RELAZIONE PROVA FINALE DI RETI LOGICHE

DOCENTE: Fornaciari William

POLITECNICO
MILANO 1863

STEFANO BERNARDOTTO - Codice Persona: 10853915

MARTA SILVIA BERNARDIS - Codice Persona: 10841045

1. Introduzione

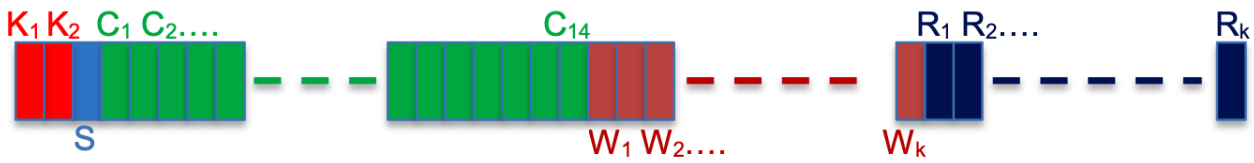
1.1 Specifiche Generali

1.1.1 Descrizione

Il progetto consiste nella realizzazione in **linguaggio VHDL** un modulo hardware che sia in grado di leggere una sequenza di dati dalla memoria, applicarne un filtro digitale ed infine scrivere i risultati nuovamente in memoria.

La sequenza in ingresso è organizzata come segue:

- **K1, K2**: sono i primi due byte della sequenza che rappresentano la lunghezza della sequenza di dati da elaborare
- **S**: è il terzo byte che indica il filtro da applicare durante la fase di computazione.
- **C1-C14**: I seguenti 14 byte rappresentano i coefficienti dei filtri
- **K byte**: ogni byte è un numero della sequenza che dovrà essere elaborato



Ogni **filtro** segue la seguente funzione:

$$f'(i) = \frac{1}{n} * \sum_{j=-l}^{+l} C_j * f(i+j)$$

Dove:

- **n** è il valore di normalizzazione
- **C_j** è il coefficiente j-esimo
- **l** è il numero di coefficienti che vengono considerati

Se il bit meno significativo del byte S ha valore "0", verrà applicato il filtro di **ordine 3**, il quale prevede:

- Utilizzo dei primi 7 coefficienti (C1-C7)
- Valore di normalizzazione pari a 12
- Il valore l considerato è 2, cioè vengono utilizzati 5 valori della sequenza: quello corrente, i due precedenti e i due successivi

Se il bit meno significativo del byte S ha, invece, valore "1", sarà applicato il filtro di **ordine 5**, il quale prevede:

- Utilizzo degli ultimi 7 coefficienti (C8-C14)
- Valore di normalizzazione pari a 60
- Il valore l considerato è 3, cioè vengono utilizzati 7 valori della sequenza: quello corrente, i tre precedenti e i tre successivi

1.1.2 Funzionamento

Il funzionamento del modulo prevede **tre ingressi primari**:

- **START**: un segnale a 1 bit, nel momento in cui viene portato a un valore alto, permette di iniziare l'elaborazione
- **ADD**: un segnale a 16 bit, che rappresenta l'indirizzo di memoria da cui iniziare la lettura della sequenza in ingresso.
  Dunque, la sequenza su cui svolgere i calcoli è situata all'indirizzo ADD+17 e l'output del modulo verrà poi scritto in memoria a partire dall'indirizzo ADD+K+17
- **DONE**: un segnale a 1 bit, verrà portato a un valore alto al termine dell'elaborazione. In particolare, non è possibile portare alto oil valore di START finché il segnale DONE non avrà indicato il termine dell'elaborazione.

Inoltre, vengono considerati anche i seguenti segnali, unici per il sistema:

- **CLK**: è il segnale di clock
- **RESET**: un segnale asincrono che, se posto a valore alto, fa re-inizializzare il sistema. All'inizio dell'elaborazione, è necessario porre alto il segnale di RESET per far sì che la macchina attenda l'attivazione del segnale di START

1.2 Strumenti Forniti

Tra gli strumenti forniti per l'implementazione del modulo si evidenziano:

- **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);

