Relazione Progetto di Reti Logiche

Anno Accademico 2024-2025

Studenti: Barillari Francesco 10858068,
Benvenuti Andrea Roberto 10682511

Docente di riferimento: Fornaciari William

POLITECNICO
MILANO 1863

18 agosto 2025

Indice

1 Introduzione

1.1 Scopo del progetto

L'obiettivo del progetto è la realizzazione, in linguaggio VHDL, di un modulo hardware capace di elaborare una sequenza di dati salvata in memoria. In particolare, il modulo deve leggere la sequenza un byte per volta, applicare un filtro differenziale e scrivere il risultato finale in una zona specifica della memoria.

1.2 Specifiche generali

Il messaggio da elaborare è composto da una sequenza di k parole $W_1, W_2, \ldots, W_k$, ciascuna rappresentata in complemento a due e con valori compresi nell'intervallo $[-128, +127]$. Questi valori sono già memorizzati in memoria e vengono letti dal modulo uno alla volta.

Dopo l'elaborazione tramite il filtro differenziale, il modulo scrive la sequenza risultante in una posizione ben definita della memoria.

Il primo byte utile è posizionato all'indirizzo indicato dal segnale `i_add`, fornito direttamente dal testbench.

Il sistema utilizza un segnale di clock (`i_clk`) e un segnale di reset (`i_rst`), entrambi comuni all'intero progetto.

1.2.1 Applicazione del filtro

In base al valore del bit meno significativo del registro `regS`, che memorizza il parametro S, viene selezionato un filtro differente:

- Se il bit meno significativo di `regS` è 0, viene utilizzato un filtro di ordine 3, con normalizzazione $n = 12$.
- Se invece il bit meno significativo di `regS` è 1, viene applicato un filtro di ordine 5, con normalizzazione $n = 60$.

Il filtro è definito dalla seguente formula:

$$F'(i) = \frac{1}{n} \sum_{j=-l}^{+l} C_j \cdot f[j+i] \tag{1}$$

dove l vale 2 per il filtro di ordine 3 e 3 per il filtro di ordine 5; n è il valore di normalizzazione.

Dove:

- $F'(i)$ rappresenta il valore filtrato in posizione i
- C_j è il coefficiente moltiplicativo corrispondente all'indice j
- $f[i+j]$ è l'elemento della sequenza da filtrare
- n è il fattore di normalizzazione che varia in base al filtro selezionato

1.3 Gestione della Memoria

La memoria utilizzata dal modulo è istanziata all'interno del testbench ed è composta da 65536 locazioni, indirizzabili tramite parole di 16 bit. Ogni posizione contiene un dato codificato su 8 bit (1 byte). Le informazioni sono organizzate secondo il seguente schema:

- Gli indirizzi **i_add** e **i_add+1** contengono i byte `K1` e `K2`, che rappresentano rispettivamente la parte alta e bassa del valore K, ovvero la lunghezza della sequenza da elaborare.
- L'indirizzo **i_add+2** ospita il parametro S, il cui bit meno significativo determina il tipo di filtro da applicare (ordine 3 se $S = 0$, ordine 5 se $S = 1$).
- Le locazioni da **i_add+3** a **i_add+16** sono riservate ai coefficienti del filtro differenziale C_j (14 in totale).
- A partire da **i_add+17** fino a **i_add+17+K-1**, sono memorizzate le parole W_1 fino a W_K da elaborare.

Tabella 1: Mappa della memoria di input

Indirizzo	**Contenuto**
i_add	K1
i_add+1	K2
i_add+2	S
i_add+3, ..., i_add+16	Coefficienti C1, ..., C14
i_add+17, ..., i_add+17+K-1	Sequenza di input W1, ..., WK

1.4 Interfaccia del Componente

Il modulo è stato implementato in linguaggio VHDL e presenta la seguente interfaccia:

```
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;
```

Di seguito è riportata una descrizione sintetica delle porte del componente:

- **i_clk**: segnale di clock globale, generato dal testbench.
- **i_rst**: segnale di reset asincrono, utilizzato per inizializzare la macchina a stati finiti.

- **i_start**: segnale di avvio dell'elaborazione, controllato dal testbench.
- **i_add**: indirizzo di base della memoria contenente i dati di input e la configurazione.
- **o_done**: segnale di uscita che indica il termine dell'elaborazione.
- **o_mem_addr**: indirizzo di memoria corrente utilizzato per accedere in lettura o scrittura.
- **i_mem_data**: dati letti dalla memoria; validi un ciclo di clock dopo l'attivazione di o_mem_addr.
- **o_mem_data**: dati da scrivere in memoria, validi quando o_mem_we è attivo.
- **o_mem_en**: segnale di abilitazione della memoria; deve essere alto sia per lettura che scrittura.
- **o_mem_we**: segnale di abilitazione scrittura; alto solo in fase di scrittura.

2 Architettura

2.0.1 Processi utilizzati

L'architettura è suddivisa in più processi, distinti tra combinatori e sequenziali, con chiara separazione delle responsabilità:

Processi combinatori

- **Signals Process:** abilita accesso alla memoria e il caricamento nei registri in base allo stato corrente della Macchina a Stati Finiti (FSM).
- **Next State Process:** definisce le transizioni della macchina a stati, sulla base di condizioni logiche e segnali di stato.

Processi sequenziali

- **FSM Process:** aggiorna lo stato corrente della FSM al fronte di clock.
- **Processo di aggiornamento dei segnali di indirizzamento e sincronizzazione di memoria:** implementa la logica di indirizzamento della memoria, la gestione del ciclo di lettura e l'avanzamento della finestra mobile.
- **Processo di assegnamento o_done:** controlla il segnale di fine elaborazione, basandosi sul flag `last`.
- **Processo di caricamento regK (calcolo K):** combina i due byte `K1` e `K2` per ottenere la lunghezza della sequenza K.
- **Processo di caricamento regS (calcolo S):** memorizza il parametro S utilizzato per la selezione del filtro.

