POLITECNICO
MILANO 1863

Relazione prova finale

Progetto di Reti Logiche

Grillo Valerio

Prof. *Salice Fabio*

Giugno 2024

Indice

1 Introduzione

1.1 Obiettivo

L'obiettivo di questo progetto è sviluppare un modulo hardware, codificato in VHDL, che rispetti un'interfaccia per le porte di ingresso e di uscita, che utilizzi una memoria e rispetti delle specifiche. Il modulo dovrà poi essere simulabile e sintetizzabile correttamente utilizzando *Xilinx Vivado Webpack* su una FPGA (i.e. Artix-7 FPGA xc7a200tfbg484-1) con un periodo di clock di almeno 20 ns.

1.2 Descrizione

Il componente riceve in ingresso **ADD** (16 bit) che rappresenta il primo indirizzo valido di memoria da cui iniziare l'elaborazione e **K** (10 bit), il cui valore è compreso tra 0 e 255, rappresentante il numero di parole da processare.

Ogni parola **W** (8 bit) è salvata in memoria ad intervalli di 2 byte a partire da ADD (i.e. ADD, ADD+2, ADD+4, … , ADD+2*(K-1)). Il componente, partendo da ADD, dovrà sostituire le parole il cui valore è 0 (non valide), con l'ultimo valore di W letto diverso da 0. Se non è stata letta alcuna parola W diversa da 0, il componente non modificherà i valori fino a che nella sequenza non sarà letta una parola W valida.

Ad ogni parola è associato un valore di credibilità **C** (8 bit) compreso tra 0 e 31, salvato in memoria all'indirizzo subito successivo alla parola a cui fa riferimento (W in ADD allora C in ADD+1), quindi, anche in questo caso ad intervalli di 2 byte (i.e. ADD+1, ADD+3, ADD+5, … , ADD+2*K-1). Ogni volta che viene letta una nuova parola W valida, il componente deve reinizializzare il valore di credibilità C a 31 e scriverlo all'indirizzo di memoria opportuno. Se la parola W letta non è valida, il valore di credibilità C viene decrementato di 1 rispetto al valore precedente, scrivendolo poi in memoria. Quando il valore di C è 0, non viene decrementato ulteriormente.

1.3 Funzionamento

Gli ingressi principali ad modulo sono **START** (1 bit), **ADD** (16 bit) e **K** (10 bit), mentre l'uscita principale è **DONE** (1 bit). Vi sono poi il segnale di clock **CLK** e reset **RESET** entrambi unici per l'intero sistema. Tutti i precedenti segnali sono sincroni da interpretare sul fronte di salita ad esclusione di reset che è asincrono.

Il modulo, dopo il primo RESET (garantito), pone DONE = 0 e si inizializza, attende un segnale di START = 1, dopo il quale inizierà l'elaborazione (START rimane a 1). Durante questa fase, K e ADD rimarranno stabili e una volta completato il processo, avendo terminato le operazioni su memoria, il componente pone DONE = 1 fino a che START non torna a 0, per poi porre nuovamente DONE = 0, rendendosi pronto per una successiva elaborazione senza attendere un nuovo RESET.

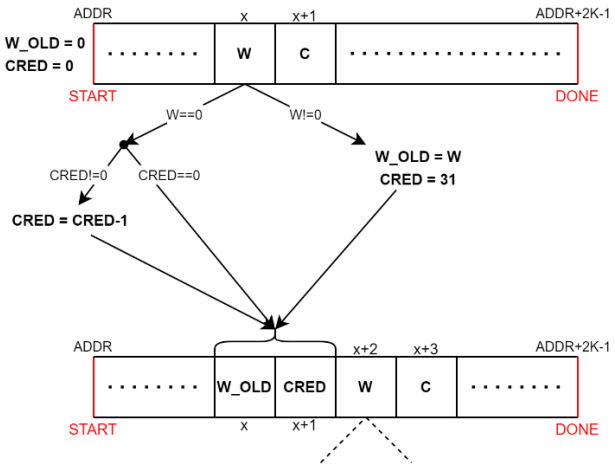

Figura 1: Passo di elaborazione della sequenza di K parole W

Iniziale: (0, 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);

Finale: (0, 0, 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);

1.4 Interfaccia del componente

L'interfaccia che il componete deve rispettare è la seguente:

```
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;
```

2 Architettura

2.1 Scelte progettuali

Per la realizzazione del componente, si è scelto di optare per un approccio modulare, in modo da rendere la struttura finale più manegevole e scalabile. Nella *Figura 2*, si possono osservare i moduli **addr_module**, **counter**, **fsm**, **word_pp_register**, **cred_pp_register** e **cred_module** che rappresentano delle sotto-entità interconnesse a livello più alto. I restanti moduli *controller*, *comparator* e *mux* sono invece più elementari, per questo non sono state create delle entità apposite, ma sono stati integrati nel componente principale. Per l'implementazione del progetto si sono prevalentemente utilizzati *process* e *dataflow* al fine di descrivere in VDHL i sopra citati moduli. La discussione dettagliata è presentata nel capitolo 2.2.