        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;
```

I cui segnali sono:

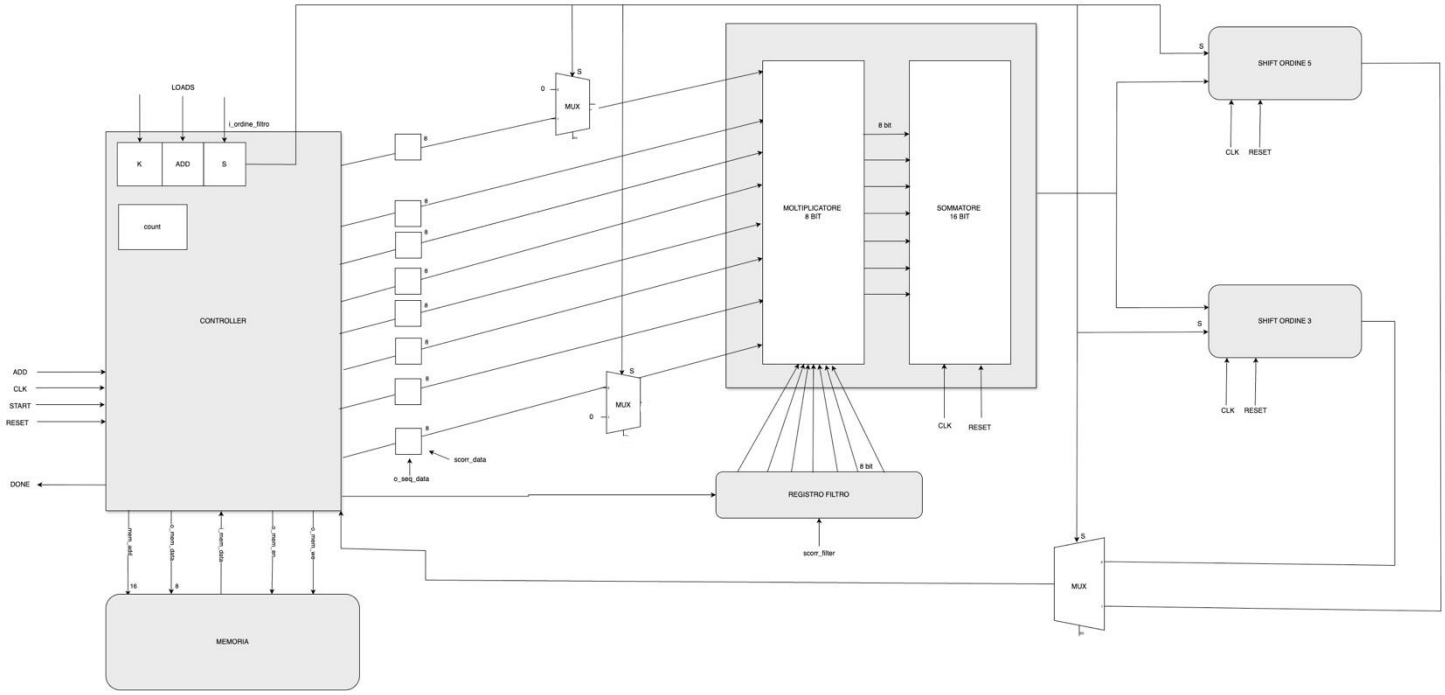
- **i_clk**: rappresenta il segnale di CLOCK generato dal Test Bench
- **i_rst**: rappresenta il segnale di RESET asincrono
- **i_start**: rappresenta il segnale di START generato dal Test Bench
- **i_add**: rappresenta il segnale (vettore) ADD generato dal Test Bench
- **o_done**: rappresenta il segnale di DONE di uscita
- **o_mem_addr**: rappresenta il segnale (vettore) che manda l'indirizzo alla memoria
- **i_mem_data**: rappresenta il segnale (vettore) che arriva dalla memoria e contiene il dato in seguito a una richiesta di lettura
- **o_mem_data**: rappresenta il segnale (vettore) che va verso la memoria e contiene il dato che dovrà essere scritto
- **o_mem_en**: rappresenta il segnale di ENABLE da mandare alla memoria per permettere comunicazione sia in lettura che in scrittura
- **o_mem_we**: rappresenta il segnale di WRITE ENABLE da mandare alla memoria che permette di scriverci. Se vale "1", la memoria verrà scritta, invece se vale "0" la memoria verrà letta

- **INTERFACCIA DI MEMORIA – inizializzata nel Test Bench**

