbassato.
Infine sia che venga alzato **o_done** o avvenga un reset asincrono con **i_rst** i valori di output e i segnali interni riassumono i valori precedenti alla salita di **i_start**.
Per una spiegazione più dettagliata si faccia riferimento al paragrafo 2.2.5

1.1.3 Comportamento

Le regole per l'assegnamento di un valore alla parola e alla credibilità sono quindi le successive: se la parola letta è composta da soli zero in memoria bisogna scrivere il valore dell'ultima parola letta diversa da 0 e nel byte successivo il valore della credibilità, che consiste nel valore della credibilità precedente-1 fino a un minimo di zero, in caso di parola diversa da 0 invece andrà riscritta in uscita la parola appena letta con credibilità associata pari a 31.
Un caso particolare avviene quando le prime parole lette sono 0, in quel caso sia la parola che la credibilità saranno 0 fino alla lettura della prima parola con valore diverso. Allegato si trova un esempio di una possibile sequenza:

- Sequenza di partenza (in grassetto i valori W):

128 0 **64** 0 **0** 0 0 **0** 0 0 0 **0** 0 **100** 0 **1** 0 **0** 0 **5** 0 **23** 0 **200** 0 **0** 0

- Sequenza finale

128 31 **64** 31 **64** 30 **64** 29 **64** 28 **64** 27 **64** 26 **100** 31 **1** 31 **1** 30 **5** 31 **23** 31 **200** 31 **200** 30

1.2 Strumenti utilizzati

La versione di Vivado utilizzata è stata la 2018.3.1; vengono inoltre aggiunte le interfacce della memoria RAM e del componente con una breve descrizione dei segnali.

```
-- Single-Port Block RAM Write-First Mode (recommended template)
--
-- File: rams_02.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity rams_sp_wf is
port(
  clk  : in  std_logic;
  we   : in  std_logic;
  en   : in  std_logic;
  addr : in  std_logic_vector(15 downto 0);
  di   : in  std_logic_vector(7 downto 0);
  do   : out std_logic_vector(7 downto 0)
);
end rams_sp_wf;

architecture syn of rams_sp_wf is
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM : ram_type;
begin
  process(clk)
    begin
    if clk'event and clk = '1' then
      if en = '1' then
        if we = '1' then
          RAM(conv_integer(addr)) <= di;
          do                      <= di after 2 ns;
        else
          do <= RAM(conv_integer(addr)) after 2 ns;
        end if;
      end if;
    end if;
  end process;
end syn;
```

Figure 1: interfaccia della memoria fornita

```
entity project_reti_logiche is
    port (
        i_clk   : in std_logic;
        i_rst   : in std_logic;
        i_start : in std_logic;
        i_add   : in std_logic_vector(15 downto 0);
        i_k     : in std_logic_vector(9 downto 0);

        o_done  : out std_logic;

        o_mem_addr : out std_logic_vector(15 downto 0);
        i_mem_data : in std_logic_vector(7 downto 0);
        o_mem_data : out std_logic_vector(7 downto 0);
        o_mem_we   : out std_logic;
        o_mem_en   : out std_logic
    );
end project_reti_logiche;
```