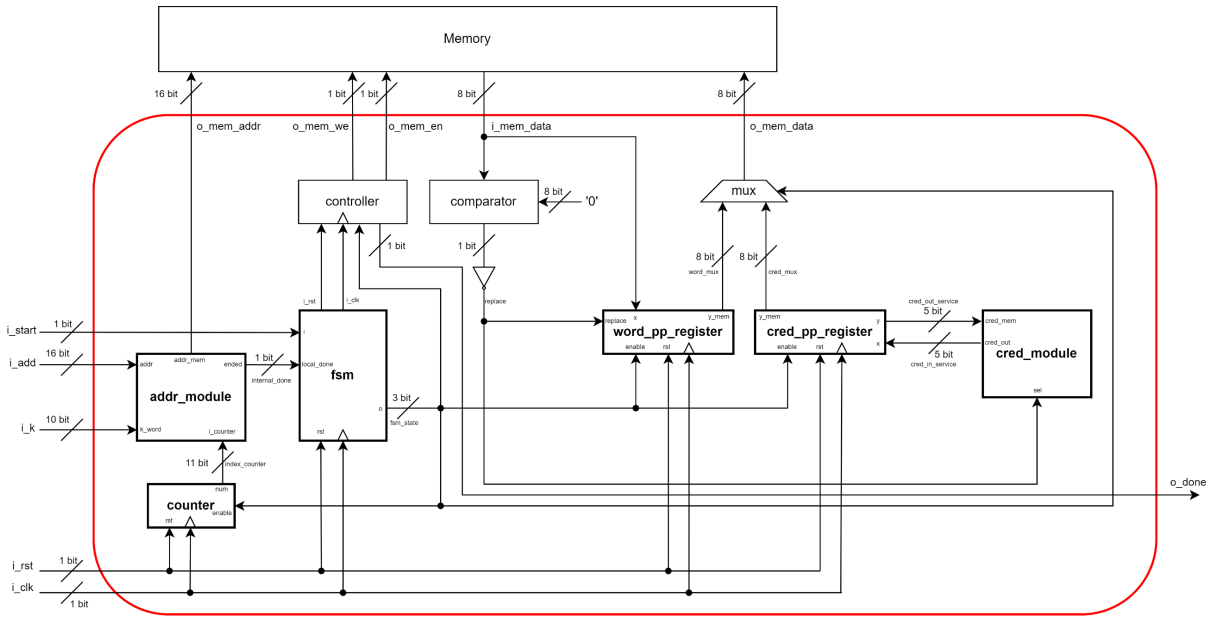

Figura 2: Rappresentazione complessiva di alto livello del componente (delineata in rosso).

2.2 Moduli

Nel seguito si analizzeranno nel dettaglio i moduli illustrati in *Figura 2* da un punto di vista strutturale, funzionale e implementativo. Per i moduli più articolati si farà ricorso ad ulteriori figure dettagliate.

2.2.1 controller

Implementato utilizzando un process, è sincronizzato dal clock e ha lo scopo di abilitare *o_mem_we*, *o_mem_en* e *o_done* in base allo stato corrente della fsm. Inoltre, osservando

il segnale di *reset*, può resettare i segnali precedenti a 0. Non si tratta di una sotto-entità, ma è integrato internamente all'entità principale.

2.2.2 comparator

Implementato in dataflow, ha lo scopo di verificare che il valore *W* letto da memoria non valga 0, se così è, l'output varrà 1 altrimenti 0. Da notare, in cascata al *comparator*, la presenza di una porta NOT che ne inverte il risultato finale. Anche in questo caso si tratta di un modulo integrato nell'entità principale.

2.2.3 mux

Implementato in dataflow, seleziona in base allo stato corrente della fsm l'output corretto, tra *word_mux* e *cred_mux*, da assegnare a o_mem_data per eseguire correttamente la scrittura in memoria. Il modulo è stato integrato nell'entità principale.

