POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Prova finale di Reti Logiche

Docente: prof. William Fornaciari

Riccardo Bonfanti

Matricola: 983038

Codice Persona: 10779115

E-mail: riccardo5.bonfanti@mail.polimi.it

Anno Accademico 2023/24

INDICE

1. Introduzione

1.1. Scopo del progetto e descrizione delle specifiche generali

Il progetto ha come obiettivo quello di descrivere, nel linguaggio VHDL, e sintetizzare, tramite la suite software VIVADO, un componente hardware che implementa la specifica fornita dai docenti. Il modulo sviluppato ha il compito di interfacciarsi con la memoria, leggendo una sequenza di i_k parole a partire da un determinato indirizzo (i_add) e scrivendo, a seguito di un'elaborazione, l'output desiderato.
Si noti che d'ora in avanti i termini modulo-componente e parola-dato saranno usati come sinonimi.
L'elaborazione consiste nel leggere una parola dalla memoria, che conterrà un valore tra 0 e 255, e assegnargli un valore di credibilità compreso tra 0 e 31, che verrà scritto all'indirizzo di memoria successivo. A seguito dell'elaborazione, il contenuto della memoria sarà simile a quello rappresentato nella *Tabella 1*.

Tabella 1 - Descrizione della memoria dopo l'elaborazione

I_ADD	I_ADD + 1	I_ADD + 2	I_ADD + 3	…	I_ADD + 2*(I_K – 1)	I_ADD + 2*I_K
Valore	Credibilità	Valore	Credibilità	…	Valore	Credibilità

Se e quando nella sequenza di parole si legge 0, esso andrà interpretato come dato non specificato e dovrà essere sostituito con il dato diverso da 0 letto più recentemente. La credibilità è assegnata in base al valore della sequenza letto: se esso è diverso da 0 allora sarà pari a 31; altrimenti se esso è 0, la credibilità verrà decrementata di 1 rispetto al valore precedente, a meno che essa non sia già 0.
Nel caso particolare in cui la sequenza inizia con 0, il valore dovrà rimanere tale e la sua credibilità posta a 0.
Per capire meglio, analizziamo la sequenza (in grassetto) dell'*Esempio 0* prima e dopo l'elaborazione:

128 0 **64** 0 **0** 0 **0** 0 **0** 0 **0** 0 **0** 0 **100** 0 **1** 0 **0** 0 **5** 0 **23** 0 **200** 0 **0** 0
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. Interfaccia del componente

L'interfaccia del componente è rappresentata nelle specifiche del progetto ed è stata riportata nel *Frammento di codice 1*.

Frammento di codice 1 - Interfaccia del componente

```
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;
```

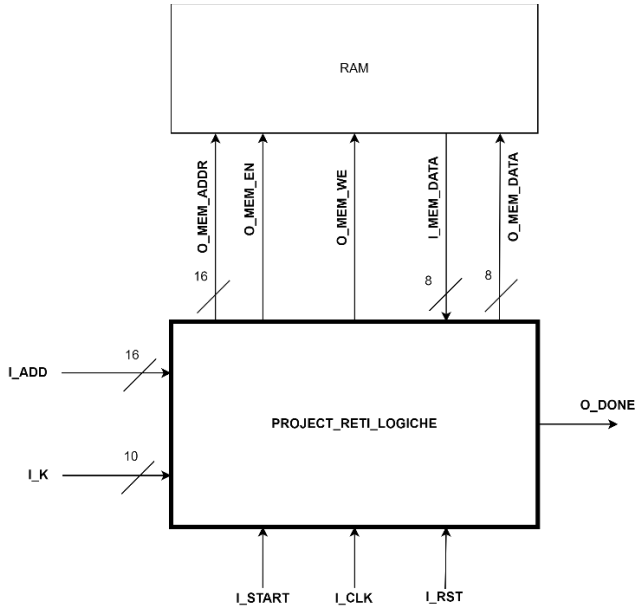
- Il segnale i_clk è il CLOCK generato dal testbench, unico per tutto il sistema.
- i_rst è invece il segnale di RESET asincrono che inizializza il componente rendendolo pronto per ricevere il segnale di START. Prima del primo reset il funzionamento del componente non è specificato.
- Il segnale i_start viene generato dal testbench e serve per far partire l'elaborazione. Esso rimarrà alto finché il segnale o_done non verrà alzato, dopodiché può essere riportato a 0. Una successiva elaborazione può riportare i_start a 1 senza dover attendere il reset del componente.
- L'indirizzo di partenza e la lunghezza della sequenza di parole da leggere saranno forniti tramite segnali, nello specifico vettori da 16 e 10 bit, in ingresso al componente (i_add, i_k).
- Il componente dovrà comunicare che ha terminato l'elaborazione tramite un segnale in uscita o_done. Questo segnale deve rimanere alzato fintantoché quello di i_start non è riportato a 0.

Il componente si interfaccia con la memoria tramite i seguenti segnali:

- o_mem_addr (vettore di 16 bit): è un'uscita che manda l'indirizzo alla memoria;
- o_mem_en: è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- i_mem_data (vettore di 8 bit): è un segnale di ingresso che contiene il dato richiesto in lettura;
- o_mem_we: è il segnale di WRITE ENABLE da dover mandare alla memoria insieme all'ENABLE per poterci scrivere. Per leggere da memoria esso deve essere 0;
- o_mem_data (vettore di 8 bit): è un'uscita che va verso la memoria e contiene il dato che verrà successivamente scritto.

A partire da queste informazioni possiamo costruire un grafico che rappresenta il componente da sviluppare come una "scatola nera", in modo tale da mettere in evidenza i segnali di ingresso e uscita e le interazioni con la memoria RAM (*Figura 1*).

Figura 1 - Interfaccia del componente



1.3. Descrizione della memoria

La memoria è istanziata all'interno del testbench e non va quindi sintetizzata. Il processo che implementa la memoria è rappresentato nel *Frammento di codice 2*.

La memoria è sincrona: il processo si attiva a ogni variazione del clock che si trova infatti nella sensitivity list. Il clock è generato dal testbench e come già detto è unico per tutto il sistema.

Se i segnali di ENABLE e WRITE-ENABLE in ingresso sono alti, allora, dopo 1 ns, sarà scritto il dato in arrivo dal segnale in ingresso all'indirizzo comunicato a sua volta in ingresso. Dopodiché metterà in uscita il dato stesso. Se invece solo ENABLE è alto sarà messo in uscita, dopo 1 ns, il dato letto all'indirizzo in ingresso.