- **Processo di caricamento W:** aggiorna il registro a scorrimento `win` con le parole lette dalla memoria.
- **Processo di calcolo parziale (moltiplicazione W × C):** esegue il prodotto tra parola e coefficiente, aggiornando il valore parziale `partial_result`.
- **Processo di normalizzazione:** effettua la divisione approssimata su `partial_result` e produce il risultato normalizzato.
- **Processo MUX o_mem_addr:** determina l'indirizzo da inviare alla memoria in base a `addr_sel`, `count` e `i_add`.
- **Processo di scrittura:** gestisce il caricamento del valore normalizzato su `o_mem_data` e abilita la scrittura su memoria tramite `o_mem_we`.

2.0.2 Segnali utilizzati

Segnali di controllo

- `k_load` : `std_logic` : attiva il caricamento del registro `regK` per calcolare la lunghezza k della sequenza da elaborare.
- `s_load` : `std_logic` : attiva il caricamento del registro `regS`, che seleziona il filtro da utilizzare.
- `f_load` : `std_logic` : attiva il calcolo del prodotto parola × `coeff_filtro`.
- `SR_load` : `std_logic` : permette il caricamento delle parole W nella finestra mobile.
- `n_load` : `std_logic` : attiva il processo di normalizzazione.

Segnali di indirizzamento

- `addr_sel` : `std_logic_vector(1 downto 0)` : seleziona la sorgente/destinazione degli indirizzi di memoria:
  - `"00"` = lettura header iniziale (K_1, K_2, S)
  - `"01"` = lettura coefficienti del filtro
  - `"10"` = lettura parole W
  - `"11"` = scrittura risultati R
- `curr_addr` : `std_logic_vector(15 downto 0)` : indirizzo corrente usato per calcolare l'indirizzo successivo della finestra o di scrittura.
- `last` : `std_logic` : indica che l'ultima parola del risultato è stata scritta; viene usato per segnalare la fine dell'elaborazione.
- `count` : `integer` : contatore multiuso, impiegato per:
  - selezione di $K1$, $K2$ e S;
  - riempimento iniziale della finestra mobile;
  - lettura sequenziale dei coefficienti del filtro.

Segnali di sincronizzazione memoria

- `SR_read` : `std_logic` : segnala che una nuova parola è stata inserita nella finestra mobile.
- `coeff_read` : `std_logic` : segnala che il coefficiente corrente è stato correttamente caricato.
- `first_SR` : `std_logic` : indica se la finestra mobile è al primo caricamento; serve per il padding iniziale con zeri.
- `can_write` : `std_logic` : abilita la scrittura del risultato in memoria, sincronizzando indirizzo e dati.

Segnali di stato

- `curr_state` : `S` : stato attuale della macchina a stati finiti.
- `next_state` : `S` : stato successivo, determinato dal processo combinatorio.

Segnali per la gestione dei parametri

- `regK` : `std_logic_vector(15 downto 0)` : registro che memorizza il valore completo di K, costruito da K_1 e K_2 in due cicli consecutivi.
- `regS` : `std_logic` : registro che memorizza il parametro S per la selezione del filtro.

Finestra mobile, calcolo e normalizzazione

- `win` : `std_logic_vector(55 downto 0)` : finestra mobile composta da 7 byte, contenente le parole da $W[i-3]$ a $W[i+3]$.
- `partial_result` : `std_logic_vector(18 downto 0)` : risultato intermedio della somma pesata $\sum C_j \cdot W[j+i]$; 19 bit per prevenire overflow.
- `normalize` : `std_logic_vector(7 downto 0)` : valore finale filtrato e normalizzato, pronto per essere scritto in memoria.

2.1 Modulo Controller (FSM)

2.1.1 Scelte progettuali

Gestione delle parole *W* in memoria: Per la lettura della sequenza di input W, è stato adottato un registro a scorrimento di 56 bit, denominato `win`, suddiviso in sette parole da 8 bit ciascuna. Tale struttura funge da finestra mobile che, ad ogni ciclo di elaborazione, fornisce accesso a un sottoinsieme centrato di sette parole della sequenza.

Questa scelta architetturale è stata motivata dalla necessità di applicare un filtro convoluzionale centrato, che, in ogni istante, richiede l'accesso non solo alla parola corrente W_i, ma anche a quelle immediatamente precedenti e successive (da W_{i-3} a W_{i+3} nel caso del filtro di ordine 5).

Nel nostro progetto, la stessa finestra mobile (da W_{i-3} a W_{i+3}) viene utilizzata anche per il filtro di ordine 3: in questo caso, i coefficienti corrispondenti agli estremi (ovvero

$j = \pm 3$) sono semplicemente posti a zero. In tal modo, il supporto della convoluzione resta inalterato, semplificando l'architettura complessiva senza introdurre casi speciali nella gestione del registro.

L'utilizzo della finestra mobile consente un aggiornamento efficiente dei dati, evitando il ricalcolo esplicito degli indirizzi e riducendo la complessità del controllo. In particolare:

- nella prima fase (riempimento iniziale), la finestra viene popolata con le prime 4 parole della sequenza e 3 zeri iniziali (padding) per rappresentare i valori non ancora presenti;
- nei cicli successivi, la finestra viene aggiornata tramite uno *shift* verso sinistra di 8 bit, con inserimento in coda della nuova parola letta dalla memoria;
- i valori non disponibili (inizio e fine sequenza) sono gestiti mediante inserimento esplicito di zeri nel registro a scorrimento, come richiesto dalla specifica.

Questa strategia consente di mantenere costante il centro della convoluzione e uniformare il trattamento delle condizioni iniziali e finali, con notevole semplificazione della logica di indirizzamento.

Gestione di K1 e K2: Come per le parole W, anche per i primi due dati letti da memoria, si è deciso di adottare un registro a scorrimento di 16 bit. In questo modo è possibile comporre agevolmente il valore K, ottenuto dalla concatenazione dei byte K1 e K2.