2.2.4 addr_module

L'idea di questo modulo è quella di, dato ADD, K e il valore di counter in ingresso, avere direttamente come output l'indirizzo corrente per le operazioni in memoria tramite *addr_mem* e un segnale, *ended*, che faccia da sentinella per la fine elaborazione, una volta raggiunto l'indirizzo ADD + 2*K. Nel modulo si fa uso di uno shifter per raddoppiare K al fine di confrontarlo con l'output di counter attraverso comparator, che restituire 1 se e solo se sono uguali. Il valore di *addr_mem* è calcolato utilizzando un adder direttamente sui segnali di ingresso ADD e *i_counter* (Figura 3). L'intera implementazione è stata realizzata in dataflow, evitando quindi di richiedere cicli di clock aggiuntivi.

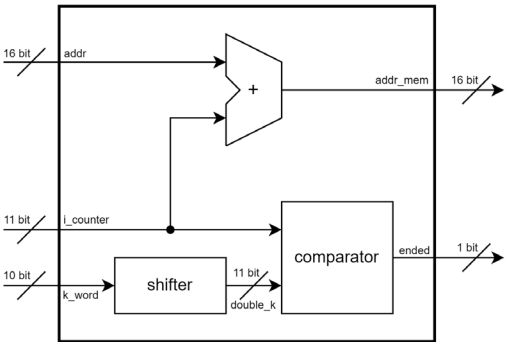

Figura 3: addr_module dettagliato.

2.2.5 counter

Questa sotto-entità è stata implementata quasi completamente da un process ed ha il ruolo di incrementare il suo registro da 0 (ad inizio elaborazione), avanzando di 1, solo per gli stati *GET_DATA* e *WRITE_C* della *fsm*, fino a che l'elaborazione non termina, a seguito della verifica fatta da *addr_module*. Il modulo è sincronizzato dal *clock* e osservando anche il segnale di *reset*, può essere resettato al suo valore massimo (tutti i bit a 1 su 11bit) in modo da iniziare da 0 al primo passo di una nuova elaborazione. Similmente al *reset* che è per sua natura asincrono, si è aggiunto un reset di tipo sincrono attivato dallo stato *REINIT* della *fsm*, al fine di reinizializzare il counter per renderlo pronto ad una nuova elaborazione senza richiedere un *reset* asincrono.

2.2.6 fsm

Questo modulo rappresenta una macchina a stati finiti di *Moore*, che svolge il ruolo di coordinare tutti i moduli tramite l'output associato ad ogni stato. L'implementazione è una collezione di 3 process: *state_reg*, *lambda* e *delta*. Il process *state_reg* è incaricato di salvare lo stato corrente in un registro, *lambda* rappresenta le transizioni tra uno stato e l'altro in base a condizioni diverse e infine *delta* descrive l'output associato ad ogni stato. Si

tratta di una sotto-entità sincronizzata e resettabile tramite *reset*, in qualunque stato essa si trovi. Ulteriori dettagli sono riportati nel capitolo 2.3.

2.2.7 word_pp_register

Questa sotto-entità vuole rappresentare un registro parallelo per memorizzare l'ultima parola W valida letta nell'elaborazione corrente. La sua implementazione è stata realizzata quasi interamente da un unico process, trattandosi di un componente sincrono e resettabile, tramite *reset*, al valore 0. Disponendo anche di un reset sincrono, può essere reinizializzato anche dallo stato *REINIT* della fsm, per ulteriori sequenze da processare. La memorizzazione della parola letta da memoria, avviene solo nel caso in cui lo stato corrente della fsm è *WRITE_W* e l'esito del *comparator*, a seguito della porta NOT, è uguale a 1, cioè, se e solo se la parola letta da memoria non è 0.

2.2.8 cred_pp_register

Anche in questo caso si tratta di un registro parallelo, ma utilizzato per memorizzare il valore di credibilità C associato alla parola W da scrivere in memoria. Implementato quasi interamente da un unico process, è un modulo sincronizzato, resettabile dal segnale di *reset* al valore 0 e può essere reinizializzato da un reset sincrono quando lo stato corrente della fsm è *REINIT.* Il valore della credibilità C è memorizzato ogni volta che la fsm si trova nello stato *WRITE_W*, poiché è il modulo *cred_module* ad occuparsi della corretta computazione del valore C.