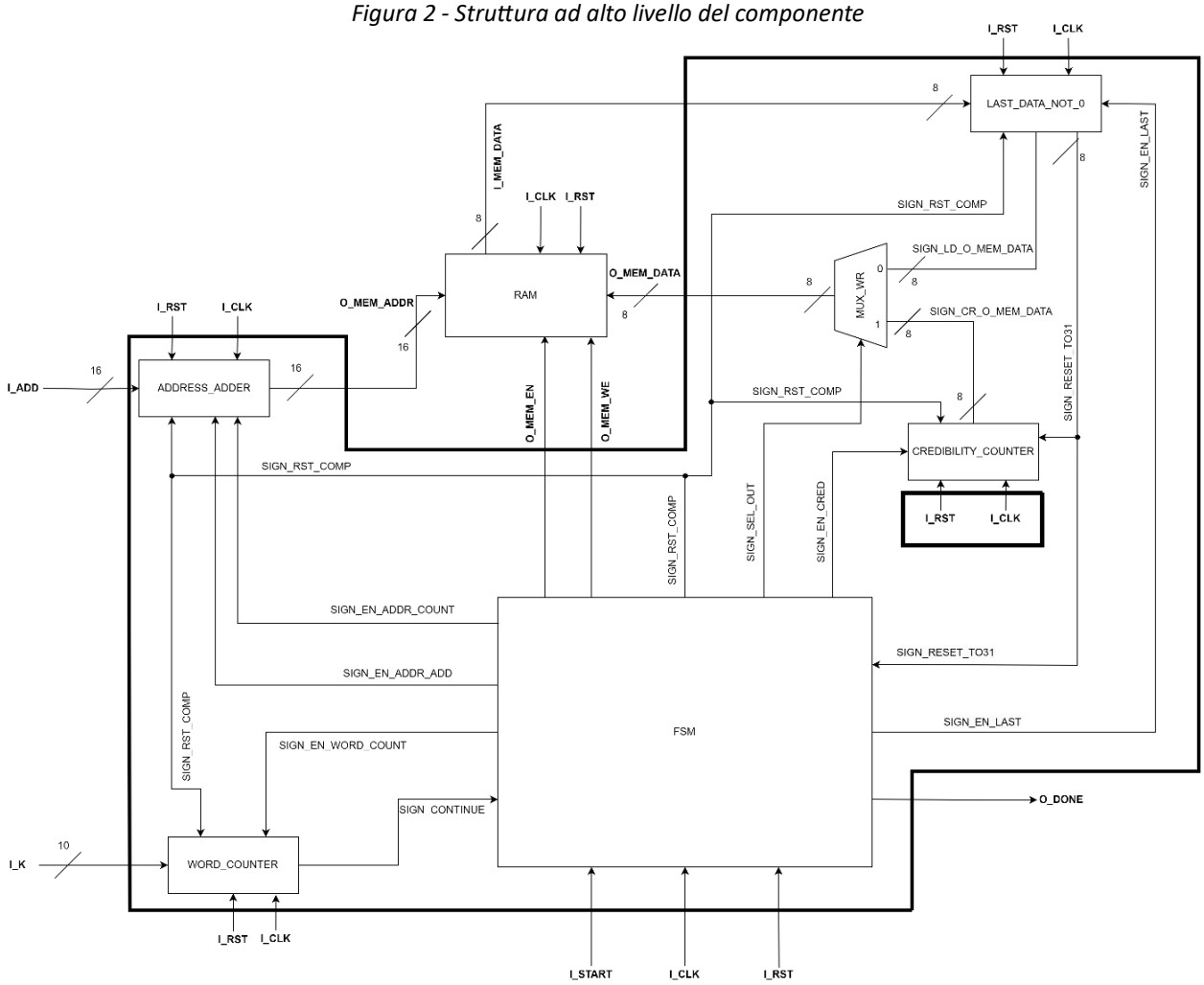
Frammento di codice 2 - Implementazione della memoria

```
MEM : process (tb_clk)     -- Process related to the memory
begin
   if tb_clk'event and tb_clk = '1' then
       if tb_o_mem_en = '1' then
           if tb_o_mem_we = '1' then
               RAM(to_integer(unsigned(tb_o_mem_addr))) <= tb_o_mem_data after 1 ns;
               tb_i_mem_data <= tb_o_mem_data after 1 ns;
           else
               tb_i_mem_data <= RAM(to_integer(unsigned(tb_o_mem_addr))) after 1 ns;
           end if;
       end if;
   end if;
end process;
```

2. Architettura

Ad alto livello, la struttura del componente si presenta come nella *Figura 2*.

Figura 2 - Struttura ad alto livello del componente



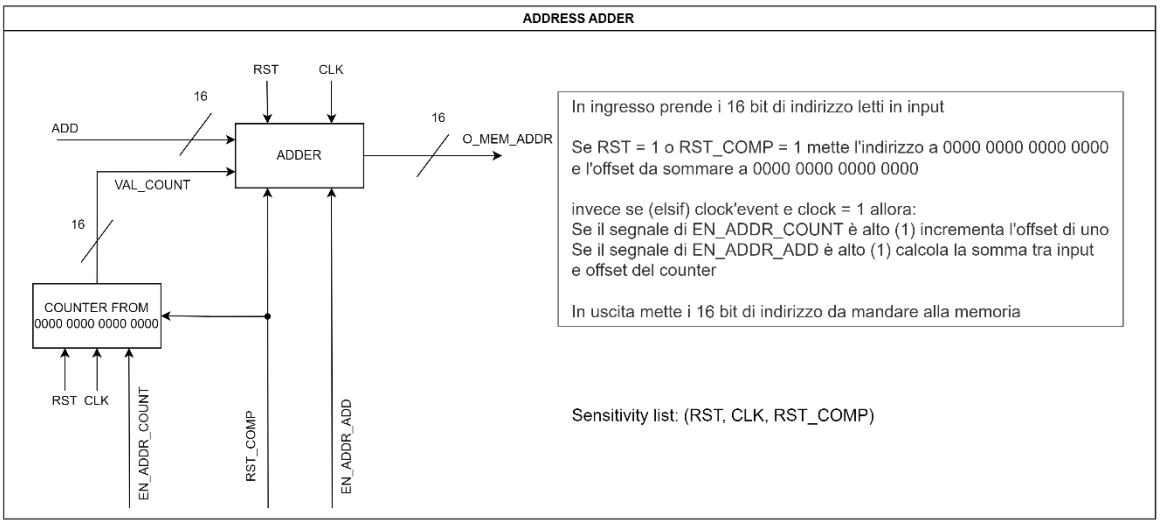
Il componente, delimitato dalla linea nera più spessa, si suddivide a sua volta in 6 sottocomponenti che svolgeranno una funzione specifica, descritta dal loro nome. Abbiamo quindi: address_adder, word_counter, last_data_not_zero, credibility_counter, mux_wr e FSM.
Tramite questa frammentazione del problema è stato più semplice gestire la logica del componente e regolare il suo funzionamento. Inoltre, il codice risulta essere più ordinato in quanto la divisione in entity diverse, con la loro architettura, evidenzia una separazione netta tra i moduli.
Grazie a questa divisione, l'unico compito del componente principale sarà quello di connettere i sottocomponenti tramite segnali.
Vediamo ora i moduli nel dettaglio.

2.1. Modulo address_adder

Il modulo address_adder ha il compito di leggere il segnale i_add ricevuto in ingresso, che corrisponde all'indirizzo iniziale da cui leggere la sequenza in memoria. Siccome la specifica dice che il segnale i_add rimarrà fisso per tutta l'elaborazione non sarà necessario salvarlo in un registro. Il segnale, che è un vettore di 16 bit, sarà sommato al valore di output di un contatore che, per ogni parola letta o credibilità scritta, incrementa l'offset da cui leggere o scrivere sulla memoria.
Il contatore parte da un valore di 0 (espresso in 16 bit) e aumenta di 1 quando riceve il segnale di enable fornito dall'FSM: en_addr_count.
Il sommatore, invece, viene attivato da un altro segnale di enable fornito dalla macchina a stati finiti: en_addr_add. L'uscita del modulo è un segnale a 16 bit che rappresenta l'indirizzo inviato alla memoria.

Sia il sommatore che il contatore vengono resettati a 0 quando ricevono il segnale di reset del sistema o quando ricevono il segnale di reset del componente (fornito dalla finite state machine).
La *Figura 3* presenta una descrizione grafica del componente e lo pseudocodice del suo funzionamento.