Gestione dei coefficienti del filtro: A differenza della lettura delle parole W, la gestione dei coefficienti del filtro differenziale prevede l'accesso sequenziale a 7 valori memorizzati consecutivamente in memoria. In base al valore del parametro `regS` (caricato nella fase `REG_S`), viene selezionata la posizione iniziale dei coefficienti: se `regS` = 0, vengono letti i coefficienti del filtro di ordine 3 (da `i_add + 3` a `i_add + 9`); se `regS` = 1, quelli del filtro di ordine 5 (da `i_add + 10` a `i_add + 16`).

Durante il calcolo convoluzionale, ogni coefficiente C_j viene caricato dalla memoria e moltiplicato per l'elemento corrispondente della finestra, utilizzando il contatore `count` come indice implicito del ciclo. Questa modalità evita la necessità di una struttura array interna per memorizzare i coefficienti: essi vengono letti direttamente dalla memoria al momento dell'utilizzo, riducendo l'area occupata in sintesi e semplificando il controllo.

Gestione dell'indirizzamento (MUX): L'indirizzo da inviare alla memoria esterna è determinato combinando il contatore `count`, l'indirizzo base `i_add` e il segnale `addr_sel`. Il calcolo dell'indirizzo avviene dinamicamente nel processo MUX, considerando anche condizioni particolari come il primo caricamento della finestra (`first_SR`) e l'avanzamento di `curr_addr`. In particolare:

- `count`, un contatore utilizzato in tutte le fasi: lettura dei 3 byte iniziali, accesso ai coefficienti e gestione della finestra;
- `curr_addr`, che mantiene il riferimento all'indirizzo corrente della parola `W[i]` e serve per calcolare l'indirizzo di scrittura.

- `addr_sel`, che specifica l'operazione corrente:
  - `addr_sel = ''00''` → lettura header iniziale (`K1`, `K2`, `S`);
  - `addr_sel = ''01''` → lettura coefficienti del filtro (in base a `regS`);
  - `addr_sel = ''10''` → lettura parole W della sequenza (costruzione finestra);
  - `addr_sel = ''11''` → scrittura dei risultati normalizzati (parole R).

L'indirizzamento corretto è essenziale per garantire che i dati vengano letti e scritti nei cicli appropriati, in coerenza con il comportamento sincrono della memoria.

```
process(i_clk, i_rst)
-- code
    elsif rising_edge(i_clk) then
        case addr_sel is
            when "00" => -- leggiamo k1, k2, s
                o_mem_addr <= std_logic_vector ( unsigned(i_add) + count
                    );
            when "01" =>-- lettura dei filtri
                if regS = '0' then
                    o_mem_addr <= std_logic_vector ( unsigned(i_add) + 3
                        + count );
                elsif regS = '1' then
                    o_mem_addr <= std_logic_vector ( unsigned(i_add) +
                       10 + count );
                end if;
            when "10" => -- lettura delle k parole
                if first_SR = '1' then

                    o_mem_addr <= std_logic_vector ( unsigned(i_add) +
                       17 + count );
                elsif first_SR = '0' then
                    o_mem_addr <= curr_addr;
                end if;
            when "11" => -- scrittura di R
                o_mem_addr <= std_logic_vector ( unsigned(curr_addr) +
                   unsigned(regK) - 3);
            when others =>
                o_mem_addr <= ( others => '0' );
        end case;
    end if;
end process;
```

Qui di seguito illustriamo la parte piu significante del nostro datapath, su cui si basa la stesura del codice VHDL riportato appena sopra, in particolare riguardante l'assegnamento dell'indirizzo di accesso alla memoria in lettura e scrittura.