2.2.9 cred_module

L'idea di questo modulo è avere il corretto valore della credibilità C, associato alla prossima parola W da scrivere in memoria, a partire dal valore precedente di C e dal risultato ottenuto dal *comparator* negato, ottendendo come output il valore C da memorizzare in *cred_pp_register* per una successiva scrittura su memoria. L'implementazione è stata completamente realizzata in dataflow per evitare di richiedere cicli di clock aggiuntivi ed è composta da 2 mux, 1 adder e 1 comparator (Figura 4). Un mux si occupa di selezionare il risultato C finale tra il mux precedente o il valore 31, in base al valore di *sel* che indica se l'ultimo valore W letto è un valore valido. Il mux più interno seleziona invece 0 se il valore di C precedente era 0 o il risultato di adder, che decrementa di 1 il precedente valore di C.

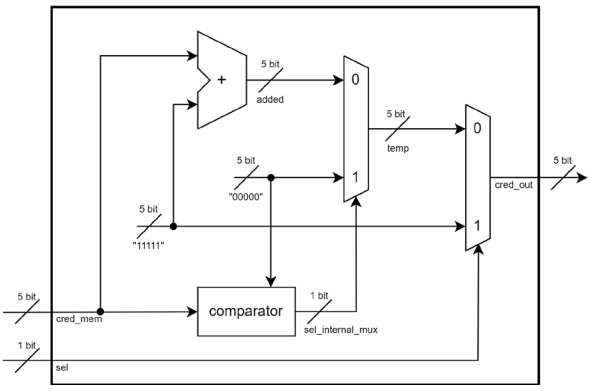

Figura 4: cred_module dettagliato.

2.3 FSM

La fsm utilizzata nel progetto è una macchina di Moore. Nel seguito si analizzeranno gli stati che la compongono.

IDLE	000	Stato in cui si trova la fsm a seguito di un RESET. Il componente è in attesa del segnale START = 1.

GET_DATA	001	Attiva il modulo *counter* e di conseguenza, il modulo *addr_module* calcolerà il prossimo indirizzo di memoria dal quale effettuare la lettura di W, oltre a calcolare se si è raggiunto il termine della sequenza di parole. Anche il modulo *controller* viene attivato, che si occuperà di abilitare la memoria in lettura.
WAIT_CHECK	010	Ha la duplice funzione di controllare il valore di fine sequenza calcolato in precedenza (local_done = 1 allora il prossimo stato sarà REINIT) o attendere semplicemente la lettura da memoria di W.
WRITE_W	011	Attiva il modulo *controller* che a sua volta abilita la memoria in scrittura. Attiva il modulo *word_pp_register* che memorizzerà il nuovo valore di W se previsto dal *comparator* negato.
WRITE_C	100	Attiva il modulo controller che a sua volta abilita la memoria in scrittura. Attiva il modulo *cred_pp_register* che memorizzerà il nuovo valore di C calcolato da *cred_module*. Seleziona il *mux* in modo tale che su *o_mem_data* ci sia la parola W dello stato precedente per l'effettiva scrittura (negli altri stati il *mux* pone la credibilità C su *o_mem_data*).
REINIT	101	Tramite il controller, viene posto o_done = 1 e i moduli *counter*, *word_pp_register* e *cred_pp_register* vengono reinizializzati. Si rimane in questo stato finchè START non ritorna a 0.