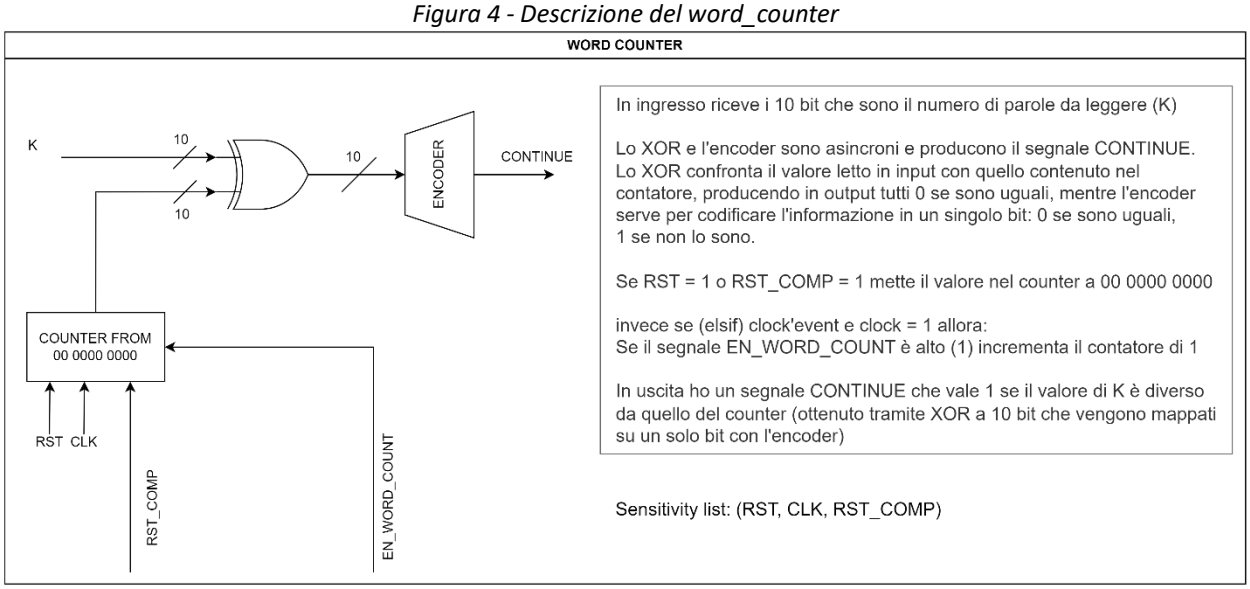
Figura 3 - Descrizione dell'address_adder



2.2. Modulo word_counter

Il modulo word_counter ha il compito di leggere la lunghezza della sequenza di parole in ingresso (i_k) e confrontarla con il valore di output di un contatore.
Il contatore partirà da 0 (espresso in 10 bit) e verrà incrementato ogni volta che un nuovo valore della sequenza viene letto e il segnale di enable fornito dalla FSM (en_word_count) è alto.
Il confronto tra l'output del contatore e il segnale i_k avviene tramite uno XOR e ha come obiettivo quello di controllare se i due segnali sono uguali. Una volta che il contatore avrà raggiunto il valore i_k avremo infatti letto tutte le parole della sequenza e l'uscita dello XOR sarà un vettore di 10 bit tutti a 0.
Il compito dell'encoder è infine quello di codificare l'output dello XOR in un segnale di 1 singolo bit. Il segnale continue avrà quindi valore 1 tutte le volte che l'ingresso dell'encoder sarà un segnale in cui almeno uno dei 10 bit avrà valore 1 (ciò indica una differenza tra l'input e il contatore in quel bit), mentre avrà valore 0 quando l'ingresso dell'encoder sarà un vettore di soli 0.
Il segnale continue è inviato alla FSM, che in base al suo valore decide se rimanere nel ciclo di stati di elaborazione oppure passare alla sequenza di stati che porteranno alla terminazione dell'elaborazione stessa.
Il counter è resettato a 0 quando riceve il segnale di reset del sistema o quello di reset del componente.
La *Figura 4* presenta una descrizione grafica del componente e lo pseudocodice del suo funzionamento.

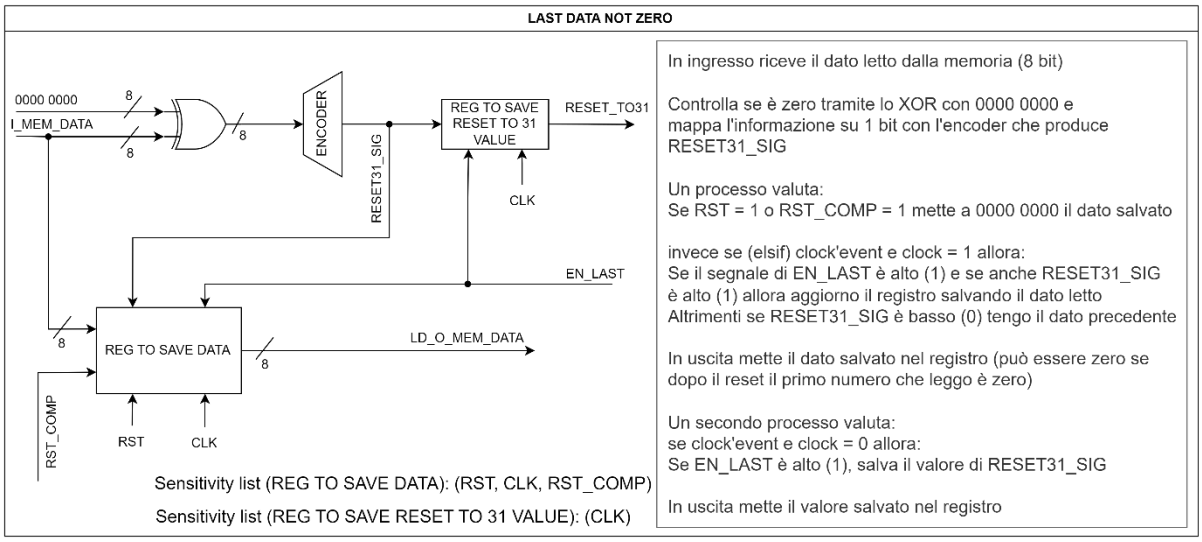
Figura 4 - Descrizione del word_counter



2.3. Modulo last_data_not_zero

Il componente last_data_not_zero ha il compito di ricevere il dato dalla memoria e, dopo aver verificato se è 0 confrontandolo con un vettore di otto 0 tramite uno XOR, decidere se salvare tale dato in un registro.
Se quindi il dato letto è “0000 0000” allora lo XOR produrrà un vettore di soli 0 che sarà codificato dall’encoder come un segnale di un bit con valore 0, altrimenti in tutti gli altri casi avrà valore 1.
Quando il segnale di enable fornito dalla FSM (en_last) è alto allora se anche il segnale prodotto dall’encoder è alto verrà aggiornato il dato salvato nel registro con quello attuale (REG TO SAVE DATA) e fornito in uscita tramite un segnale a 8 bit verso il componente mux_wr, altrimenti verrà mantenuto il dato precedente.
Il segnale prodotto dall’encoder verrà a sua volta salvato in un registro. Quest’ultimo è collegato a un segnale di uscita reset_to31 che sarà servirà al componente credibility_counter e all’FSM per prendere decisioni.
Il registro contenente il dato è resettato a “0000 0000” quando riceve il segnale di reset del sistema o quello del componente.
La *Figura 5* presenta una descrizione grafica del componente e lo pseudocodice del suo funzionamento.