```
-- 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;
```

2. Descrizione dell'Architettura

2.1 Descrizione ad alto livello



Il modulo è composto dai seguenti sotto-moduli, ognuno con una specifica funzione necessaria al corretto funzionamento dell'intero modulo.

In particolare, questi sotto-moduli hanno in comune il medesimo segnale CLK e lo stesso segnale di RESET, utile per resettare i singoli sotto-moduli.

2.1 Componenti Utilizzati

2.1.1 Controller

È il modulo principale che si occupa dell'implementazione vera e propria della macchina a stati del componente. In particolare, nel momento dell'attivazione del segnale di START, acquisisce negli stati iniziali i segnali K, S e ADD per poi utilizzare gli altri registri per svolgere i calcoli sulla sequenza in ingresso ed infine ottenere la sequenza di output, azionando infine il segnale DONE.

Inoltre, è il modulo che interagisce direttamente con la memoria, azionando i segnali *mem_add* (16 bit), *o_mem_data* (8 bit), *o_mem_en*, *o_mem_we* e ricevendo il segnale *i_mem_data*.

La sequenza letta viene inviata al modulo dedicato alle moltiplicazioni e alle somme, attraversando una prima fase in cui vengono considerati solo i valori su cui svolgere la prima operazione, riempiendo le posizioni iniziali. Vengono applicati due multiplexer, uno al primo valore e l'altro all'ultimo, che hanno come selettore il valore del filtro S. Se viene scelto il filtro 3, il modulo considererà 5 valori inserendo un valore nullo in corrispondenza dei multiplexer. Se invece il filtro è 5, il modulo considererà tutti i 7 valori in ingresso.

Una volta letti questi valori, grazie ai segnali ***o_seq_data*** e ***scorr_data***, i valori scaleranno di posizione fino al termine della sequenza.

2.1.2 Registro Filtro

È un registro che memorizza i valori che il componente controller legge in ingresso per quanto riguarda i coefficienti del filtro.

In particolare, nel momento di svolgere i calcoli, vengono letti in sequenza grazie al segnale ***scorr_filter*** che permette di selezionare il coefficiente corretto.

2.1.3 Moltiplicatore

Utilizzando i valori presenti nel registro filtro, svolge le moltiplicazioni (ad 8 bit), tra il valore della sequenza che viene considerato ed un valore del filtro.

2.1.4 Sommatore

I valori in uscita dal moltiplicatore vengono poi sommati in questo sotto-modulo. Il segnale in uscita considera 17 bit per considerare un possibile over-flow.

In un primo momento, somma il primo valore in ingresso col secondo, il terzo con il quarto, il quinto con il sesto e non considera il settimo.
In seguito, il risultato della somma del primo con il secondo viene sommato con il risultato del terzo con il quarto; ed il risultato del quinto con il sesto viene sommato al settimo valore in ingresso.
Infine, quest'ultimi valori vengono sommati producendo il risultato che sarà normalizzato prima della scrittura della sequenza di output nella posizione dove era posizionato il valore centrale in ingresso.

2.1.5 Shifter

I due registri, hanno la funzione di applicare la normalizzazione, ovvero la divisione dell'output del registro Sommatore per un valore che dipende dalla scelta del filtro. Siccome in VHDL la divisione è un'operazione costosa e poco sintetizzabile, si è preferito approssimare l'operazione tramite somma dei valori ottenuti da uno shift logico verso destra, equivalente alla divisione per potenze di due.

Nel caso di filtro di ordine 3, il risultato ottenuto dalle operazioni precedenti deve essere diviso per 12. Tale operazione equivale ad effettuare le seguenti somme:

$$\frac{1}{12} \approx \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \frac{1}{1024}$$

Cioè:

$$val\ srl\ 12 = (val\ srl\ 4) + (val\ srl\ 6) + (val\ srl\ 8) + (val\ srl\ 10)$$