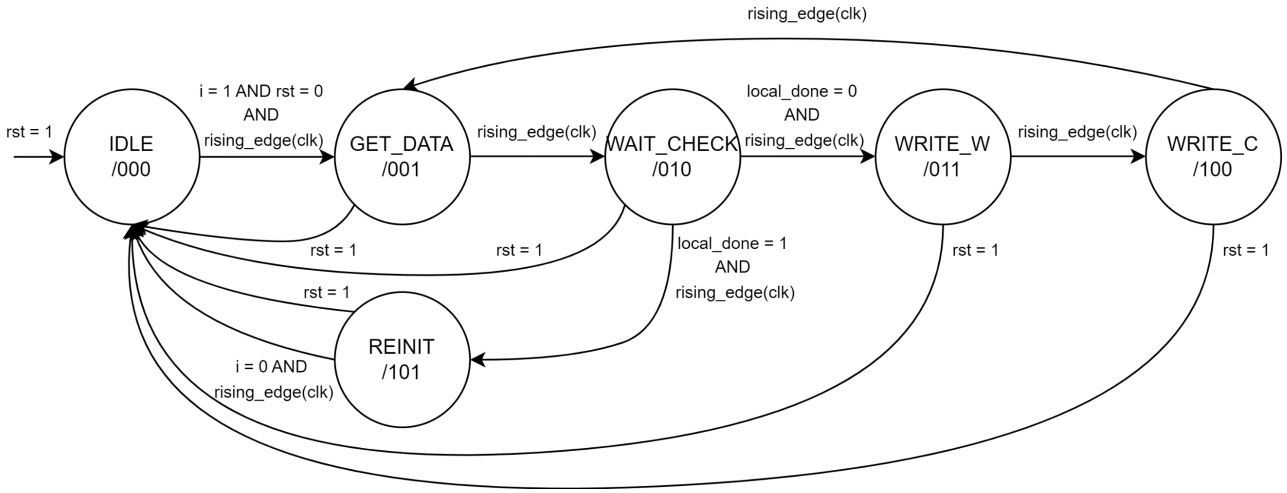

Figura 5: Rappresentazione grafica della macchina a stati finiti utilizzata nel progetto.

3 Risultati sperimentali

3.1 Sintesi

Il componete soddisfa i requisiti ed è correttamente sintetizzabile superando le varie simulazioni sia in *Behavioral* che *Post-Synthesis Functional*. Durante la fase di progettazione, si è prestata particolare attenzione al fine di evitare la generazione di latch, e il report di sintesi (Figura 6) ne conferma chiaramente il risultato.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	53	0	0	134600	0.04
LUT as Logic	53	0	0	134600	0.04
LUT as Memory	0	0	0	46200	0.00
Slice Registers	33	0	0	269200	0.01
Register as Flip Flop	33	0	0	269200	0.01
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

Figura 6: report_utilization

Anche il requisito di tempo di 20 ns di clock è stato rispettato, riportando uno slack di 16.498 ns.

3.2 Simulazioni

Nel seguito si riporteranno nel dettaglio i test benches più significativi. Non sono i soli ad essere stati eseguiti, infatti sono stati valutati molti altri test, creati dalla combinazione di quelli riportati, ma anche test generati casualmente tramite uno script Python per verificare la robustezza del componente.

3.2.1 Test in dotazione

Il primo test ad essere stato eseguito è *project_tb.vhd,* fornito insieme alla specifica. Esso verifica il corretto funzionamento del componente per un caso di base e consiste di una sola sequenza da processare.