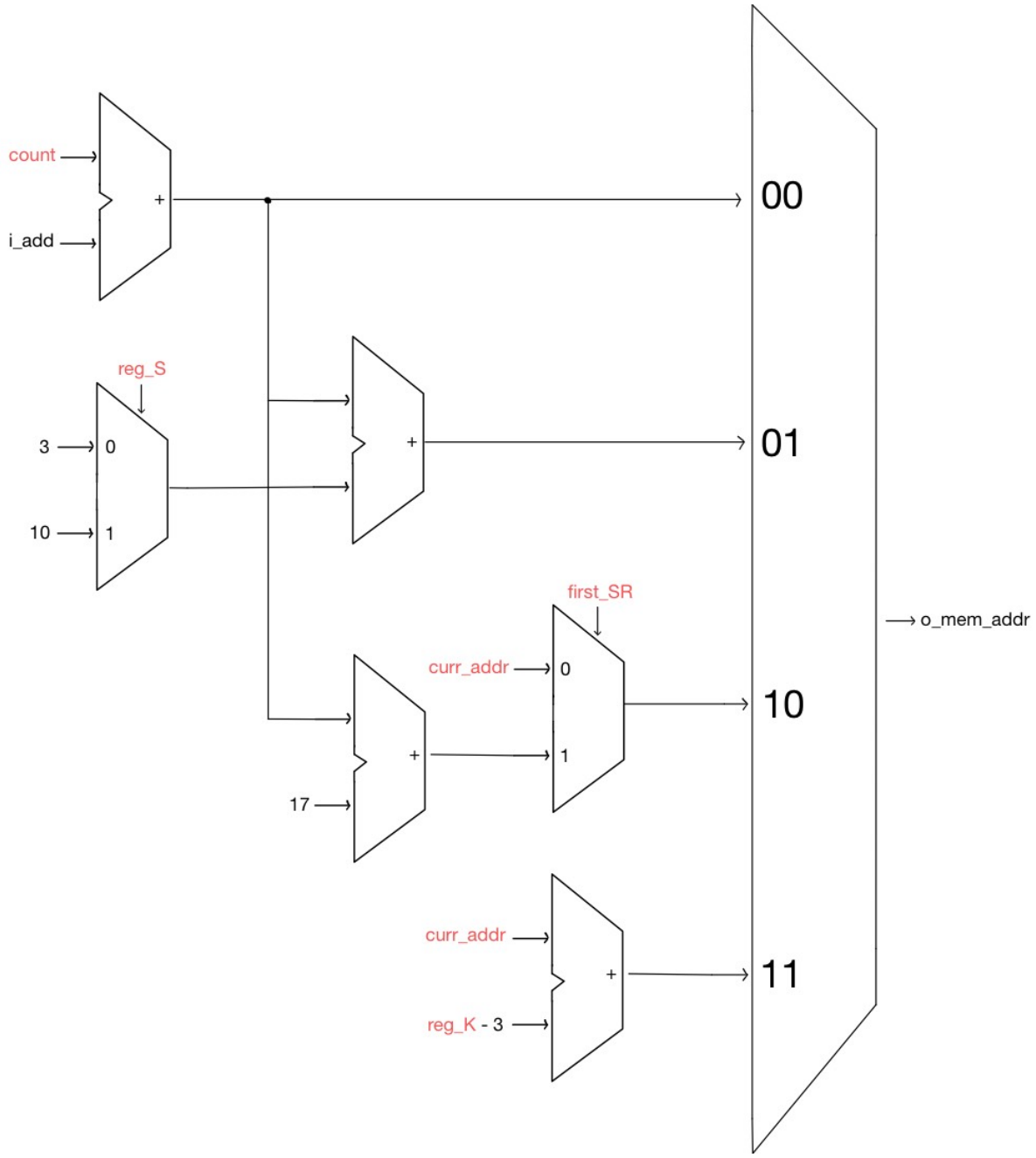


Figura 1: Datapath

Data la complessità del datapath, abbiamo deciso di non rappresentare il funzionamento dei segnali in rosso nell'immagine, conferendo maggiore leggibilità e chiarezza. Questi vengono gestiti nei vari stati della FSM, come già ampiamente discusso.

2.1.2 Macchina a Stati Finiti (FSM)

La FSM progettata si compone di 16 stati principali, organizzati logicamente in quattro fasi sequenziali:

1. Lettura dei primi 3 byte di configurazione (K_1, K_2, S)
2. Riempimento iniziale e aggiornamento della finestra mobile `win`

3. Lettura dei coefficienti e calcolo convoluzionale

4. Scrittura del risultato filtrato e normalizzato in memoria

A causa della natura sincrona della memoria (latenza di un ciclo tra indirizzo e dato valido), ogni lettura richiede uno stato di attesa intermedio, implementato tramite stati `WAIT_x` dedicati per ogni fase.

2.1.3 Descrizione dettagliata degli stati

IDLE Stato iniziale e di inattività del sistema. Vengono inizializzati tutti i segnali di controllo e stato. La FSM resta in questo stato finché non riceve un impulso positivo sul segnale `i_start`.

ASK_L Abilita la lettura dei tre byte iniziali dalla memoria (K_1, K_2, S) alzando il segnale `o_mem_en` e settando `addr_sel = "00"`; l'indirizzo è calcolato tramite `i_add + count`.

WAIT_L Gestisce la latenza di lettura della memoria. Se `count < 2` si passa a `REG_K`, altrimenti a `REG_S`.

REG_K Carica il valore di K_1 o K_2 (in base al valore corrente di `count`) nel registro `regK` tramite il segnale `k_load`.

REG_S Carica il valore di S nel registro regS tramite `s_load`. Al termine, count viene azzerato e `curr_addr` viene inizializzato a `i_add + 19`, per consentire la lettura delle parole W, ad eccezione del primo riempimento.

ASK_W Attiva la lettura delle parole W da caricare nella finestra mobile. ASK_W è lo stato che collega la parte iniziale della FSM con gli stati seguenti, consentendo la scrittura di tutte le parole. Qui, inizia il ciclo di scrittura per ogni parola.

WAIT_W Stato di attesa per la latenza della memoria nella lettura di `W`. Se `first_SR = '1'` si avvia il riempimento iniziale, altrimenti si prosegue con lo shift della finestra.

ADD_W Aggiorna la finestra mobile `win`. Se è il primo caricamento, si aggiungono i primi 4 valori effettivi e 3 zeri iniziali (padding). Negli altri casi, lo shift di 8 bit consente di caricare la nuova parola in coda.

TAKE_COEFF Imposta l'indirizzo di lettura del coefficiente corretto e attiva `addr_sel = "01"`. Qui inizia il ciclo per il calcolo del risultato.

WAIT_COEFF Gestisce il tempo di latenza per la lettura del coefficiente dalla memoria.

COMPUTE Esegue la moltiplicazione tra la parola `W[i+j]` e il coefficiente `C[j]` e accumula il risultato in `partial_result`, tramite l'attivazione del segnale `f_load`.