Nel caso, invece di filtro di ordine 5, il risultato precedente deve essere diviso per 60. Tale operazione equivale ad effettuare le seguenti somme:

$$\frac{1}{60} \approx \frac{1}{64} + \frac{1}{1024}$$

Cioè:

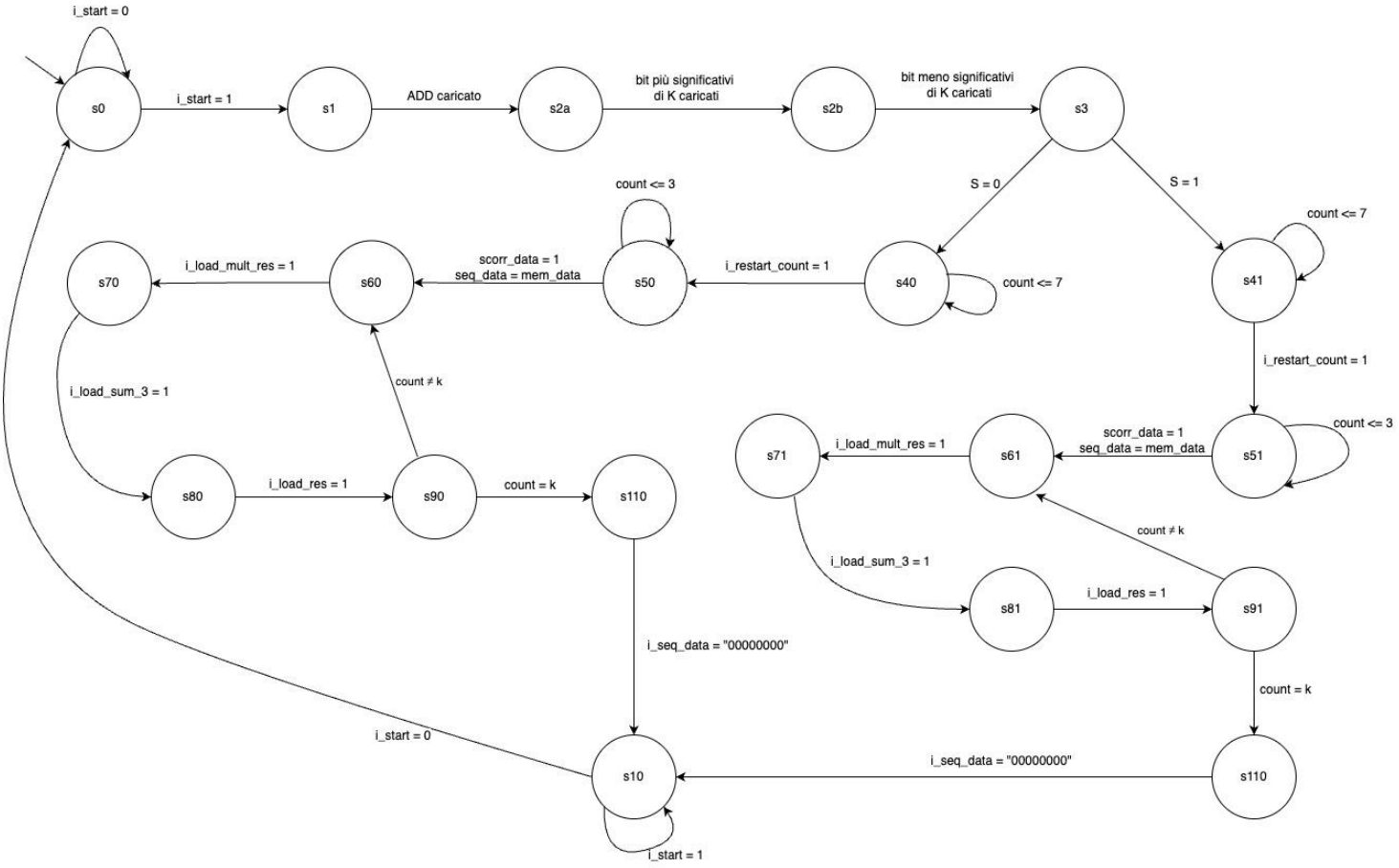
$$val\ srl\ 60 = (val\ srl\ 6) + (val\ srl\ 10)$$

L'unico accorgimento a cui porre attenzione è che in caso di numeri negativi, lo shift logico non conserva il bit di segno, introducendo un errore sistematico che potrebbe falsare il risultato.

Tale bias è facilmente rimediabile ricordando di aggiungere 1 al termine dell'operazione di shift logico, in modo tale da compensare l'errore tramite approssimazione empirica.

2.2 Macchina a Stati e Descrizione degli Stati

La seguente macchina a stati rappresenta ciò che avviene all'interno del controller. Ad ogni stato è stato posto un codice (***o_state***) che è stato utilizzato in fase di debugging per comprendere a pieno in quale stato e per quale motivo si verificassero eventuali errori.



In particolare, ogni stato a partire dal quarto, nel codice VHDL, è stato suddiviso in **sottostati** per far in modo che lettura del valore, svolgimento delle operazioni e caricamento del valore finale avvengano correttamente in cicli di clock diversi.

Gli stati dal quarto al nono svolgono le stesse funzioni, sono stati suddivisi per gestire in modo funzionale l'elaborazione, differenziando i calcoli in base all'ordine del filtro scelto.

Sono in seguito spiegati dettagliatamente tutti gli stati:

STATO	SEGNALI FONDAMENTALI	FUNZIONE
S0	-	Si tratta dello stato di RESET, il quale ha il compito di inizializzare tutti i segnali
S1	add <= i_add	Legge il valore ADD
S2a	o_mem_en <= '1' k (15 downto 8) <= i_mem_data	Legge gli 8 bit più significativi del valore K, corrispondente alla lunghezza della sequenza da considerare
S2b	o_mem_en <= '1' k (7 downto 0) <= i_mem_data	Legge gli 8 bit meno significativi del valore K, corrispondente alla lunghezza della sequenza da considerare
S3	i_ordine_filtro <= i_mem_data(0)	Legge il valore S che determina l'ordine del filtro: $0 \rightarrow 3$; $1 \rightarrow 5$
S40, S41	o_mem_addr <= std_logic_vector(unsigned(add) + 3 + unsigned(o_count)) i_inc_count <= '1'	Carica nel registro filtro la sequenza dei coefficienti del filtro da utilizzare. Inoltre, imposta la memoria e inizializza il contatore per il prossimo ciclo
S50, S51	i_seq_data <= i_mem_data i_scorr_data <= '1'	Posiziona i primi quattro valori della sequenza in ingresso nel registro *seq* per poi eseguire le moltiplicazioni
S60, S61	i_load_mult_res <= '1'	È lo stato in cui vengono eseguite le moltiplicazioni
S70, S71	i_load_sum1 <= '1' i_load_sum2 <= '1' i_load_sum3 <= '1'	È lo stato in cui vengono eseguite le somme

S80, S81	i_load_shift_res0 <= '1' i_load_shift_res1 <= '1' i_load_shift_res2 <= '1' i_load_res <= '1' i_seq_data <= i_mem_data;	È lo stato che esegue la normalizzazione tramite shift ed inoltre carica il prossimo valore nella sequenza. *o_val_data* è il risultato effettivo della computazione
S90, S91	o_mem_we <= '1' o_mem_en <= '1' o_mem_data <= o_val_data i_inc_count <= '0' o_done <= '1'	È lo stato che ha lo scopo di scrivere il risultato finale in memoria. Inoltre, questo stato gestisce il valore di *count*
S110, S111	i_seq_data <= "00000000"; i_inc_count_aux <= '1';	Una volta finita la sequenza nello stato precedente, carica una sequenza in ingresso interamente nulla in memoria, facendo eseguire il ciclo per tre volte grazie al segnale *i_inc_count_aux*
S10	o_done <= '1'	È lo stato finale che porrà ad “1” il valore di DONE, preparando l’architettura alla prossima elaborazione nel momento in cui verrà ricevuto un nuovo segnale di START

In seguito, vengono riportati tutti gli stati (con i relativi sottostati) e la loro **codifica**, utilizzata in fase di debugging per comprendere quali fossero le linee di codice in cui il progetto presentava errori; in modo tale di agire in maniera mirata per risolverli.