Figure 2: interfaccia del componente fornito

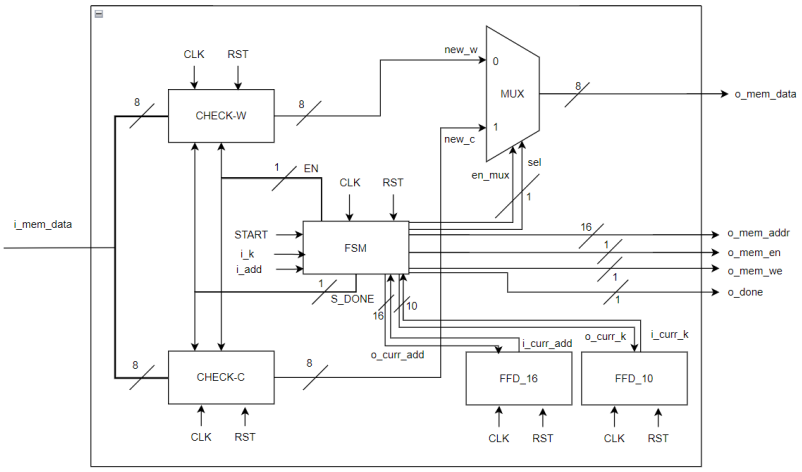
- i_clk è il segnale di CLOCK in ingresso generato dal Test Bench;
- i_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- i_start è il segnale di START generato dal Test Bench;
- i_k è il segnale (vettore) K generato dal Test Bench rappresentante la lunghezza della sequenza;
- i_add è il segnale (vettore) ADD generato dal Test Bench che rappresenta l'indirizzo dal quale parte la sequenza da elaborare;
- o_done è il segnale DONE di uscita che comunica la fine dell'elaborazione;
- o_mem_addr è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- i_mem_data è il segnale (vettore) che arriva dalla memoria e contiene il dato in seguito ad una richiesta di lettura;
- o_mem_data è il segnale (vettore) che va verso la memoria e contiene il dato che verrà successivamente scritto;
- o_mem_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- o_mem_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0.

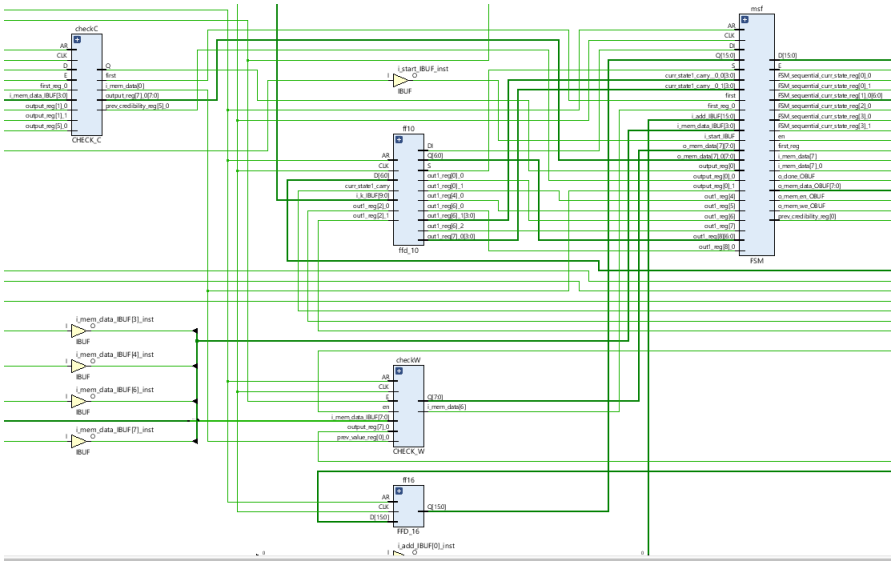
2 Architettura

2.1 Descrizione di altro livello

Leggendo la specifica si può notare che il modulo si può simplicisticamente dividere in tre componenti: la prima dedicata alla sostituzione della parola W con l'ultimo valore valido, la seconda dedicata al calcolo della credibilità C della parola e una terza parte più raffinata, che infatti si è deciso di realizzare tramite Macchia a Stati Finiti (FSM), che si occupa di tenere il conto del numero di iterazioni effettuate e della gestione di lettura e scrittura in memoria; inoltre, poiché il numero di iterazioni e l'indirizzo attuale di memoria non vengono

aggiornati a ogni ciclo di **clock**, sono stati creati 2 componenti di supporto con comportamento uguale a quello di FLIP-FLOP di tipo D per evitare la creazione di **latch**. Sotto viene mostrato il confronto tra il disegno iniziale realizzato dallo studente e una parte della schematica post-sintesi realizzata da Vivado, si noti che a parte per il Multiplexer che viene espanso la corrispondenza è quasi uno a uno.