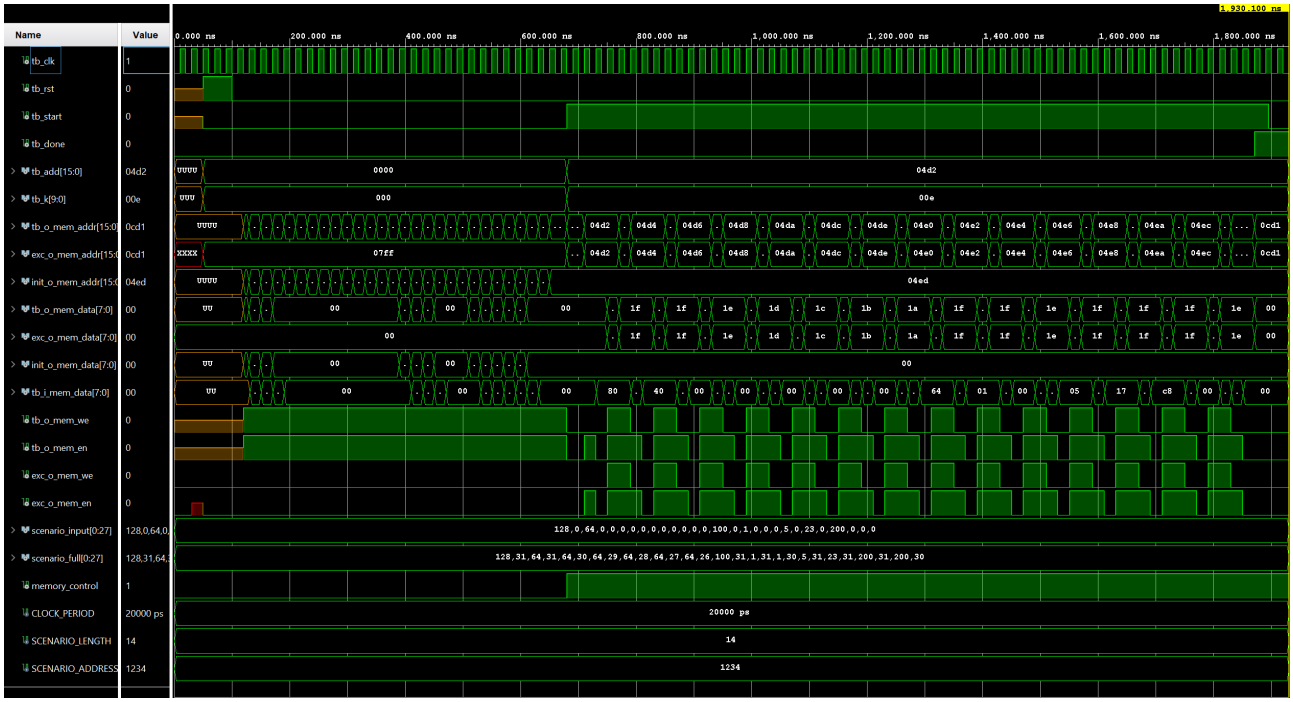

Figura 7: Post-Synthesis Functional Simulation di Test in dotazione

3.2.2 Test_1

Questo test verifica se il componente valuta correttamente più sequenze in successione e non solo, infatti le 3 sequenze sono:

- K = 0, cioè lunghezza nulla
- sequenza generica
- sequenza in cui la prima parola W vale 0