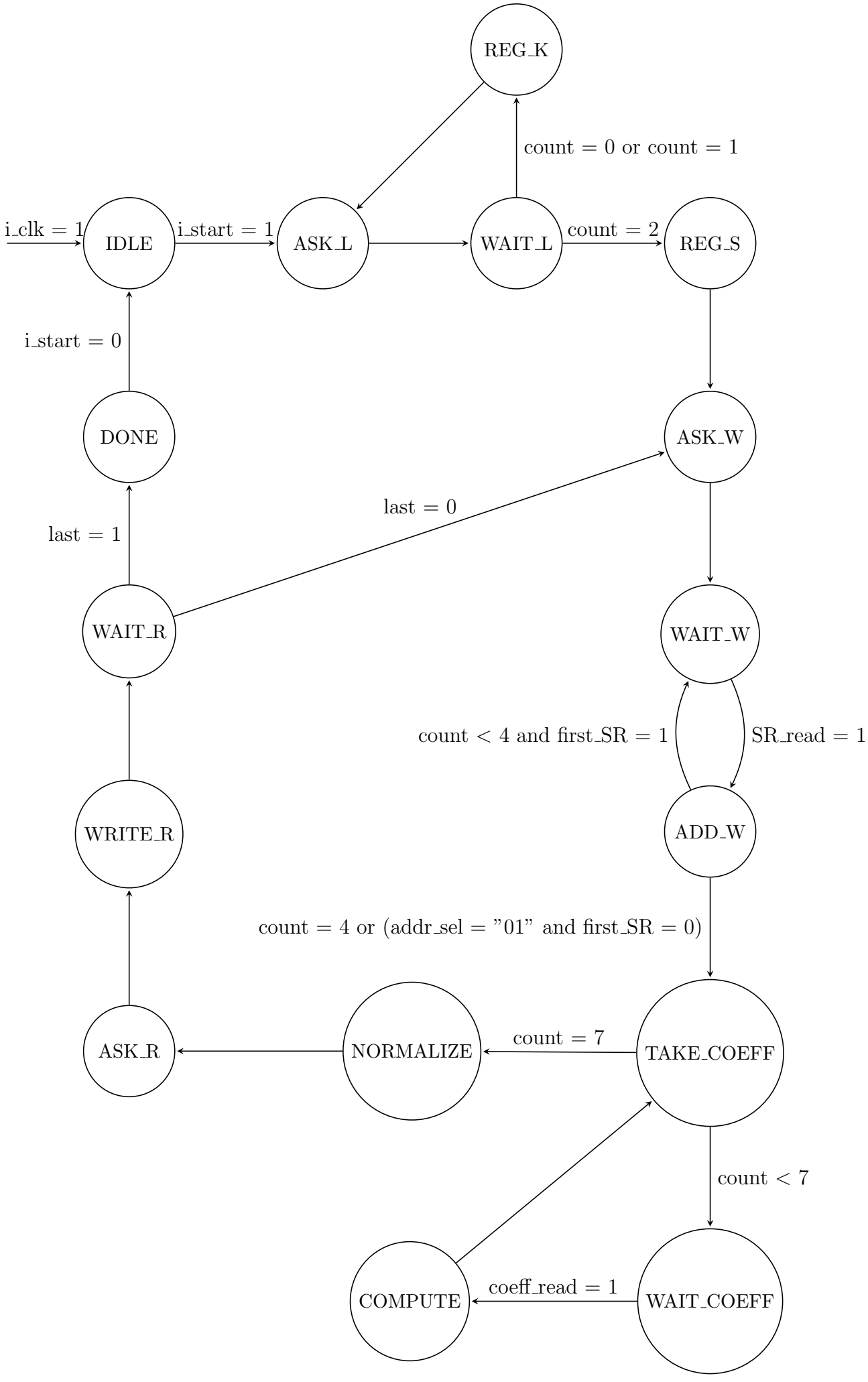
NORMALIZE Al termine della convoluzione, normalizza il valore ottenuto secondo la formula specificata (con approssimazione tramite shift logici) e lo salva nel registro `normalized`, tramite `n_load`.

ASK_R Prepara la scrittura del risultato normalizzato in memoria. Calcola l'indirizzo corretto e abilita il processo di scrittura ponendo can_write a 1.

WRITE_R Effettua la scrittura su memoria: `o_mem_we = '1'` e `o_mem_data = normalized`. Verifica se l'indirizzo corrente corrisponde all'ultima parola (`last = '1'`).

WAIT_R Controlla se la sequenza è terminata. Se `last = '1'` si passa a `DONE`, altrimenti si ritorna a `ASK_W` per elaborare la prossima parola.

DONE Segnale finale del processo. Viene mantenuto attivo `o_done = '1'` fino a quando `i_start` non torna a zero, riportando il sistema in `IDLE`.


REG_K
count = 0 or count = 1
i_clk = 1
IDLE
i_start = 1
ASK_L
WAIT_L
count = 2
REG_S
i_start = 0
DONE
ASK_W
last = 0
last = 1
WAIT_R
WAIT_W
count < 4 and first_SR = 1
SR_read = 1
WRITE_R
ADD_W
count = 4 or (addr_sel = "01" and first_SR = 0)
ASK_R
NORMALIZE
count = 7
TAKE_COEFF
count < 7
COMPUTE
coeff_read = 1
WAIT_COEFF

3 Risultati Sperimentali

3.1 Sintesi

3.1.1 Configurazione Target

- FPGA: Artix-7 xc7a200tfbg484-1
- Tool: Xilinx Vivado WebPACK
- Frequenza target: 50 MHz (periodo 20 ns)

3.1.2 Utilizzo Risorse

Tutti i testbench esaminati a livello comportamentale, come precedentemente documentato, hanno completato con esito positivo la simulazione post-Synthesis.

- **report_utilization:**

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	496	0	0	134600	0.37
LUT as Logic	496	0	0	134600	0.37
LUT as Memory	0	0	0	46200	0.00
Slice Registers	191	0	0	269200	0.07
Register as Flip Flop	191	0	0	269200	0.07
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

Tabella 2: Resource Utilization Summary

Dalla tabella riportata sopra emerge chiaramente che il modulo non fa uso di latch.

- **report_timing:**

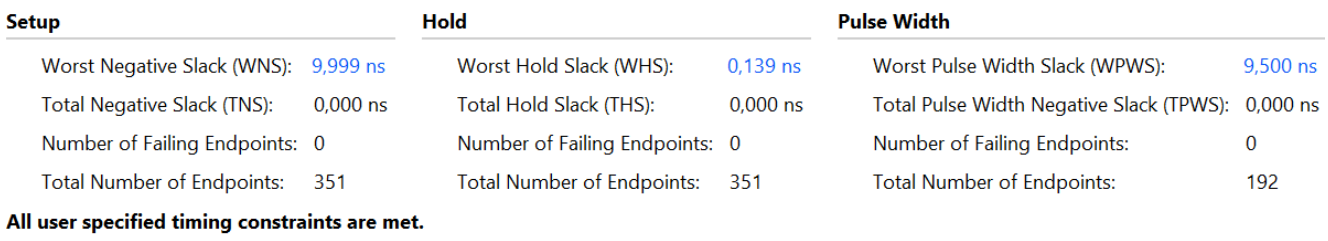


Figura 2: Design Timing Summary

Il report di timing dimostra la conformità del progetto alle specifiche richieste. La tabella seguente presenta il timing report, dal quale emerge che il Worst Negative Slack (WNS) risulta essere di circa 10ns.