2.2 Descrizione componenti

2.2.1 CHECK-W

Questo sottomodulo si occupa di leggere **i_mem_data** e analizzarne il contenuto, se il valore è 0 allora viene posto in uscita il valore della parola letta precedentemente (questo valore viene inizializzato a zero quindi se la prima parola letta è 0 in output si avrà ancora 0). Viene controllato tramite un segnale di **enable** dalla FSM per garantire che la lettura avvenga al momento opportuno, ovvero dopo aver caricato i dati corretti dalla memoria.

2.2.2 CHECK-C

Questo sottomodulo si occupa di leggere **i_mem_data** e analizzarne il contenuto assegnando poi una credibilità adeguata. Il valore della credibilità dipende dalla parola: se questa è diversa da 0 allora ritornerà il valore 31 altrimenti ritornerà il valore della credibilità precedente -1 fino a un minimo di 0, possiede inoltre un flag in modo che se le prime parole lette siano 0 allora ritornerà quel valore. Questo componente viene controllato tramite un segnale di **enable** dalla FSM per garantire che la lettura avvenga al momento opportuno, ovvero dopo aver caricato i dati corretti dalla memoria, ed evitare più elaborazioni durante la lettura della stessa parola, che abbasserebbero eccessivamente il valore di output nel caso questa sia 0.

2.2.3 MULTIPLEXER

Il multiplexer è un componente asincrono che riceve in input 2 valori, che sono rispettivamente l'output fornito da CHECK-W e quello fornito da CHECK-C, oltre a 2 segnali di **enable** e **select(sel)** dalla FSM. Quando **enable**=1 in outupt ripete il valore della porta 0 se **sel** vale 0, il valore della porta 1 altrimenti.

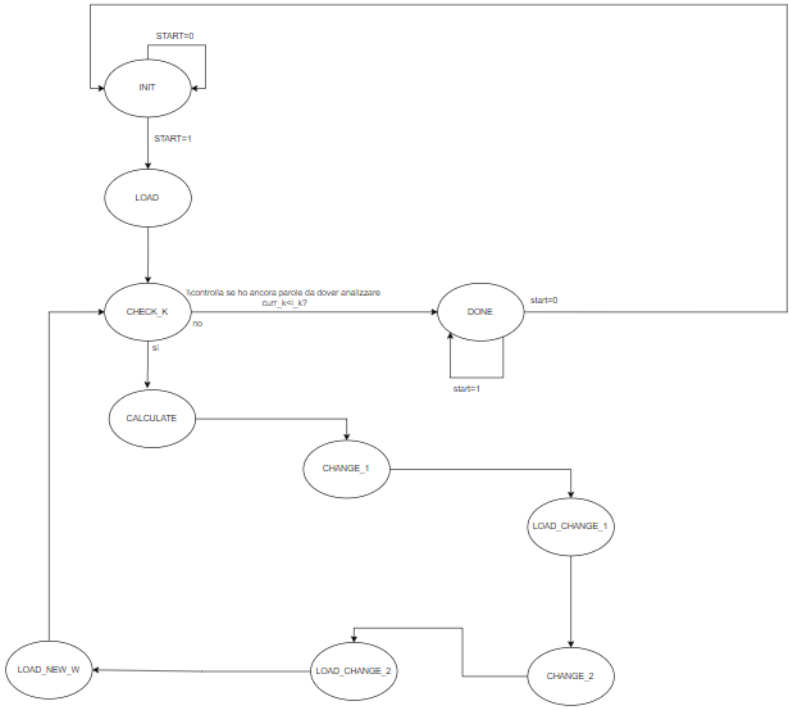
2.2.4 FFD_10 & FFD_16

Questi sottomoduli hanno una funzionalità quasi identica, cambia solo la dimensione del dato salvato, quindi la loro descrizione sarà accorpata. Come si potrebbe intuire dal nome entrambi hanno le funzionalità di un Flip-Flop di tipo D tuttavia il numero a destra, che indica la lunghezza del dato salvato, cambia, questo perché FFD_16 viene utilizzato per salvare l'indirizzo di memoria corrente (composto da 16 bit) mentre FFD_10 il numero di cicli della FSM completati (che non potrà mai superare i 10 bit in quanto sarà sempre minore o uguale a **i_k**).Entrambi ricevono in ingresso un valore che, al fronte di salita di **clock** successivo, riportano in uscita.