State	New Encoding	Previous Encoding
s0	000000	000000
s0a	000001	000001
s1	000010	000010
s2a	000011	000011
s2a1	000100	000100
s2a2	000101	000101
s2b	000110	000110
s2b1	000111	000111
s2b2	001000	001000
s3	001001	001001
s3a	001010	001010
s3b	001011	001011
s40	001100	001100
s40a	001101	001101
s40b	001110	001110
s40c	001111	001111
s50	010000	010100
s50a	010001	010101
s50b	010010	010110
s50c	010011	010111
s60	010100	011100
s70	010101	011110
s70a	010110	011111
s70b	010111	100000
s80a	011000	100100
s80b	011001	100101
s80c	011010	100110
s80d	011011	100111
s90	011100	101100
s90a	011101	101101
s90b	011110	101110
s90c	011111	101111
s110	100000	111001
s90d	100001	110000
s90e	100010	110001
s41	100011	010000
s41a	100100	010001
s41b	100101	010010
s41c	100110	010011
s51	100111	011000
s51a	101000	011001
s51b	101001	011010
s51c	101010	011011
s61	101011	011101
s71	101100	100001
s71a	101101	100010
s71b	101110	100011
s81a	101111	101000
s81b	110000	101001
s81c	110001	101010
s81d	110010	101011
s91	110011	110010
s91a	110100	110011
s91b	110101	110100
s91c	110110	110101
s111	110111	111010
s10	111000	111000
s91d	111001	110110
s91e	111010	110111

3. Risultati Sperimentali

3.1 Report di Sintesi

Il progetto è stato testato sullo strumento di sintesi **XILINX VIVADO WEBPACK**, utilizzando come **FPGA xc7k70tfbv676-1**.

Il codice VHDL è correttamente sintetizzabile senza generare alcun errore.

Sono in seguito riportati i componenti utilizzati sul design sintetizzato.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	991	0	41000	2.42
LUT as Logic	991	0	41000	2.42
LUT as Memory	0	0	13400	0.00
Slice Registers	561	0	82000	0.68
Register as Flip Flop	471	0	82000	0.57
Register as Latch	90	0	82000	0.11
F7 Muxes	0	0	20500	0.00
F8 Muxes	0	0	10250	0.00

Il modello sintetizzato utilizza le **LUT** (*lookup table*) che hanno lo scopo di implementare la logica combinatoria del progetto.

Sono inoltre presenti un numero considerevole di **flip-flop**, utilizzati per salvare lo stato dei segnali e per poterlo aggiornare ad ogni passaggio di esecuzione.

Il numero di **latch** utilizzati è piuttosto basso in modo tale di evitare di memorizzare segnali con valori errati. Nonostante ciò, in alcuni casi sono risultati necessari per il corretto sviluppo del modello; in particolare sono stati utilizzati per salvare i segnali K, ADD e S.

3.2 Simulazioni

La fase di simulazione e testing è una delle fasi più cruciali del progetto. In particolare, è fondamentale per verificare il corretto comportamento del componente in relazione a stimoli differenti e considerando casi limite e casi particolari.

Il componente è risultato correttamente sintetizzabile e simulabile in **pre-sintesi** e in **post sintesi comportamentale**. Esso è infatti stato testato con tutti i test bench seguenti in:

- *Behavioral simulation*,
- *Post-synthesis Functional simulation*
- *Post-implementation Functional simulation*

Il primo test bench utilizzato è stato quello fornito dal docente, il quale è risultato cruciale nelle prime fasi di sviluppo del progetto per perfezionare il comportamento di base del componente.

In seguito, riportiamo i risultati ottenuti testando il componente con alcuni casi limite, utilizzati per osservarne la completezza e la robustezza.

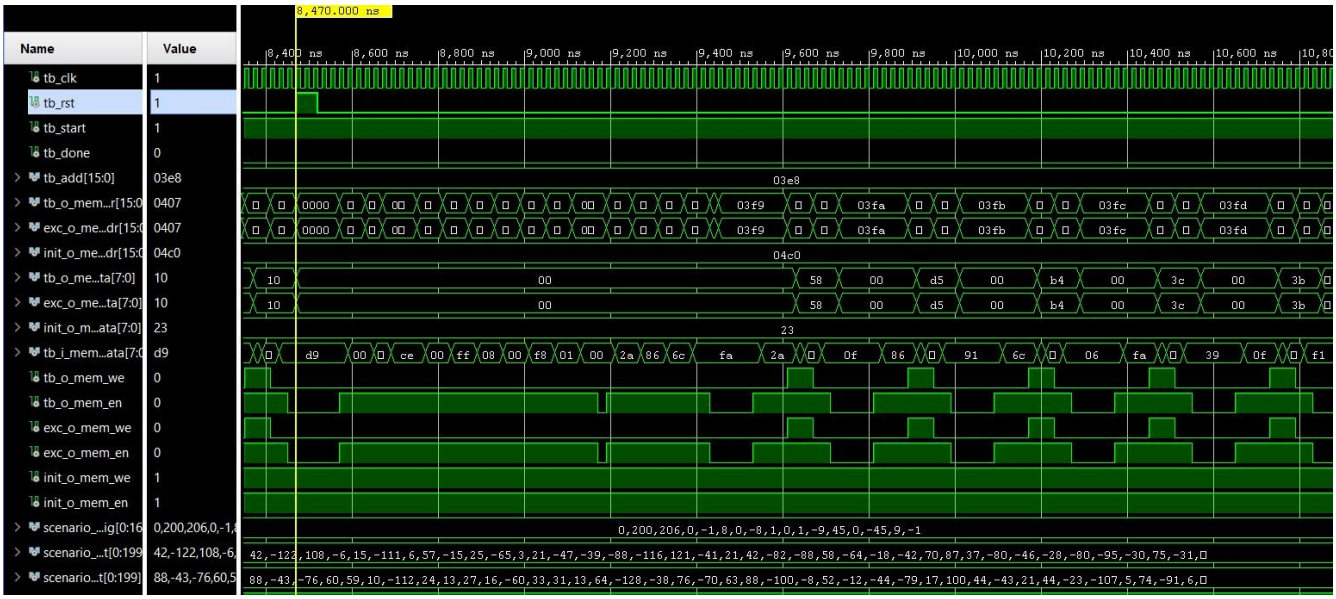
3.2.1 Primo Test Bench

Il primo test bench che riportiamo verifica che il componente si comporti come desiderato nel caso venga attivato il segnale di reset asincrono.

Si osserva che il componente è in grado di ricominciare l'esecuzione dopo l'inserimento del reset, reimpostando l'indirizzo iniziale, leggendo i dati di configurazione ed infine ricominciando l'elaborazione.