3.2 Simulazioni Pre-Sintesi

3.2.1 Test Bench "TB_1" - Esempio dalla Specifica

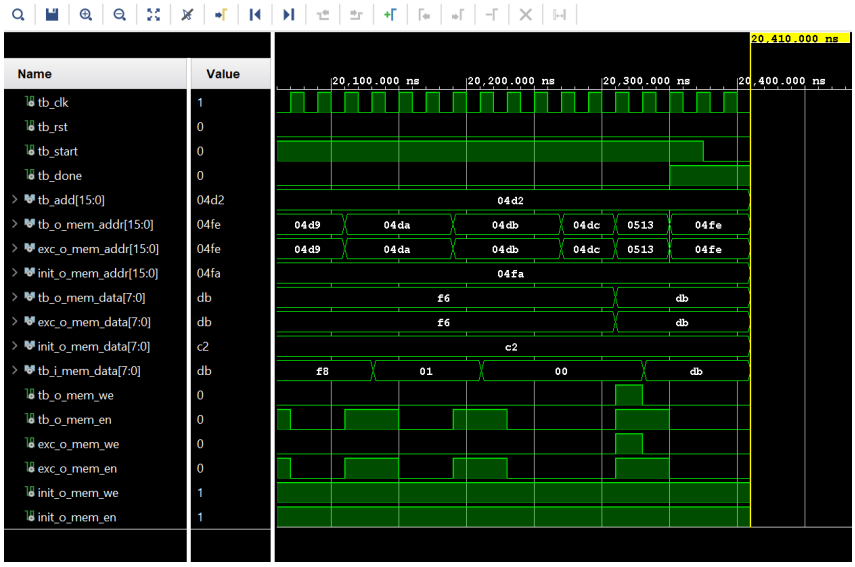


Figura 3: Waveform Behavioral del testbench fornito dal prof Fornaciari

Obiettivo: Verifica funzionamento base del componente con dati ufficiali
Base dati: ESEMPIO 3 dalla specifica (filtro ordine 3, 24 elementi)

Configurazione test:

- Lunghezza sequenza: 24 byte
- Filtro: Ordine 3 (S=0), coefficienti [0, -1, 8, 0, -8, 1, 0]
- Sequenza input: [32, -24, -35, 0, 46, -54, -39, -22, -53, -47, 12, 11, 11, 45, -30, -14, -35, -25, -19, -35, -41, -61, -24, -62]
- Output atteso: [11, 43, -13, -54, 33, 53, -28, 8, 18, -38, -31, 7, -24, 23, 33, -1, 7, -11, 5, 10, 15, -12, 3, -10]

Aspetti verificati:

- Protocollo START/DONE corretto
- Lettura coefficienti e configurazione dalla memoria
- Calcolo filtro differenziale ordine 3
- Normalizzazione 1/12 con approssimazione shift
- Scrittura risultati nella posizione corretta (ADD+17+K)

Risultato:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 20410 ns  Iteration: 1  Process: /test2425/test_routine
```

3.2.2 Test Bench "TB_2" - Saturazione Limite Inferiore

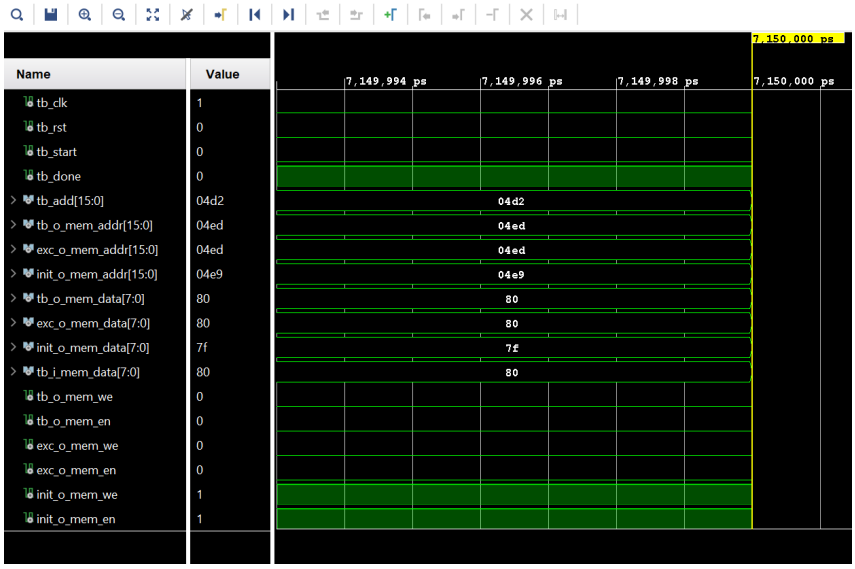


Figura 4: Waveform Behavioral del testbench TbNegativi

Obiettivo: Verifica robustezza per valori negativi estremi e rappresentazione intermedia adeguata
Configurazione critica: Progettato per generare i risultati intermedi negativi più ampi possibili

Configurazione test:

- Lunghezza sequenza: 7 byte (minimo consentito)
- Filtro: Ordine 5 (S=1) per massimizzare i coefficienti
- Sequenza input: [127, 127, 127, 127, 127, 127, 127] (tutti valori massimi positivi)
- Output atteso: [-128, -128, -128, -128, -128, -128, -128] (saturazione al limite inferiore)

Aspetti verificati:

- Saturazione: Risultati < -128 correttamente saturati a -128
- Troncamento: Normalizzazione applicata correttamente prima della saturazione
- Gestione edge case: Sequenza lunghezza minima gestita correttamente

Risultato:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 7150 ns  Iteration: 1  Process: /tb2425N/test_routine
```