2.2.5 FSM

La FSM si comporta come il "cervello" di questo circuito infatti si occupa degli aspetti più delicati (come la gestione della memoria) e del coordinamento degli

altri componenti usando i segnali di **enable**. Sotto si può trovare un diagramma con gli stati della FSM uniti a una loro breve descrizione.



- INIT: lo stato iniziale, tutti i segnali sono posti ai valori di reset e si attende che start venga alzato.
- LOAD: vengono caricati i valori di **i_k** e **i_add**.
- CHECK_K: viene valutata la condizione per cui il numero di parole lette sia minore di **i_k**, è stato posizionato qui e non alla fine per gestire anche l'evenienza di k=0.
- CALCULATE: tramite il segnale di **en** vengono accesi, e poi allo stato successivo spenti, le componenti CHECK-C e CHECK-W per il calcolo della parola e della sua credibilità.
- CHANGE_1: viene caricato il valore calcolato da CHECK-W in **o_mem_data**.
- LOAD_CHANGE_1: vengono alzati i segnali di **o_mem_en** e **o_mem_we** per consentire la scrittura della memoria.
- CHANGE_2:viene caricato il valore calcolato da CHECK-W in **o_mem_data** e aumentato di 1 il segnale di **o_curr_add**.

- LOAD_CHANGE_2:vengono alzati i segnali di **o_mem_en** e **o_mem_we** per consentire la scrittura della memoria.
- LOAD_NEW_WORD: viene aumentato di 1 il segnale di **o_curr_add** e abbassato **o_mem_we** in modo da avere il dato disponibile in **i_mem_data** per il prossimo CALCULATE, se necessario.
- DONE: lo stato finale, viene alzato il segnale di **o_done** e si aspetta ad abbassarlo fino a quando **i_start** non venga posto a 0.

3 Risultati sperimentali

3.1 Report sintesi

3.1.1 Risorse utilizzate

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	83	0	134600	0.06
LUT as Logic	83	0	134600	0.06
LUT as Memory	0	0	46200	0.00
Slice Registers	63	0	269200	0.02
Register as Flip Flop	63	0	269200	0.02
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