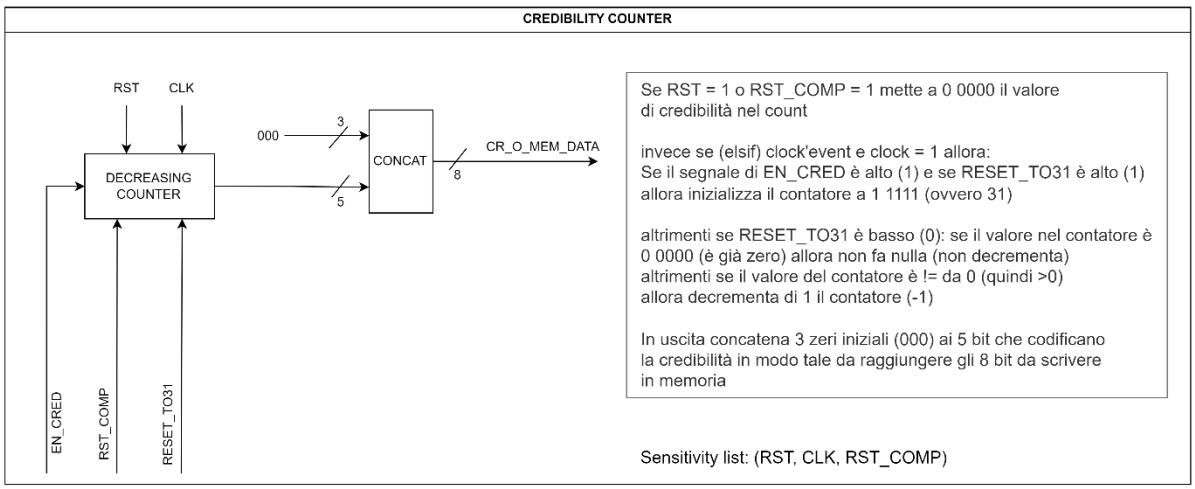
Figura 5 - Descrizione di last_data_not_zero



2.4. Modulo credibility_counter

Il modulo credibility_counter ha il compito di calcolare il valore di credibilità associato alla parola letta.
Quando il segnale di enable fornito dalla FSM (en_cred) è alto, allora valuta se anche il segnale di reset_to31 è alto e in questo caso riporta il valore del contatore a 31, ovvero “1 1111” poiché significa che il dato letto non è 0, altrimenti viene decrementato di 1 il valore del contatore (solo se questo è superiore a “0 0000”).
Per produrre il segnale di uscita, concateno “000” al valore del contatore in modo da raggiungere gli 8 bit.
Il contatore è resettato a 0 quando riceve il segnale di reset del sistema o quello del componente.
La *Figura 6* presenta una descrizione grafica del componente e lo pseudocodice del suo funzionamento.

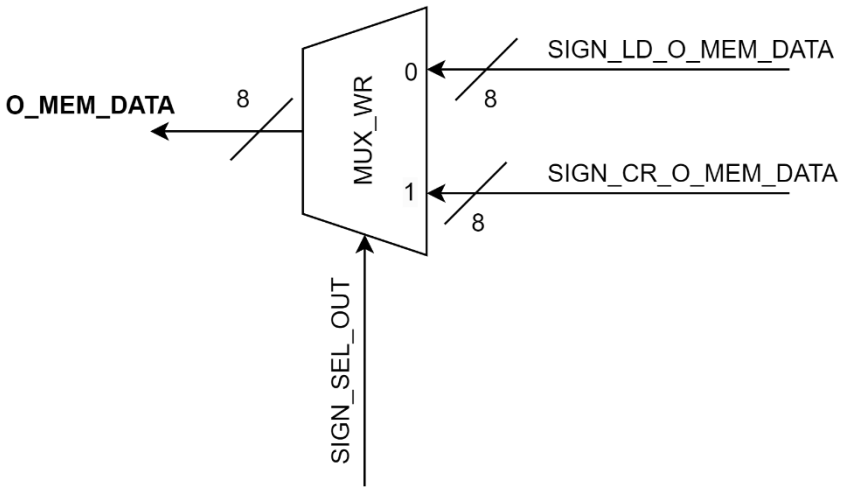
Figura 6 - Descrizione del credibility_counter



2.5. Modulo mux_wr (multiplexer to write)

Il modulo mux_wr ha il compito di selezionare il segnale che sarà inviato alla memoria per essere scritto.
Come ingressi riceve i segnali provenienti dai componenti last_data_not_zero e credibility_counter, che contengono i valori da scrivere in memoria. Il multiplexer seleziona uno di questi due vettori e lo mette in uscita in base a un segnale sel_out di ingresso proveniente dalla FSM. Se il segnale ha valore 0 allora sarà collegato all'uscita il segnale proveniente dal modulo last_data_not_zero, altrimenti se ha valore 1 verrà collegato in uscita il segnale a 8 bit proveniente dal credibility_counter.
La *Figura 7* presenta una descrizione grafica del componente.

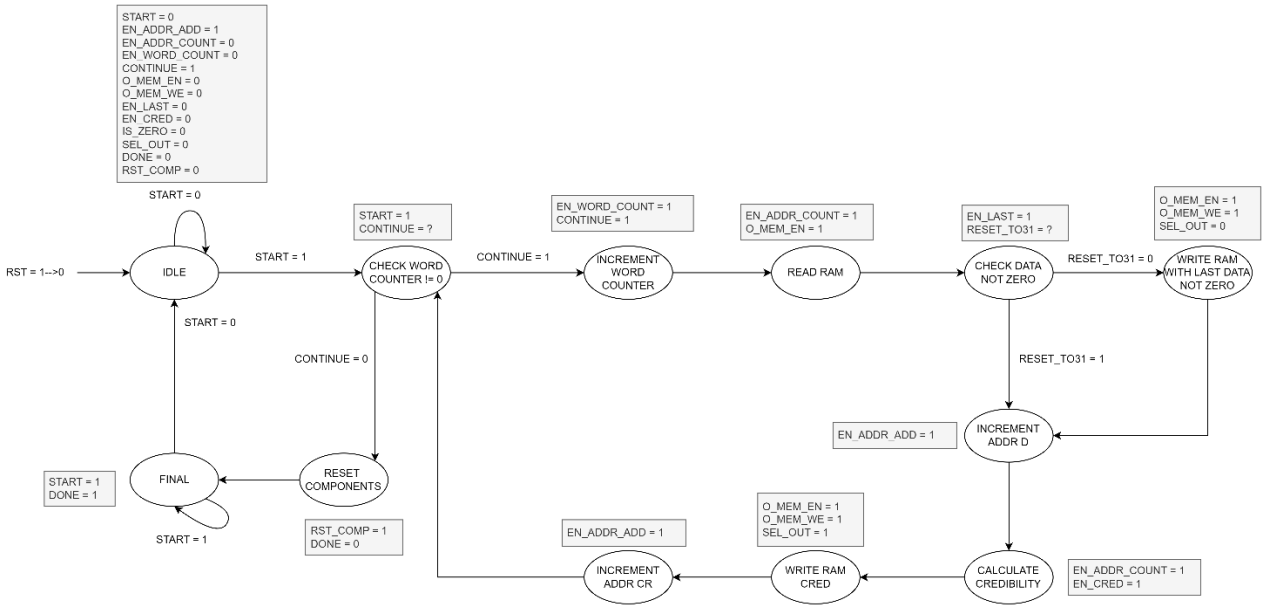
Figura 7 - Descrizione del multiplexer mux_wr



2.6. Modulo FSM (Finite State Machine)

Il modulo FSM ha il compito di definire gli stati in cui il componente può trovarsi e di gestire quindi il flusso elaborativo. La *Figura 8* presenta una descrizione grafica del diagramma degli stati.