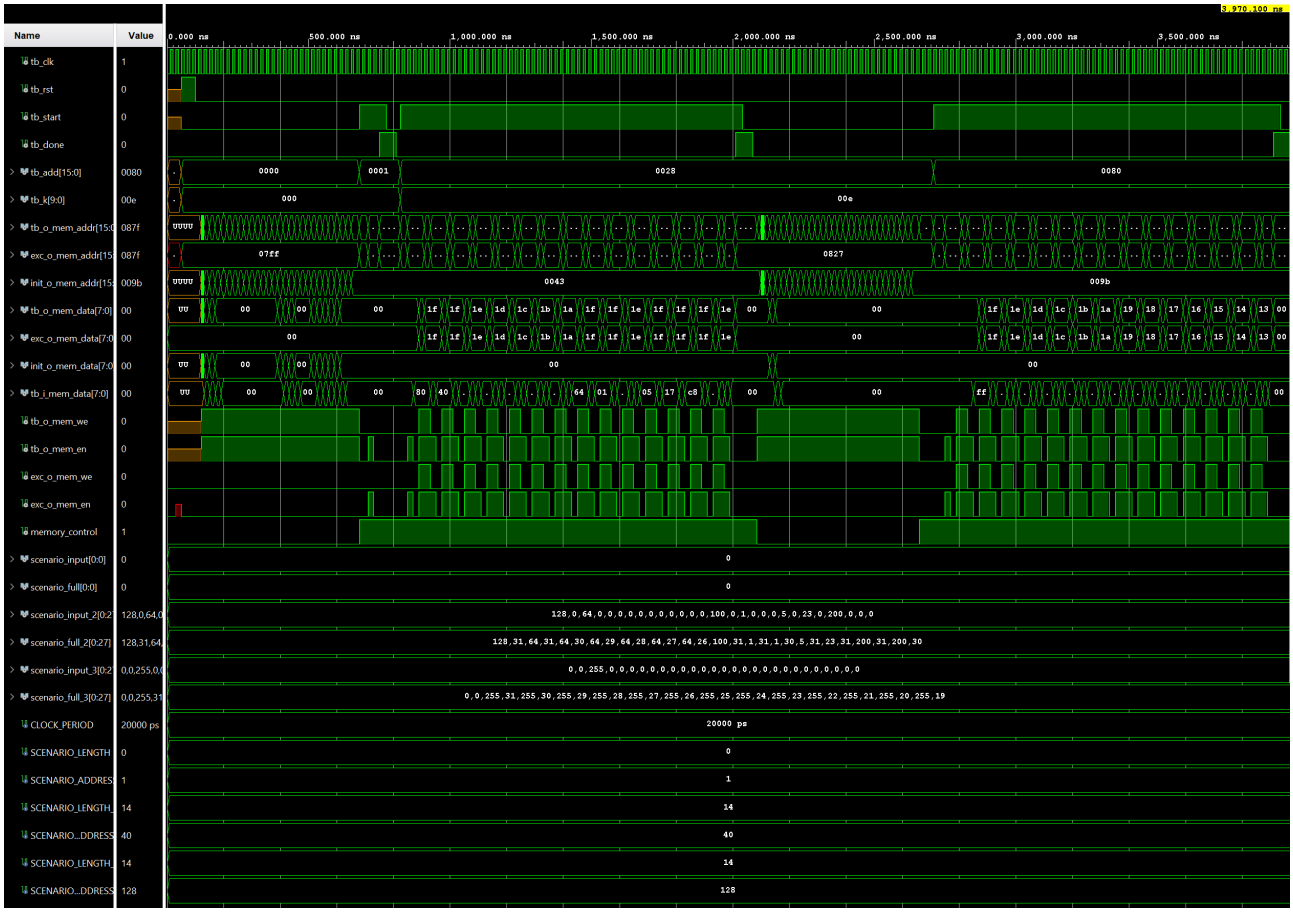

Figura 8: Post-Synthesis Functional Simulation di Test_1

3.2.3 Test_2

Questo test verifica per ogni stato della *fsm* se il RESET asincrono funziona correttamente. Il test avvia il componente per 6 volte, partendo la prima volta con un reset durante lo stato *GET_DATA* fino al penultimo avvio in cui il reset è fatto in *REINIT*. Infine, all'ultimo avvio viene eseguita un'intera elaborazione.

Figura 9: Post-Synthesis Functional Simulation di Test_2

3.2.4 Test_3

Questo test verifica 3 sequenze particolari. Nella prima si ha una sequenza di soli zeri, nella seconda ad ogni passo si ha un valore valido di W e nella terza, la credibilità C viene decrementata fino a 0, rimanendo tale per qualche passo, per poi ri-avere in lettura parole valide.

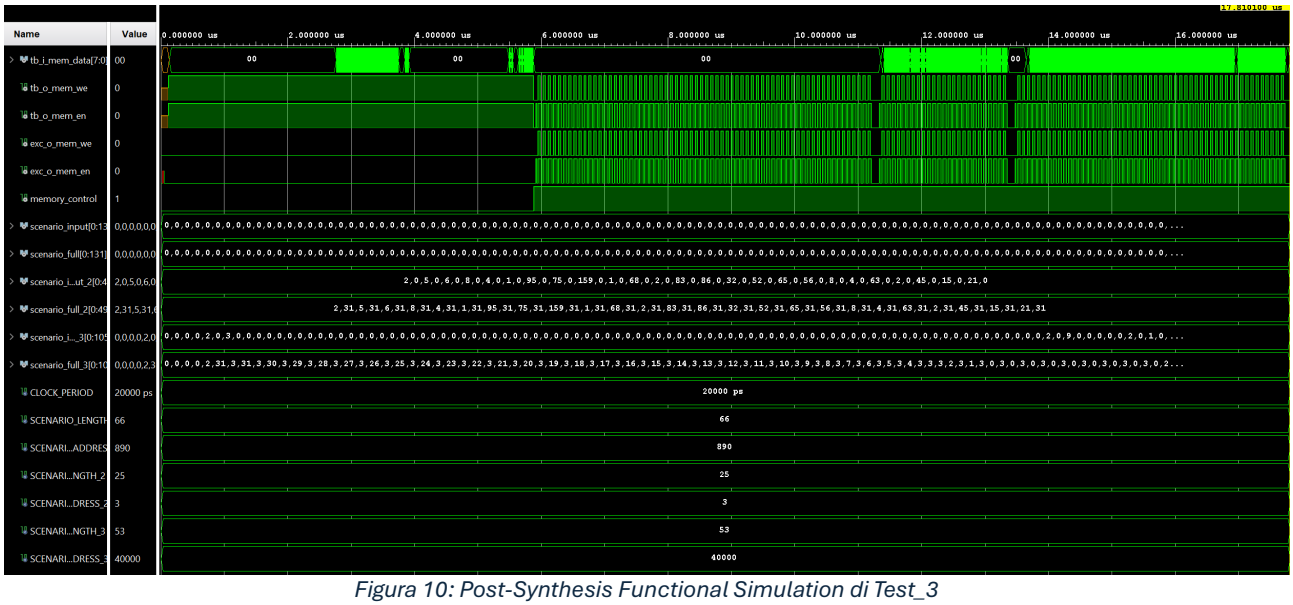

Figura 10: Post-Synthesis Functional Simulation di Test_3

3.2.5 Test_4

Questo test rappresenta un caso particolare, in cui durante RESET = 1 si ha START = 1, prima che RESET torni a 0. Il componente in questo caso inizierà l'elaborazione solo dopo che RESET è tornato a 0, se START vale ancora 1.