3.1.2 Timing report

```
Timing Report

Slack (MET) :             16.127ns  (required time - arrival time)
```

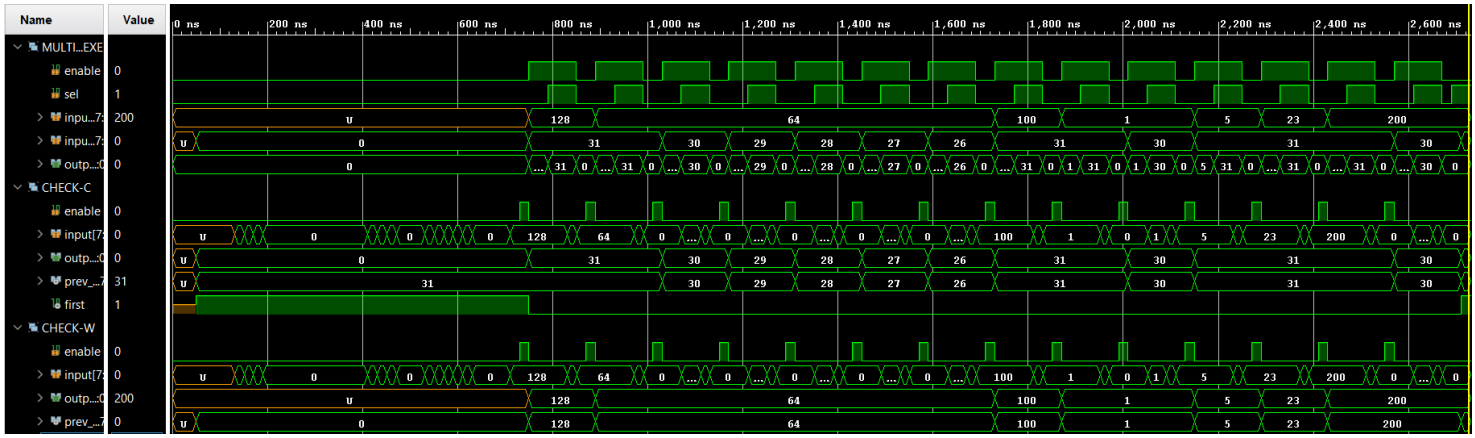
3.1.3 Netlist report

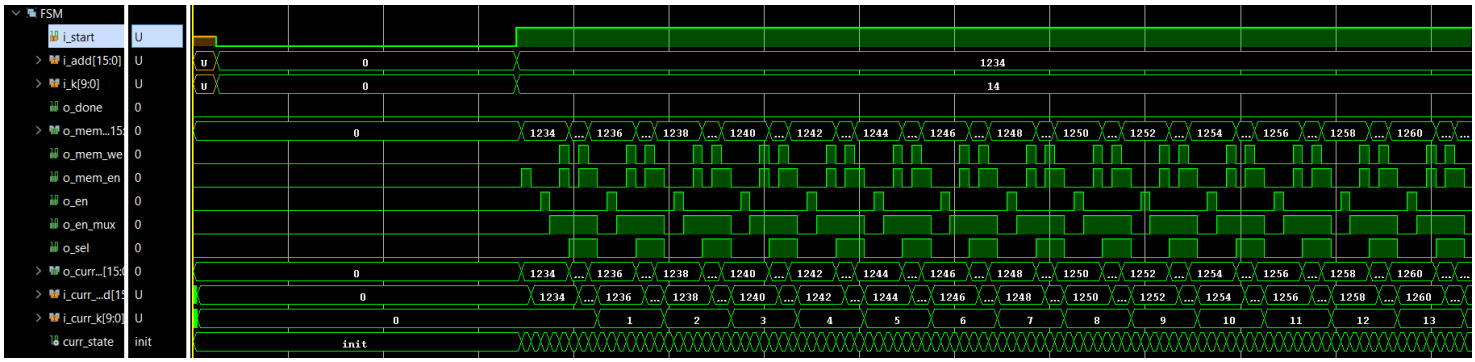
```
Report Check Netlist:
+------+------------------+-------+---------+-------+------------------+
|      |Item              |Errors |Warnings |Status |Description       |
+------+------------------+-------+---------+-------+------------------+
|1     |multi_driven_nets |      0|        0|Passed |Multi driven nets |
+------+------------------+-------+---------+-------+------------------+
--------------------------------------------------------------------------------
```

3.2 Risultati simulazioni

3.2.1 Test Bench 0

Si tratta del Test Bench fornito dai docenti, non contiene nessuna condizione limite quindi è un ottimo primo test per valutare un funzionamento generale del modulo. Vengono inseriti il comportamento della FSM e del Multiplexer che poi saranno omessi in quanto non dipendenti dal tipo di dato letto mentre alcuni segnali o costanti verranno comunque omessi perché di scarsa significatività.
Il componente ha passato questo test anche in post sintesi.