Figura 8 - Descrizione degli stati della FSM



L'architettura del componente è composta da due processi: il primo regola il passaggio di stato e contiene nella sua sensitivity list i segnali i_clk e i_rst del sistema; il secondo regola il settaggio dei segnali al variare degli stati. Il segnale di curr_state immagazzina l'informazione sullo stato attuale.
Ogni ciclo di clock comporta un passaggio di stato, ad eccezione degli stati IDLE e FINAL dove il tempo impiegato per passare al nuovo stato dipende dal segnale in ingresso i_start.
Dopo aver ricevuto il segnale di reset (i_rst), in qualsiasi stato la macchina si trovi, essa viene riportata allo

stato di IDLE. Fintantoché non viene alzato il segnale di i_start la macchina rimarrà nello stato di IDLE. Quando è in questo stato, tutti i segnali gestiti dalla FSM vengono posti a 0 tranne continue ed en_addr_add (che viene posto a 1 per far sì che l'indirizzo iniziale sia collegato alla memoria).
Quando i_start viene posto a 1 passiamo allo stato di CHECK_WCOUNT in cui si valuta il valore del segnale continue. Se esso ha valore 1 si continua nell'elaborazione passando allo stato INCR_WCOUNT, altrimenti si entra nello stato di RESET_COMPONENTS.
Nello stato INCR_WCOUNT impostiamo en_word_count a 1 in modo tale che si attivi il modulo word_counter, che incrementa il contatore delle parole lette.
Lo stato successivo è quello di READ_RAM dove en_addr_count è settato a 1, in modo tale che il contatore dell'offset da sommare all'indirizzo contenuto nell'address_adder sia incrementato, e il segnale o_mem_en è anch'esso portato a 1, cosicché sia possibile leggere il dato dalla memoria.
Si passa poi a CHECK_DATA_NOT0 che alza il segnale di en_last diretto al modulo last_data_not_zero in modo tale che si possa valutare se la parola letta sia diversa da 0 oppure no. Nel primo caso l'FSM leggerà in ingresso un segnale reset_to31 = 1 e procederà quindi con lo stato INCR_ADDR_D, mentre se invece il segnale di reset_to31 è basso dobbiamo prima sovrascrivere il valore non valido 0 con l'ultimo valore valido letto che è salvato nel registro, passando allo stato WRITE_RAM_DATA.
Nello stato WRITE_RAM_DATA vengono alzati i segnali o_mem_en e o_mem_we, in modo tale che sia possibile scrivere sulla memoria, e viene settato il segnale sel_out a 0, cosicché il multiplexer colleghi l'uscita del registro del componente last_data_not_zero alla memoria.
Lo stato INCR_ADDR_D alza il segnale en_addr_add, che permette la somma tra l'indirizzo fornito in ingresso e l'offset del contatore all'interno del modulo address_adder e collega il risultato al segnale che indica alla memoria l'indirizzo da cui leggere (o su cui scrivere) il dato.
Lo stato successivo si chiama CALC_CRED e alza il segnale di en_cred, che attiva il componente credibility_counter e permette di calcolare la credibilità associata al dato appena letto dalla memoria. Inoltre, viene anche alzato il segnale en_addr_count, che incrementa il contatore dell'offset relativo all'indirizzo, in modo tale che esso sia pronto quando verrà eseguita la somma.
Abbiamo poi lo stato WRITE_RAM_CRED, che alza i segnali o_mem_en e o_mem_we, per poter scrivere sulla memoria, e imposta sel_out a 1 in modo tale che a essere scritto sarà il segnale di uscita inviato dal modulo credibility_counter.
Nello stato INCR_ADDR_CR viene alzato il segnale en_addr_add che calcola, tramite la somma tra indirizzo in iniziale e offset, il nuovo indirizzo di memoria su cui andremo ad operare.
Lo stato successivo a INCR_ADDR_CR è CHECK_WCOUNT che è già stato descritto in precedenza.
Passiamo quindi alla descrizione di RESET_COMPONENTS, che alza il segnale di reset dei singoli componenti (non quello generale del sistema) in modo tale che siano pronti per una nuova elaborazione senza che sia necessario il segnale di reset globale del componente.
Infine, abbiamo lo stato FINAL che imposta il segnale di o_done a 1 e rimane in attesa fintantoché il segnale di i_start non viene riportato a 0. A quel punto si torna nello stato di IDLE e o_done viene abbassato.

3. Risultati sperimentali

Dopo la fase di progetto e implementazione tramite linguaggio VHDL, il componente è stato sintetizzato, tramite il tool fornito da VIVADO, su un design fisico reale. Seguendo il suggerimento proposto dalla specifica è stata scelta la seguente FPGA: Artix-7 FPGA xc7a200tfbg484-1.
Come vedremo tra poco nel dettaglio, il componente è stato testato nella maniera più approfondita possibile, tramite simulazioni behavioural pre-synthesis e functional post-synthesis, ed è stato in grado di superare ogni testbench a cui è stato sottoposto.

3.1. Sintesi

Il tool di sintesi di VIVADO è stato in grado di sintetizzare il progetto senza generare alcun “Error” o “Warning” relativo alle scelte implementative.
Usando il comando “report_utilization” nella console TCL di VIVADO verrà stampato un report contenente informazioni relative al componente fisico sintetizzato. Come è possibile osservare nella *Figura 9*, che riporta una delle tabelle generate tramite il comando appena descritto, il design sintetizzato da VIVADO è composto da 51 Look-Up Tables e 68 Registers. Ciò che interessa è che tutti i register generati siano Flip-Flop e che quindi non venga generato alcun Latch, poiché la loro presenza nel componente sintetizzato potrebbe provocare funzionamenti indesiderati in simulazione functional post-synthesis. L’assenza di Latch, a meno che siano voluti, è quindi indicatore di un buon design, che molto probabilmente funzionerà anche considerando i ritardi introdotti dalla simulazione post-sintesi del componente fisico.

Figura 9 - Tabella riassuntiva sull'utilizzo generata dal report_utilization

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	51	0	0	134600	0.04
LUT as Logic	51	0	0	134600	0.04
LUT as Memory	0	0	0	46200	0.00
Slice Registers	68	0	0	269200	0.03
Register as Flip Flop	68	0	0	269200	0.03
Register as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

Con il comando “report_timing” andremo invece a verificare se è stato soddisfatto il requisito sul tempo. La console anche questa volta stamperà un lungo report, della quale ci interessa solo la parte riportata nella *Figura 10*.