3.2.3 Test Bench "TB_3" - Saturazione Limite Superiore

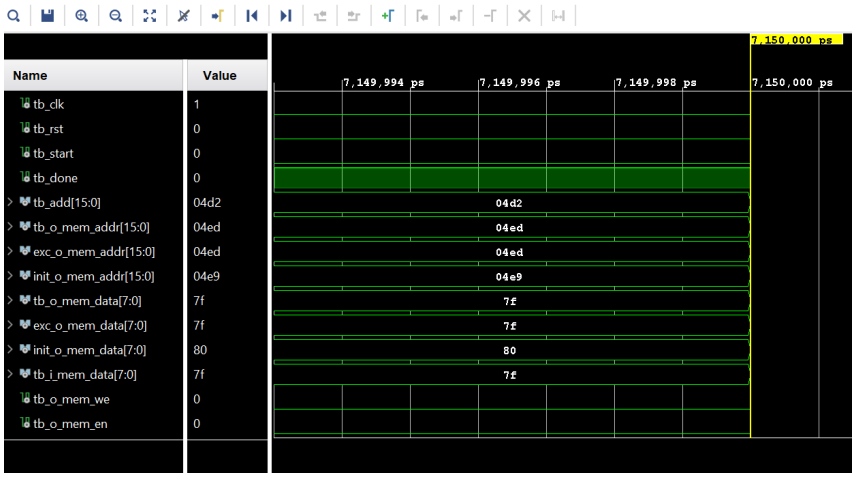


Figura 5: Waveform Behavioral del testbench TbPositivi

Obiettivo: Verifica robustezza per valori positivi estremi e rappresentazione intermedia adeguata
Configurazione critica: Progettato per generare i risultati intermedi positivi più ampi possibili

Configurazione test:

- Lunghezza sequenza: 7 byte (minimo consentito)
- Filtro: Ordine 5 (S=1) per massimizzare i coefficienti
- Sequenza input: [-128, -128, -128, -128, -128, -128, -128] (tutti valori minimi negativi)
- Output atteso: [127, 127, 127, 127, 127, 127, 127] (saturazione al limite superiore)

Aspetti verificati:

- Saturazione: Risultati $> +127$ correttamente saturati a $+127$
- Troncamento: Normalizzazione applicata correttamente prima della saturazione
- Simmetria: Comportamento speculare rispetto al test negativo

Risultato:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 7150 ns  Iteration: 1  Process: /tb2425P/test_routine
```

3.2.4 Test Bench "TB_4" - Caso Risultato Nullo

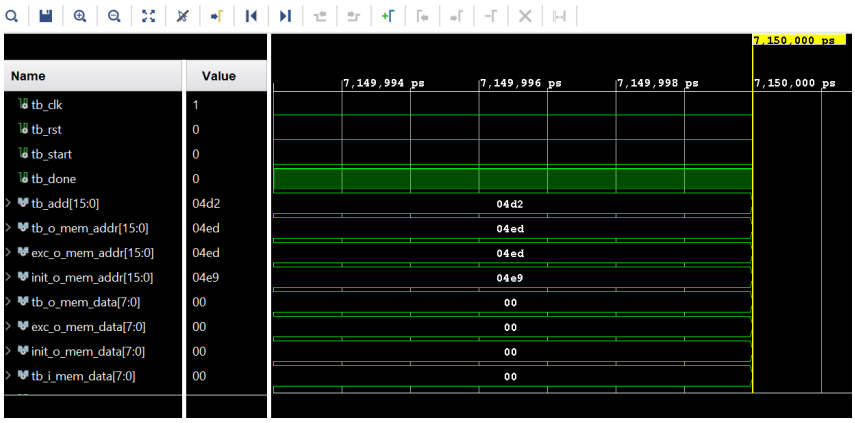


Figura 6: Waveform Behavioral del testbench TbZero

Obiettivo: Verifica gestione del caso particolare in cui tutti i risultati sono zero
Scopo specifico: Validare comportamento con risultati intermedi nulli e normalizzazione di valori zero

Configurazione test:

- Lunghezza sequenza: 7 byte
- Filtro: Ordine 5 (S=1)
- Sequenza input: [0, 0, 0, 0, 0, 0, 0] (tutti zeri)
- Output atteso: [0, 0, 0, 0, 0, 0, 0] (tutti zeri)

Verifiche specifiche:

- Calcoli con zeri: Moltiplicazioni e somme con operandi nulli
- Normalizzazione di zero: Divisioni di zero gestite correttamente
- Saturazione non necessaria: Verifica che 0 rimanga 0
- Padding implicito: Conferma che i valori "fantasma" ai bordi siano effettivamente zero

Risultato:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 7150 ns  Iteration: 1  Process: /tb24250/test_routine
```

3.2.5 Test Bench "TB_5" – Suite di Test Automatizzati Multi-Scenario

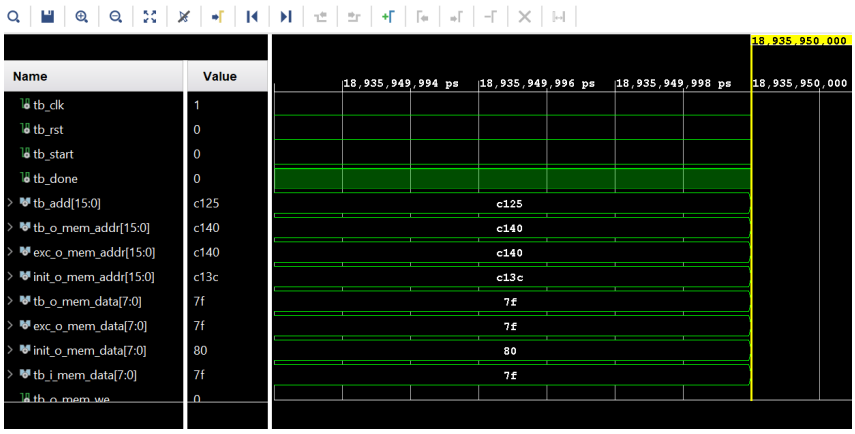


Figura 7: Waveform Behavioral del testbench TbMK2

Obiettivo: Validare in modo esaustivo il funzionamento del progetto su diversi casi d'uso, incluse sequenze lunghe, coefficienti variabili, edge case e verifiche di saturazione. Il test bench TB_5 include sei scenari di test, ciascuno con una propria configurazione completa (K1, K2, selettore filtro S, 14 coefficienti, sequenza input e output atteso).