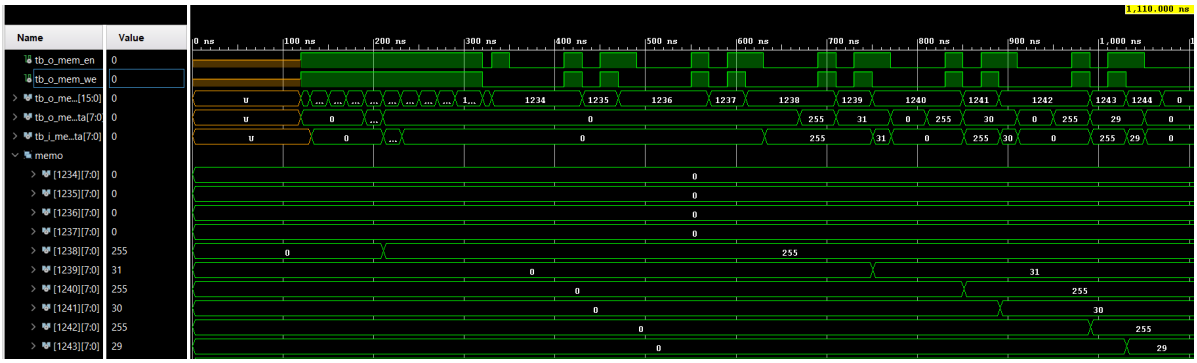




3.2.2 Test Bench 1

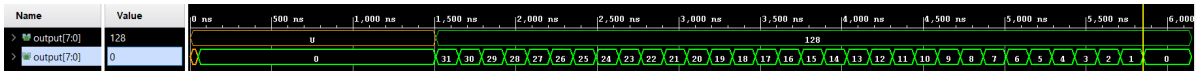
In questo Test Bench si valuta la prima condizione limite ovvero che se le prime parole lette sono 0 allora bisognerà assegnare loro credibilità 0 fino alla lettura di un valore diverso, inoltre viene testato il valore massimo che può essere assegnato alla parola. La sequenza utilizzata è breve ma sufficiente a dimostrare il punto: i valori iniziali sono (0,**0**, 0,**0**, 255,**0**, 0,**0**, 0,**0**) e quelli attesi (0,**0**, 0,**0**,

255,**31**, 255,**30**, 255,**29**); per mantenere una lettura più agevole sono rappresentati solo i segnali fondamentali, ovvero **o_mem_en**,**o_mem_we** ,**o_mem_data** e **i_mem_data**, e la memoria interessata.
Il componente ha passato il test anche in post sintesi.



3.2.3 Test Bench 2

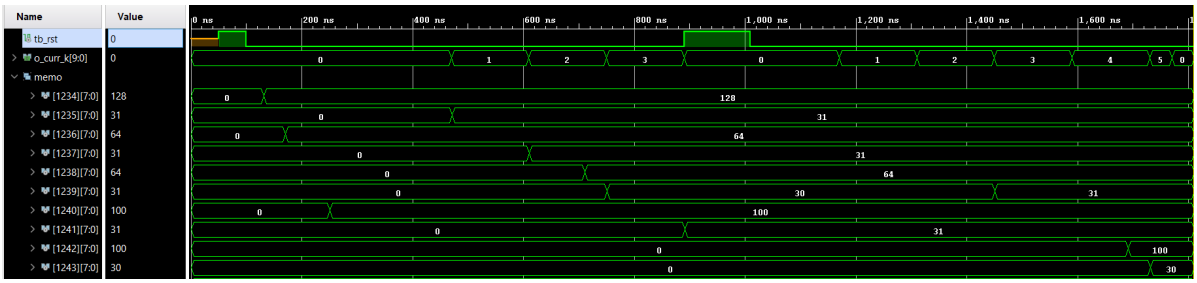
In questo Test Bench viene valutata l'altra condizione limite del componente, ovvero una sequenza di 0 di lunghezza maggiore di 31, in questo caso la credibilità C assumerà un valore dettato da **max**(0,prevC-1). A differenza del test precedente si necessita di una quantità di dati abbastanza lunga per essere valutato, quindi non verranno riportati, anche se quest'ultima è molto semplice in quanto consiste in un valore iniziale 128 seguito da una fila di 32 volte 0; inoltre data la grossa quantità di dati verranno rappresentati solo il valore assunto dalla parola e dalla credibilità associata.
Il componente ha passato il test anche in post sintesi.