Figura 10 - Informazioni generate dal comando report_timing

```
Timing Report

Slack (MET) :             6.790ns  (required time - arrival time)
  Source:                 reg_last_data/o_reset_to31_reg/C
                            (falling edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            cred_count/val_cr_count_reg[0]/CE
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (clock rise@20.000ns - clock fall@10.000ns)
  Data Path Delay:        2.828ns  (logic 0.872ns (30.835%)  route 1.956ns (69.166%))
  Logic Levels:           2  (LUT5=1 LUT6=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.100ns = ( 22.100 - 20.000 )
    Source Clock Delay      (SCD):    2.424ns = ( 12.424 - 10.000 )
    Clock Pessimism Removal (CPR):    0.178ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Come è possibile leggere, lo Slack del componente, cioè il massimo ritardo che potrebbe essere introdotto affinché il componente sia ancora funzionante, è 6.790 ns. Ciò significa che il tempo massimo necessario al circuito per commutare è di circa 3,210 ns; un valore lontano da quello del constraint introdotto sul clock.

3.2. Simulazioni

La simulazione del componente è avvenuta in due fasi. Nella prima mi sono limitato a replicare i casi di esempio forniti dalla specifica tramite piccole modifiche al testbench di prova condiviso dai docenti. Una volta superati tutti i testbench di esempio, sia in simulazione behavioural pre-sintesi, che in functional post-sintesi, mi sono dedicato alla scrittura di testbench che coprissero i casi limite che avevo individuato e che non erano già stati coperti dagli esempi. Anche in questa seconda fase le simulazioni effettuate sono state in behavioural pre-sintesi e functional post-sintesi.
I testbench si compongono in generale di diversi segnali e quattro processi: due di essi servono per gestire la memoria RAM (uno di essi ne descrive il funzionamento, mentre l'altro passa il controllo della RAM dal testbench al componente e viceversa), il terzo gestisce il funzionamento dello scenario alzando e abbassando i segnali necessari, mentre l'ultimo serve per avere un report sull'esito della simulazione.
Vediamo quindi nel dettaglio i testbench.

i. Testbench degli esempi

Gli esempi forniti nella specifica sono 6.
Gli esempi 0, cioè quello descritto nell'*Esempio 0* dell'introduzione, 1 , 4 e 5 vanno a testare il componente in casi generici in modo tale da verificare che il funzionamento sia corretto almeno in casi standard. Lo scenario di esempio viene creato tramite i segnali descritti nel *Frammento di codice 3*, che ci serviranno poi anche nella test routine per verificare la correttezza dell'elaborazione.

Frammento di codice 3 - Creazione dello scenario dell'Esempio 0

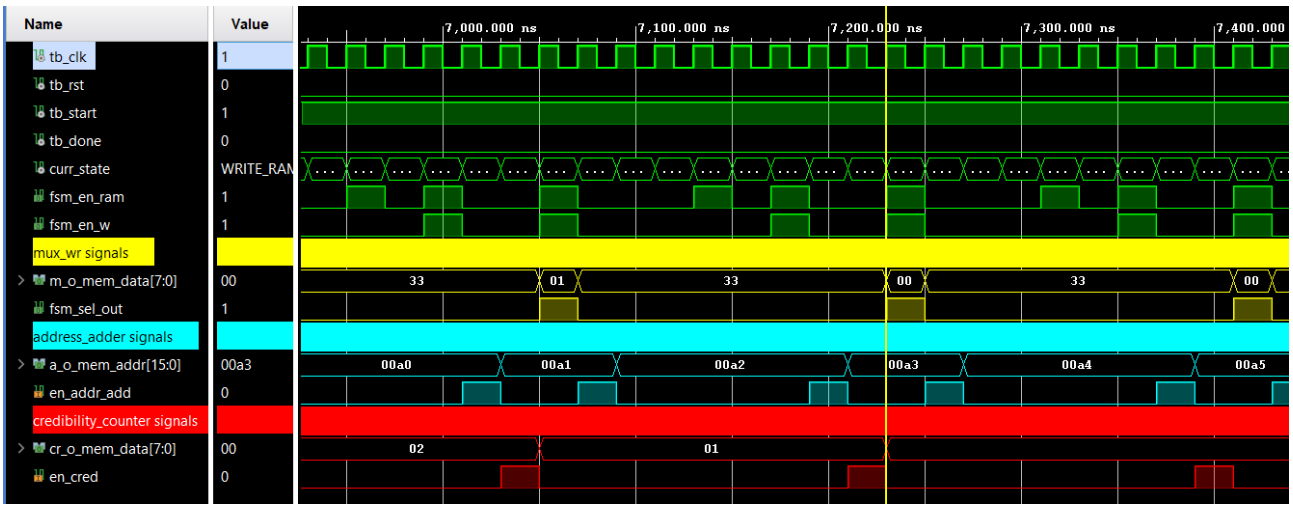
```
constant SCENARIO_LENGTH : integer := 14;
type scenario_type is array (0 to SCENARIO_LENGTH*2-1) of integer;

signal scenario_input : scenario_type := (128, 0,  64, 0,   0,  0,  0,  0,  0,  0,
0,  0,  0,  0, 100,  0, 1,  0, 0,  0, 5,  0, 23,  0, 200,  0,   0,  0 );
signal scenario_full  : scenario_type := (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 );