```
tb_start <= '1';

for i in 0 to 200 loop
    wait until rising_edge(tb_clk);
end loop;

tb_rst <= '1';

wait for 50 ns;

tb_rst <= '0';

while tb_done /= '1' loop
    wait until rising_edge(tb_clk);
end loop;

wait for 5 ns;

tb_start <= '0';
```

Si può notare nell'esecuzione del test bench come il segnale di reset faccia in modo di re-inizializzare i segnali per poi procedere con l'elaborazione.



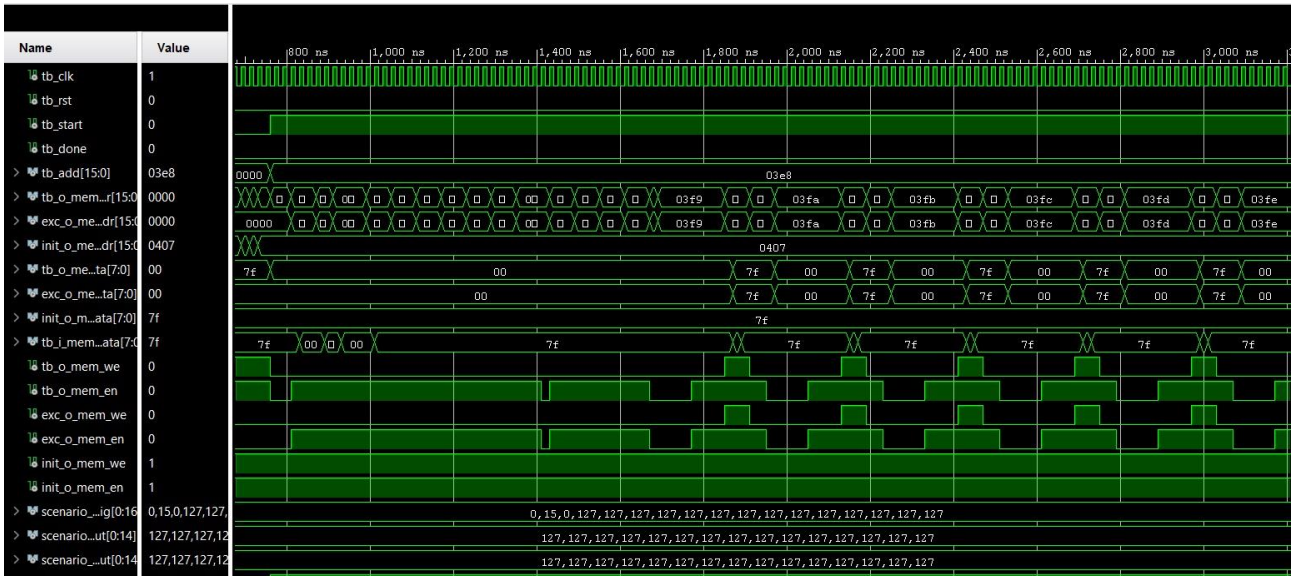
3.2.2 Secondo Test Bench

Questo secondo test bench verifica che il componente sia resistente in caso di saturazione dei valori al massimo valore positivo rappresentabile su 8 bit, cioè 127.

Si può osservare che la sequenza di valori in memoria è composta da soli valori $7F_{16}$ che corrisponde al valore 127_{10}.