si noti che il valore di 0 alla fine è più lungo in quanto è rimasto costante

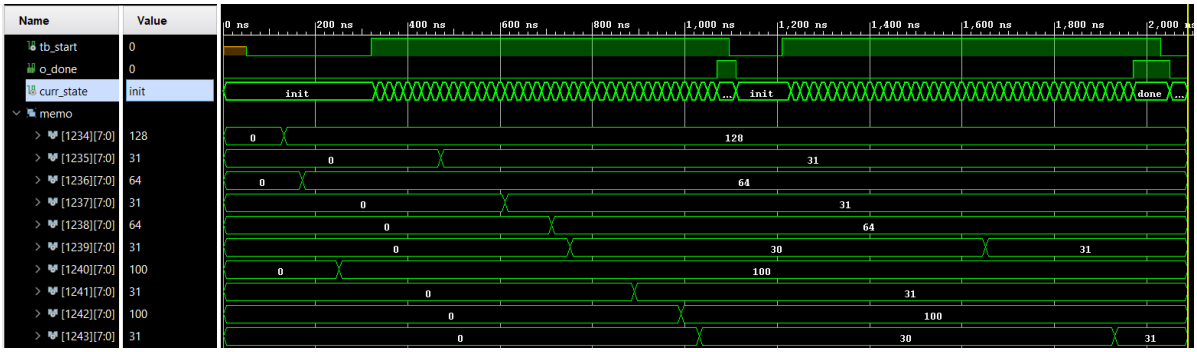
3.2.4 Test Bench 3

In questo Test Bench viene testata la funzionalità di reset del componente: verrà lasciato svolgere una parte del Test Bench 0 abbreviato e poi verrà alzato il segnale di **i_rst**, il componente quindi dovrà resettarsi correttamente e svolgere l'esercizio da capo con una parte di valori cambiati. La sequenza iniziale è (128, 0, 64, 0, 0, 0, 100, 0, 0, 0) che dopo il reset, che avviene al terzo ciclo, diventa (128, 31, 64, 31, 64, 30, 100, 0, 0, 0); il valore di output sarà quindi (128, 31, 64, 31, 64, 31, 100, 31, 100, 30).
Il componente ha passato il test anche in post sintesi.



3.2.5 Test Bench 4

In questo Test Bench viene testata l'ultima funzionalità del componente, ovvero la possibilità di svolgere altri test uno dopo l'altro senza un segnale di reset a dividere. Si è utilizzata la stessa sequenza di prima anche se ora i valori dopo la prima iterazione sono diventati (128, 31, 64, 31, 64, 30, 100, 31, 100, 30), in output quindi avrò (128, 31, 64, 31, 64, 31, 100, 31, 100, 31).
Il componente ha passato il test anche in post sintesi.



3.2.6 Test Bench 5

Infine è stato creato un generatore di Test Bench in Python 3.11.1, il cui codice è allegato qui sotto, che genera test di lunghezza variabile insieme alla loro soluzione. Tutti questi test sono stati passati anche in post sintesi.

C: > Users > alber > import random.py > testgenerator

```python
import random
def testgenerator(nW):
    prevC=0
    prevW=0
    pre =[]
    for j in range(0,nW):
        if random.randint(0,1)==0:
            pre.append(0)
            pre.append(0)
        else:
            pre.append(random.randint(0,255))
            pre.append(0)

    print(pre)
    print("\n")
    for i in range(0, len(pre),2):
        if pre[i]==0:
            pre[i]=prevW
            prevC-=1
            pre[i+1]=max(prevC,0)
        else:
            prevW=pre[i]
            prevC=31
            pre[i+1]=31
    print(pre)
testgenerator(100)

```

In questo esempio si crea un test di lunghezza 100 parole

4 Conclusioni

Il componente è stato più che ampiamente testato in ogni sua funzionalità e ha sempre esibito un comportamento corretto sia in pre che post sintesi; inoltre poiché non ci sono vincoli sulla quantità di tempo che deve impiegare per la risoluzione di un compito si è deciso di andare verso una struttura più semplice e meno efficiente, rendendola tuttavia più chiara e di facile comprensione.
In conclusione il circuito, nonostante alcune componenti non perfettamente ottimizzate, rispetta la specifica in ogni suo termine.