constant SCENARIO_ADDRESS : integer := 1234;
```

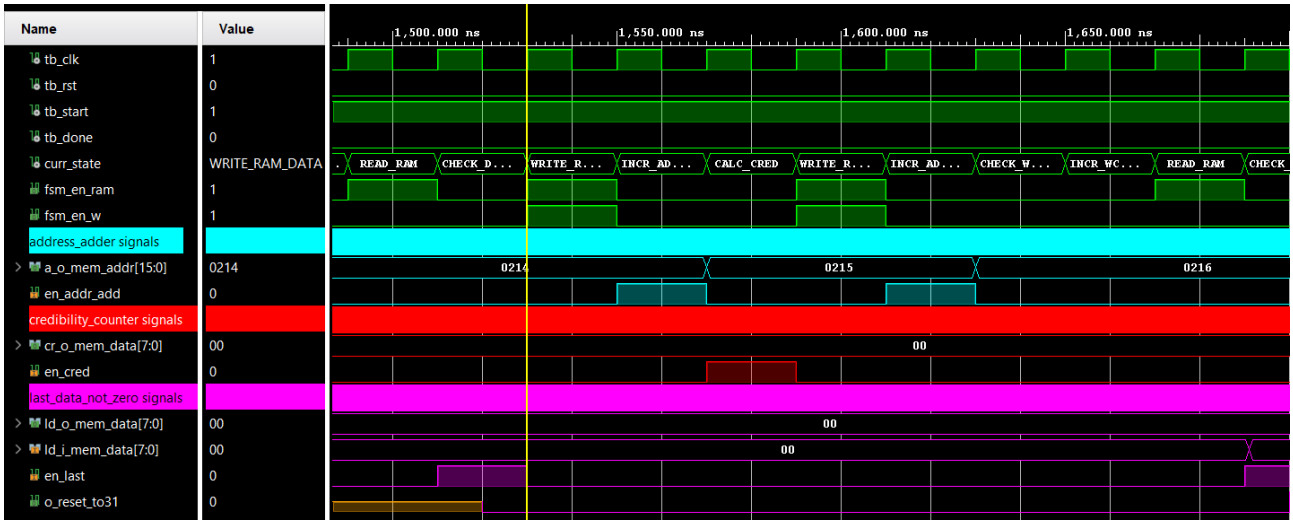
Il processo create_scenario, che regola lo svolgimento della simulazione, inizia scrivendo sulla memoria lo scenario_input. Dopodiché passa il controllo della memoria al componente e alza il segnale di tb_start, collegando anche i segnali di tb_add e tb_k a SCENARIO_ADDRESS e SCENARIO_LENGTH. Si mette quindi ad attendere che il componente alzi il segnale tb_done e infine abbassa il segnale di tb_start.
Il processo di test routine, oltre a valutare la correttezza dell'elaborazione tramite una "assert" che confronta quanto scritto sulla memoria con quanto contenuto nello scenario_full, controlla anche che dei segnali come tb_done e tb_o_mem_en non siano portati a 1 in momenti in cui non dovrebbe essere possibile.
L'esempio 2 testa, invece, il caso limite in cui abbiamo più di 31 zero di fila all'interno della sequenza. Come possiamo osservare dall'*Oscilloscopio 1*, quando il contatore della credibilità arriva a 0 deve smettere di essere decrementato e continuare a stampare 0. La credibilità, quindi, non può mai essere negativa.

Oscilloscopio 1 - Simulazione dell'esempio 2



L'esempio 3, infine, testa il caso limite in cui la sequenza inizia per 0. Come si può notare dall'*Oscilloscopio 2*, quando il modulo last_data_not_zero legge il valore 0 dalla memoria, non avendo ancora salvato un dato valido, stamperà il valore salvato nel registro dopo il reset che è appunto 0. Allo stesso modo il credibility_counter, non avendo ancora ricevuto il segnale reset_to31 da quando è avvenuto il reset del sistema stamperà il valore assegnato con l'inizializzazione, ovvero 0.

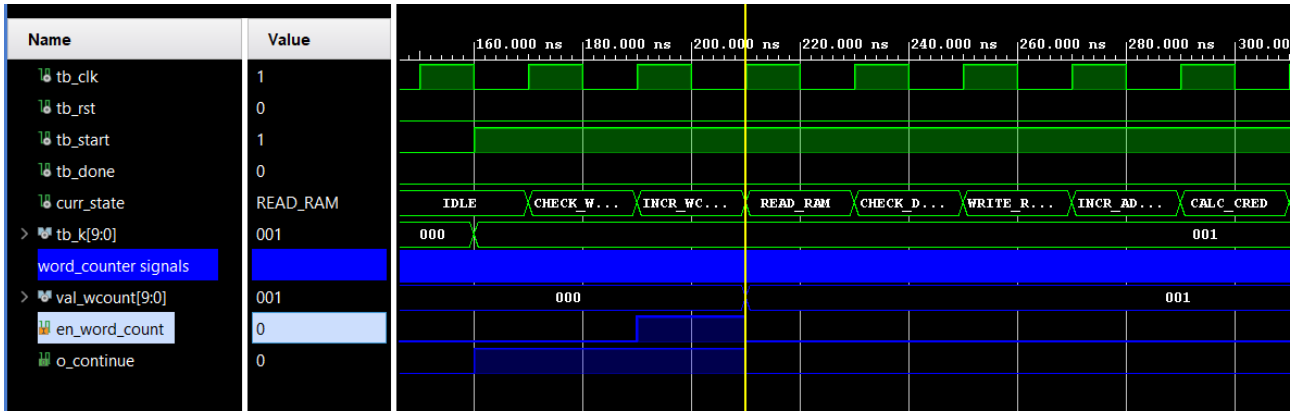
Oscilloscopio 2 - Simulazione dell'esempio 3



ii. Testbench i_k = 1

Il primo caso limite che ho preso in considerazione è stato quello con i_k = 1, ovvero il numero minimo possibile di parole nella sequenza. Come si nota dall'*Oscilloscopio 3*, lo stato successivo al primo controllo incrementa il contatore e a seguito del confronto con i_k (che avviene in modo asincrono), viene abbassato il segnale o_continue. Ciò significa che la prossima volta che si entrerà nello stato CHECK_WCOUNT, che controlla se il numero di parole da leggere è uguale a quello del contatore, si uscirà dal ciclo di elaborazione delle parole per concludere con i due stati finali.

Oscilloscopio 3 - Simulazione del testbench con i_k = 1



iii. Testbench con solo zeri

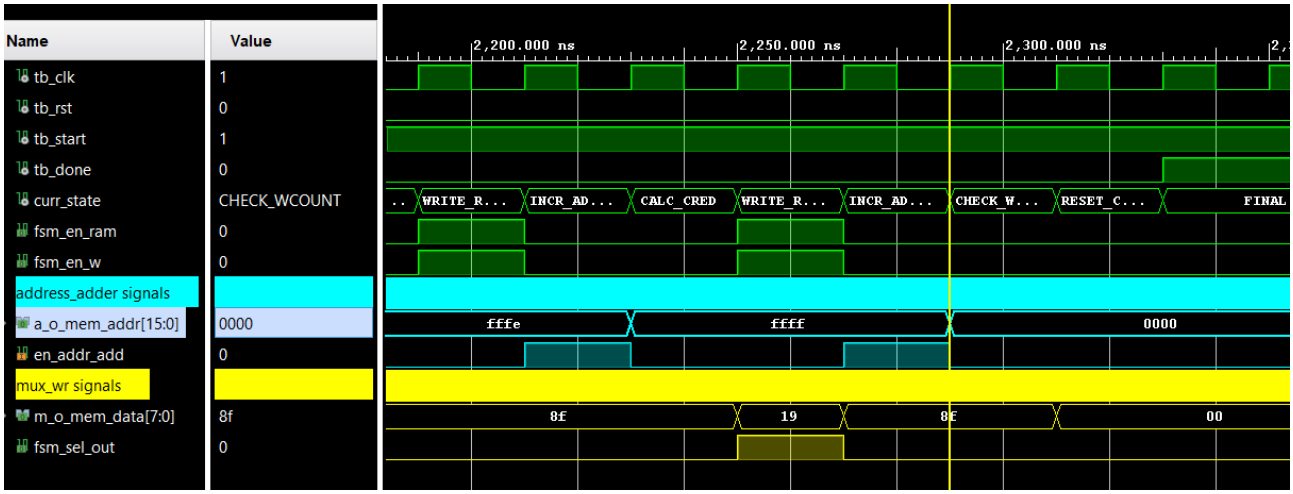
Questo testbench verifica il caso in cui la sequenza è composta da solo 0. Il comportamento deve essere simile a quanto visto nell'Esempio 3: il componente dovrà stampare solo 0 poiché dopo il reset, il dato salvato nei moduli last_data_not_zero e credibility_counter è 0 e non viene mai aggiornato visto che non viene mai letto dalla memoria un valore valido (diverso da 0).

iv. Testbench in cui si raggiunge il massimo indirizzo in memoria

Il testbench analizza il caso limite in cui la sequenza raggiunge l'ultimo indirizzo disponibile per scrivere sulla memoria, ovvero 65.555 (in esadecimale: FFFF). L'*Oscilloscopio 4* mostra come una volta raggiunto l'indirizzo massimo, la somma provoca un overflow che in questo caso riporta l'indirizzo a 0000 (esadecimale). Continuando nell'elaborazione si andrebbe quindi a scrivere nel primo indirizzo della memoria. Tuttavia, il comportamento del componente in caso di overflow non è specificato quindi il test deve fermarsi una volta

arrivato all'indirizzo massimo.

Oscilloscopio 4 - Simulazione del test che raggiunge l'indirizzo massimo

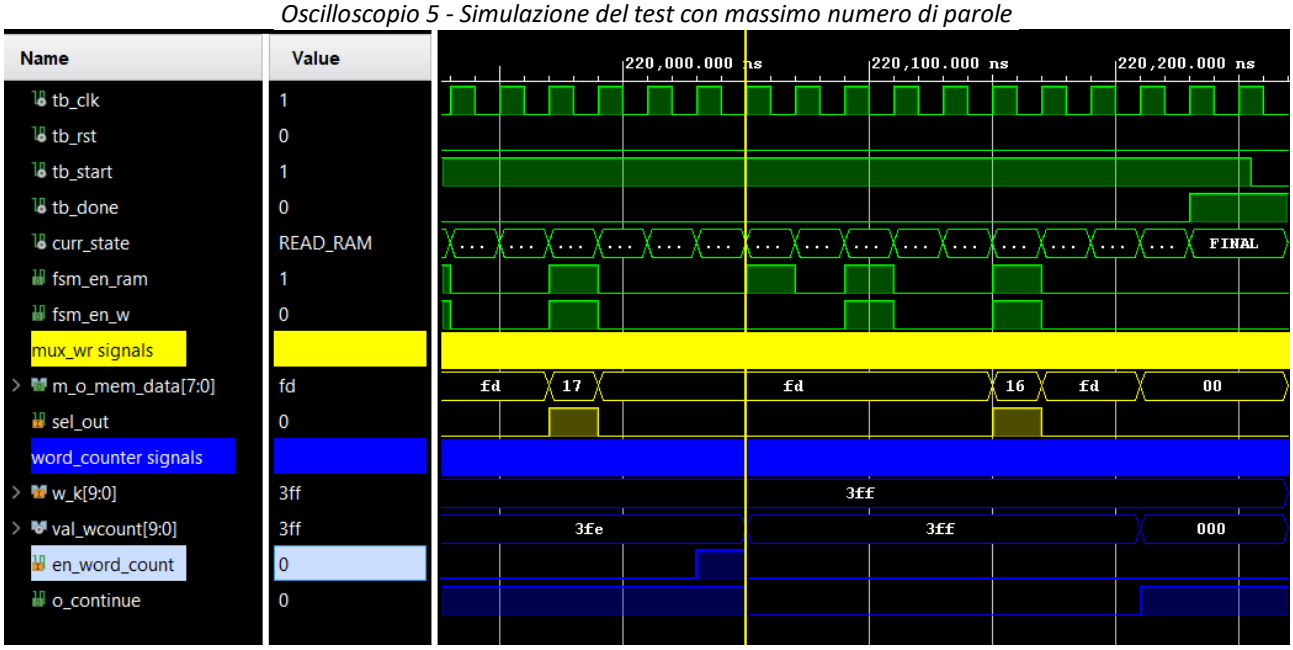


v. Testbench con numero massimo di parole nella sequenza

Il testbench verifica il caso limite in cui abbiamo il massimo numero possibile di parole della sequenza da elaborare. Siccome il numero di parole è codificato come un vettore da 10 bit i_k, allora il numero massimo possibile sarà 2^{10} - 1 = 1023. Per inizializzare i segnali scenario_input e scenario_full (si veda il *Frammento di codice 3*) è stato usato un programma java, ideato dal sottoscritto, che automatizzasse la generazione di casi di test con sequenze di 1023 parole.

Osservando l'*Oscilloscopio 5* notiamo che il contatore del componente word_counter raggiunge il valore 3FF (in esadecimale) e al successivo ingresso nello stato CHECK_WCOUNT passa agli stati di fine elaborazione. Se si provasse ad assegnare un valore maggiore di 1023 come SCENARIO_LENGTH, solamente i 10 bit meno significativi del numero inserito convertito in binario sarebbero considerati. Ad esempio, inserendo 1025, verrà considerato i_k = 1. Tuttavia, le specifiche non prevedono che sia possibile inserire valori che superano i 10 bit, quindi l'esempio appena descritto non sarebbe un test valido.