```
signal scenario_config : scenario_config_type := (to_integer(unsigned(SCENARIO_LENGTH_STL(15 downto 8))),    -- K1
                                                  to_integer(unsigned(SCENARIO_LENGTH_STL(7 downto 0))),     -- K2
                                                  0,                                                         -- S
                                                  127,127,127,127,127,127,127,127,127,127,127,127,127,127    -- C1-C14
                                                  );
signal scenario_input : scenario_type := (127,127,127,127,127,127,127,127,127,127,127,127,127,127,127);
signal scenario_output : scenario_type := (127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127);
```

È stato verificato che funzioni anche il comportamento duale negativo, ovvero con valori strettamente minori di -128. Viene qui riportato solo la schermata di esecuzione dei valori positivi.

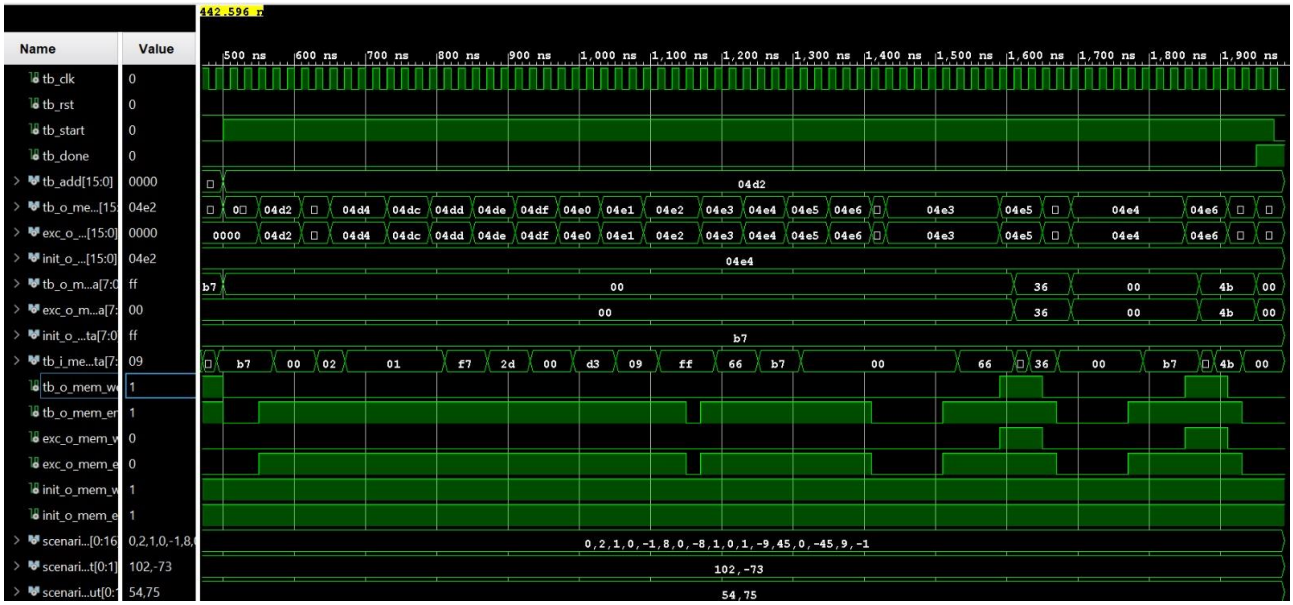


3.2.3 Terzo Test Bench

Il terzo test bench che riportiamo verifica il corretto funzionamento in caso di una micro-sequenza formata da soli due valori.