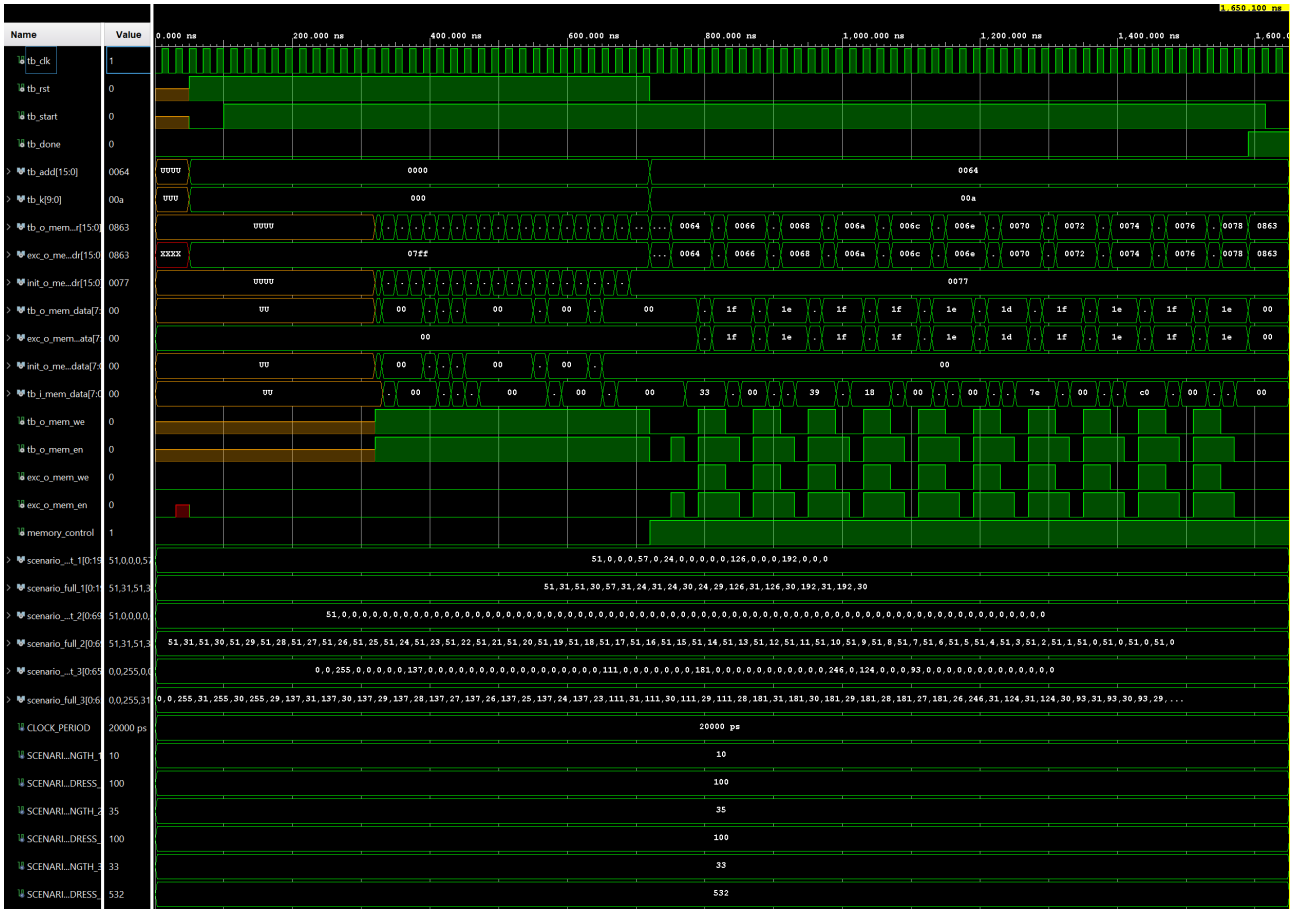

Figura 11: Post-Synthesis Functional Simulation di Test_4

4 Conclusioni

A fronte dei risultati ottenuti da tutti i test, passati sia in *Behavioral* che in *Post-Synthesis Functional*, il componente progettato soddisfa tutti i requisiti della specifica, inoltre è possibile elaborare con successo sequenze di lunghezza compresa tra 0 e 1023 (quindi oltre la richiesta di 255).

In fase di progettazione si era pensata anche ad un'implementazione alternativa a quella attuale, fatta da soli *process*, ma questa richiedeva ulteriori stati della fsm, impattando quindi sul tempo di esecuzione complessivo, come conseguenza di un aumento del numero di clock necessari per l'elaborazione di una singola parola W. Questo, come la volontà di ottimizzare il numero di operazioni per singolo clock, mi ha portato a scegliere una progettazione mista, facendo ricorso a *process* e *dataflow* nell'implementazione finale, utilizzando principalmente *process* per moduli sequenziali e *dataflow* per quelli combinatori.