Oscilloscopio 5 - Simulazione del test con massimo numero di parole



vi. Testbench con più sequenze

Questo testbench analizza il caso in cui alla fine di un'elaborazione, dopo che o_done è riportato a 0, viene iniziata una nuova elaborazione riportando i_start = 1 e ricevendo un nuovo indirizzo i_add e un nuovo numero di parole della sequenza i_k. Osservando l'*Oscilloscopio 6* si nota che prima dello stato finale tutti i sottocomponenti sono riportati nel loro stato iniziale e quindi sono pronti per una successiva elaborazione. Quando i_start torna a 1 il funzionamento dei moduli è analogo a quello della prima elaborazione.

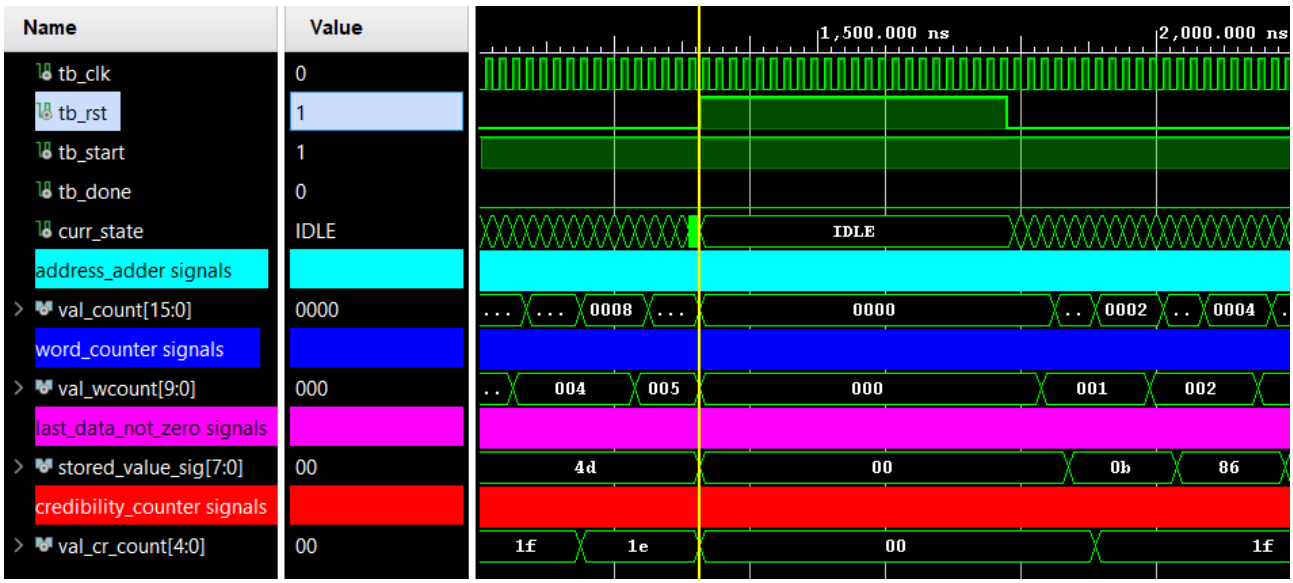
Oscilloscopio 6 - Simulazione del testbench con più elaborazioni

Questo testbench ha richiesto una modifica più ampia del codice del testbench che ci è stato dato di esempio. È stata infatti aggiunta una nuova sezione equivalente a quella del *Frammento di codice 3*, in cui però il nome dei segnali cambia in: SCENARIO_LENGTH2, SCENARIO_ADDRESS2… Di conseguenza anche nel processo create_scenario sono state aggiunte nuove istruzioni per lanciare la seconda elaborazione (i_start torna a 1 dopo un certo numero di ns) e infine è stata aggiunta una “assert” nella test routine per verificare che la seconda elaborazione fosse avvenuta con successo.

vii. Testbench con reset a metà esecuzione

Il testbench verifica il caso in cui è ricevuto un segnale di reset mentre l’esecuzione è in corso. Come da specifica, il reset forza la FSM nello stato IDLE, impedendole di cambiare stato fintantoché il segnale rimane alto, e inizializza tutti i sottocomponenti. A seguito del reset si può scegliere se ricominciare da capo con l’elaborazione precedente (lasciando invariati i_k e i_add e lasciando i_start alto), facendo tuttavia attenzione al fatto che la sezione di memoria scritta prima del reset sarà stata sporcata e dovrà essere ripulita prima di abbassare il reset, oppure posso scegliere di iniziare una nuova elaborazione (cambiando i_k e i_add e alzando i_start dopo averlo abbassato). Parte della simulazione è riportata nell’*Oscilloscopio 7*.

Oscilloscopio 7 - Simulazione del testbench con reset



viii. Testbench con più sequenze e reset

Questo testbench non verifica alcun caso limite in particolare, ma mi è servito per sottoporre il componente ad uno stress maggiore. Ho infatti combinato diversi dei casi limite visti in precedenza, come quello delle sequenze multiple, quello del reset, quello della sequenza di lunghezza massima, in modo tale che potessero essere evidenziate eventuali problematiche che con testbench mirati non venivano identificate.

4. Conclusioni

Questo progetto è stata la prima occasione in cui mi sono trovato a dover sviluppare un componente hardware che soddisfacesse delle specifiche.
Le parti che mi hanno richiesto più impegno sono state la suddivisione delle funzionalità richieste dalla specifica nei sottocomponenti, la progettazione della FSM e la fase di testing.
Il progetto può essere sicuramente modificato, ottimizzato e migliorato in alcuni punti; ad esempio, se si volesse evitare che la sintesi generi sul design fisico molti buffer, potrebbe essere utile inserire dei registri che salvano gli ingressi i_add e i_k (seppur, come detto, non sia strettamente necessario visto che rimangono costanti per tutta l'elaborazione). Un'altra modifica che si potrebbe fare sarebbe quella di aggiungere un nuovo stato per salvare il segnale reset_to31 nel registro e quindi evitare di attivare il registro sul fronte di discesa del clock, ottenendo così uno Slack time di 16.790. Tuttavia, si è deciso di condensare tutto all'interno dello stesso stato CHECK_DATA_NOT0 perché il segnale da salvare (reset31_sig) è asincrono e quindi è pronto molto prima del fronte di discesa del clock.
Grazie al progetto sono riuscito a familiarizzare con un metodo di approcciarsi alla risoluzione dei problemi mai visto in precedenza: ho dovuto imparare un nuovo linguaggio e ad orientarmi in un nuovo ambiente di sviluppo, oltre che a adottare un punto di vista che avesse come priorità l'hardware invece del software. Personalmente ho quindi ritenuto molto utile e educativa la sfida proposta da questo progetto e credo che le competenze acquisite mi saranno molto utili nel proseguimento dei miei studi.