Caratteristiche principali del test bench:

- Verifica automatica del risultato confrontando l'output scritto in memoria con i valori attesi
- Ogni scenario è identificato da un proprio indirizzo base e lunghezza K
- L'intero test viene eseguito senza intervento manuale, e segnala errori in caso di mismatch
- Gestione simultanea dei segnali di controllo e della memoria da parte del test bench
- Simula anche scenari estremi per valori positivi/negativi estremi e saturazioni multiple

Tabella 3: Scenari Implementati nel Test Bench tbMark2

Scenario	Lunghezza K	Coefficienti	Obiettivo specifico
1	12000	Default ordine 3	Ordine 5 su sequenza lunga
2	12000	Default ordine 5	Ordine 5 su sequenza lunga
3	47	Coefficienti modificati	Output misto
4	24	Default ordine 3	Conformità con esempio ufficiale
5	7	Input tutti a +127	Saturazione inferiore (valori → -128)
6	7	Input tutti a -128	Saturazione superiore (valori → +127)

Verifiche incluse:

- Protocollo START/DONE
- Lettura dati e coefficienti
- Normalizzazione con correzione shift per negativi
- Saturazione a ±128
- Scrittura in memoria all'indirizzo ADD + 17 + K
- Confronto automatico con scenario_output

Risultato:

```
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
Time: 18935950 ns  Iteration: 1  Process: /tb2425MK2/test_routine
```

3.2.6 Analisi Complessiva dei Test

La validazione del progetto è stata condotta attraverso un insieme completo di test, che hanno verificato in modo esaustivo il corretto funzionamento del componente VHDL in diverse condizioni operative. L'obiettivo principale era dimostrare che l'implementazione rispettasse pienamente le specifiche, garantendo robustezza, correttezza aritmetica e gestione affidabile dei casi limite.

Test singoli Sono stati eseguiti quattro test focalizzati su singoli scenari critici:

1. **Funzionalità nominale (`TB_1`)**: ha verificato il comportamento del filtro con dati ufficiali (Esempio 3 della specifica), confermando che la pipeline completa (lettura, convoluzione, normalizzazione, scrittura) funzioni come previsto.
2. **Limiti negativi (`TB_2`)**: ha testato l'intera catena operativa con input a +127, generando i valori intermedi negativi più estremi per verificare la saturazione al valore minimo rappresentabile (-128).
3. **Limiti positivi (`TB_3`)**: ha testato input a -128 per generare i valori intermedi massimi e verificare la corretta saturazione a +127.
4. **Caso particolare (`TB_4`)**: ha validato il comportamento del sistema con una sequenza interamente composta da zeri, garantendo l'assenza di errori in condizioni di segnale nullo.

Test multi-scenario automatizzato A questi si aggiunge la suite **TB_5**, composta da sei scenari eterogenei (inclusi casi estremi e sequenze lunghe), eseguiti automaticamente in successione. Questa suite consente di validare il comportamento del modulo su un ampio spettro di condizioni, tra cui variazioni nella lunghezza K, nel tipo di filtro, nei coefficienti e nei dati d'ingresso. I risultati scritti in memoria vengono confrontati direttamente con l'output atteso, rilevando eventuali incongruenze.

Copertura complessiva verificata:

- Entrambi i filtri gestiti (ordine 3 e ordine 5), con corretta interpretazione del bit meno significativo del parametro `S`
- Lunghezze minime (K=7) e molto elevate (K=12000), garantendo scalabilità
- Comportamento corretto in condizioni estreme: massimi/minimi rappresentabili
- Normalizzazione tramite somma di shift aritmetici, con corretta correzione per valori negativi
- Saturazione dei valori finali nel range $[-128, +127]$
- Calcolo convoluzionale con 7 moltiplicazioni-accumulo per ogni parola W_i
- Protocollo di comunicazione con la memoria (segnali `o_mem_addr`, `o_mem_we`, `o_mem_en`) rispettato in ogni fase
- Comportamento conforme al segnale di controllo `i_start` e segnalazione corretta con `o_done`

Nel complesso, l'implementazione si è dimostrata affidabile, funzionale e conforme alle specifiche su tutti gli scenari di test. La combinazione di test mirati e suite automatizzate ha permesso di ottenere una copertura completa sia dal punto di vista funzionale che temporale, fornendo garanzia di correttezza anche in condizioni di margine.

4 Conclusioni

4.1 Risultati Conseguiti

- Implementazione VHDL completamente sintetizzabile e priva di latch indesiderati
- Rispetto del vincolo temporale di 20 ns con margine sui cammini critici
- Comportamento identico tra simulazione pre-sintesi e post-sintesi
- Gestione completa dei casi limite: saturazioni, input nulli, massimi/minimi
- Corretto utilizzo dei segnali `i_start` e `o_done` per controllo esecuzione
- Comunicazione conforme con memoria esterna secondo protocollo richiesto

Tutte le simulazioni eseguite nella fase pre-sintesi sono state ripetute anche in post-sintesi. L'obiettivo era verificare che il comportamento funzionale del componente fosse preservato, nonostante le ottimizzazioni e trasformazioni effettuate dal tool di sintesi.

4.2 Considerazioni Finali

Il modulo da noi implementato ha superato con successo tutti i testbench sostenuti. Le principali ottimizzazioni riguardano la riduzione dell'utilizzo delle risorse e l'eliminazione completa del latch, inizialmente dovuta all'incorretta assegnazione del segnale `o_done`. La versione finale della macchina a stati avrebbe potuto presentare un numero inferiore di stati; tuttavia, considerando i ritardi dei segnali, i limiti dati dalla specifica e la necessità di accedere frequentemente alla memoria (dettata dalle nostre scelte architetturali, già descritte in precedenza) abbiamo raggiunto la soluzione proposta.