```
-- Scenario
type scenario_config_type is array (0 to 16) of integer;
constant SCENARIO_LENGTH : integer := 2;
constant SCENARIO_LENGTH_STL : std_logic_vector(15 downto 0) := std_logic_vector(to_unsigned(SCENARIO_LENGTH, 16));
type scenario_type is array (0 to SCENARIO_LENGTH-1) of integer;

signal scenario_config : scenario_config_type := (to_integer(unsigned(SCENARIO_LENGTH_STL(15 downto 8))),    -- K1
                                                  to_integer(unsigned(SCENARIO_LENGTH_STL(7 downto 0))),     -- K2
                                                  1,                                                         -- S
                                                  0, -1, 8, 0, -8, 1, 0, 1, -9, 45, 0, -45, 9, -1            -- C1-C14
                                                  );
signal scenario_input : scenario_type := (102, -73);
signal scenario_output : scenario_type :=(54, 75);
```

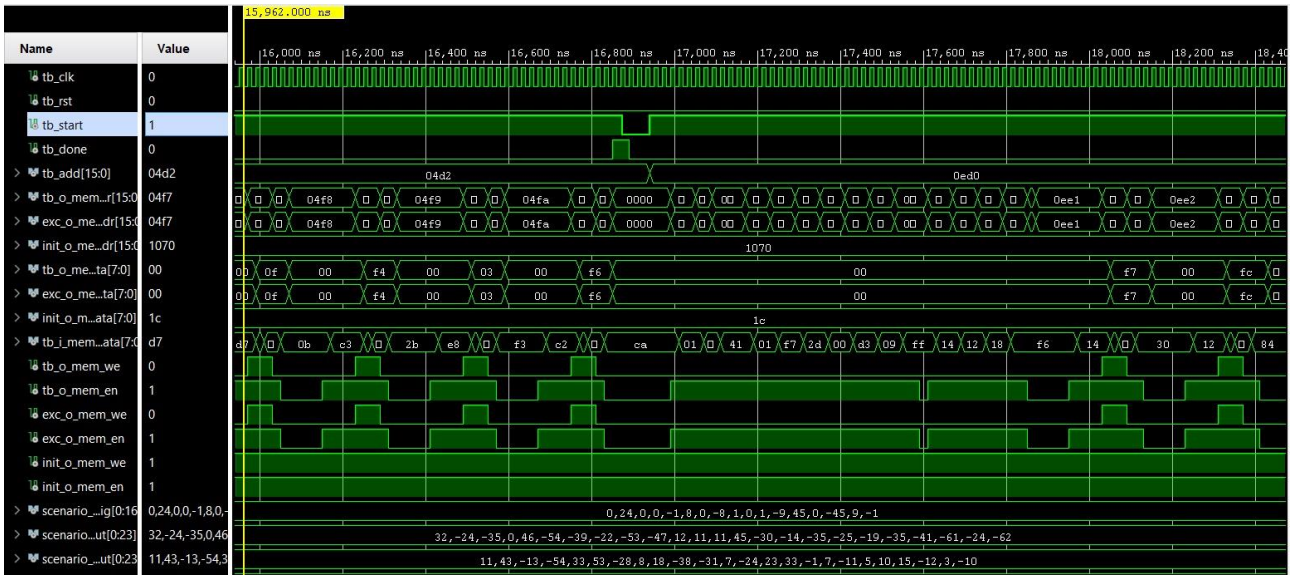
Appena viene ricevuto il segnale di START, il componente legge i due numeri e scrive in memoria i relativi risultati.



3.2.4 Quarto Test Bench

Quest'ultimo test bench prevede due esecuzioni consecutive (senza che venga utilizzato il segnale di reset).

In particolare, si può osservare che il segnale DONE viene posto ad "1" (e di conseguenza, come da specifica, il segnale START viene posto a "0"). In seguito, DONE sarà di nuovo posto a valore "0" attendendo che il segnale START abbia nuovamente valore alto e poter riprendere l'esecuzione da un nuovo indirizzo di memoria.



4. Conclusioni

Il componente implementato è correttamente **simulabile** e **sintetizzabile** sia in **pre-sintesi** che in **post-sintesi**, rispettando pienamente la specifica fornita come viene mostrato dall'estesa fase di simulazione e testing a cui è stato sottoposto. Inoltre, l'utilizzo di un approccio modulare ha garantito chiarezza procedurale e facilità di debug.

Sebbene il codice sia caratterizzato da un numero relativamente elevato di stati, che potenzialmente potrebbero rallentare l'elaborazione; dall'altra parte questo fatto rappresenta anche un punto di forza del progetto in quanto lo rende resistente e affidabile rispetto a lunghe sequenze in ingresso e a tutti i possibili scenari limite.

Le principali difficoltà che sono sorte durante lo sviluppo del componente sono dovute all'utilizzo di strumenti e di tecnologie non ancora familiari. Lo sviluppo del progetto è stato particolarmente utile per approfondire l'uso avanzato degli strumenti di sintesi e simulazione professionale e per chiarire aspetti e argomenti interessanti affrontati nel corso di Reti Logiche.