Relazione Prova Finale di Reti Logiche

De Giorgio Domenico [10854350],Cristianelli Ennio [10846158]

26 febbraio 2026

1 Introduction

1.1 Specifica

Il presente report descrive l'implementazione, in linguaggio VHDL e tramite l'ambiente di sviluppo Vivado, di un modulo hardware. Il modulo è stato sviluppato con l'obiettivo di interfacciarsi con una memoria RAM esterna, leggere un messaggio costituito da una sequenza di K parole in complemento a 2 (intervallo -128 a +127), applicare un filtro differenziale di ordine 3 o 5 selezionato dinamicamente, e scrivere in memoria il risultato filtrato. Tali interazioni con la memoria sono gestite da segnali dedicati di indirizzamento, abilitazione, lettura e scrittura. L'elaborazione ha inizio quando viene attivato il segnale START e si conclude con l'attivazione del segnale DONE. In qualsiasi momento, l'ingresso RESET può essere utilizzato per riportare il sistema allo stato iniziale. La lettura da memoria ha inizio dall'indirizzo ADD fornito al modulo e i valori letti seguono la seguente struttura:

Header (17 byte):

- **K1, K2**: 2 byte che indicano la lunghezza della sequenza K di valori su cui eseguire il calcolo.
- **S**: 1 byte, ne viene utilizzato unicamente il bit meno significativo, se ha valore "0" il modulo applica il filtro di ordine 3, se ha valore "1" applica il filtro di ordine 5.
- **C1/C14**: 14 byte, i primi 7 memorizzano i coefficienti del filtro di ordine 3, i successivi contengono i coefficienti del filtro di ordine 5

L'header è seguito dalla sequenza di K byte di payload contenenti i valori su cui è richiesto di eseguire il calcolo. Tale calcolo consiste nell'applicare la funzione: $f'(i)$= (1/n) * $\sum$ Cj * f[j+i] con j che va da –l e +l e con l che vale 2 per il filtro di ordine 3 e 3 per il filtro di ordine 5; n è il valore di normalizzazione, con n= 12 per il filtro di ordine 3 ed n= 60 per il filtro di ordine 5.

1.2 Interfaccia 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;
```

Segnali di input:

- i_clk è il segnale di CLOCK fornito dal Test Bench.
- i_rst è il segnale di RESET fornito dal Test Bench.
- i_start è il segnale di START che avvia la computazione.
- i_add è il vettore ADD generato dal Test Bench che indica l'indirizzo di memoria da cui parte la sequenza di elaborazione.
- i_mem_data è il segnale (vettore) che arriva dalla memoria e contiene il dato in seguito ad una richiesta di lettura.

Segnali di output:

- o_done è il segnale DONE di uscita che comunica la fine dell'elaborazione.
- o_mem_addr è il segnale (vettore) di uscita che manda l'indirizzo alla memoria.
- o_mem_data è il segnale (vettore) che va verso la memoria e contiene il dato che verrà successivamente scritto.
- o_mem_en è il segnale di ENABLE da dover mandare alla memoria.
- o_mem_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci, (=0) per leggere.

2 Architettura

Iniziamo mostrando la struttura della macchina a stati finiti che regola le fasi della computazione del modulo in quanto permette una visione d'insieme del suo funzionamento:

2.1 Macchina a Stati Finiti

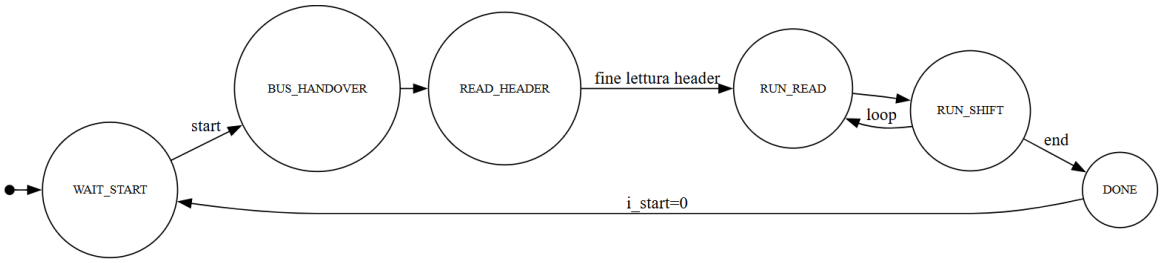
Dichiarazione del tipo state_t e inizializzazione dello stato del FSM

```
type state_t is (WAIT_START, READ_HEADER, RUN_READ, RUN_SHIFT, DONE);
    signal state : state_t := WAIT_START;
```

successivamente nella Tabella 1 è spiegato il ruolo di ciascun stato

Tabella 1: Tabella degli Stati

Stato	**Funzionamento**
WAIT_START	Si accede a questo stato a seguito di un reset o dopo il completamento di un di un calcolo, quindi passando da DONE quando i_start=0, in questo stato si attende semplicemente che quest'ultimo segnale si alzi per iniziare la computazione.
READ_HEAD	Abilita la memoria in modo da poterla leggere, in questo stato si caricano nei registri le informazioni contenute nel header in modo da poterle successivamente utilizzare nei calcoli, l'inserimento del dato nel registro corretto è gestita da un demultiplexer, si passa allo stato successivo una volta completata la lettura dell' intero header.
RUN_READ	La memoria viene utilizzata in lerttura con l'obiettivo di leggere i valori appartenenti alla sequenza di K byte necessari per effettuare il calcolo.
RUN_SHIFT	In questo stato vengono eseguite le operazioni di calcolo il cui funzionamento verrà approfondito in seguito, nel caso in cui l'uscita sia valida in oltre si abilita la memoria in scrittura in modo da salvare il risultato in memoria. da questo stato ci si sposta in RUN_READ, per ottenere i dati necessari a proseguire il calcolo, o in DONE nel caso in cui la computazione sia completata.
BUS_HANDOVER	Stato di transizione utilizzato nei casi che rischiano di disallineare il ciclo di lettura. Durante questo stato i segnali o_mem_en e o_mem_we restano a 0, viene preparato il primo indirizzo di header e la macchina passa allo stato READ_HEADER.
DONE	segnala la fine del processo di calcolo, e cessa le attività di memoria, da questo stato ci si sposta i WAIT_START solo a seguito del abbassamento del segnale di start



2.2 Header

La seguente porzione di codice garantisce il caricamento nel registro corretto dei dati relativi al header, in particolare il valore di header_count3 permette di indirizzare il dato letto nel registro corretto in base a quante letture da memoria sono già state eseguite

```
case header_count3 is
    when 0  => k1           <= mem_regi;
    when 1  => k2           <= mem_regi;
    when 2  => s            <= mem_regi;
    when 3  => c_bytes(0)   <= mem_regi;
    when 4  => c_bytes(1)   <= mem_regi;
    when 5  => c_bytes(2)   <= mem_regi;
    when 6  => c_bytes(3)   <= mem_regi;
    when 7  => c_bytes(4)   <= mem_regi;
    when 8  => c_bytes(5)   <= mem_regi;
    when 9  => c_bytes(6)   <= mem_regi;
    when 10 => c_bytes(7)   <= mem_regi;
    when 11 => c_bytes(8)   <= mem_regi;
    when 12 => c_bytes(9)   <= mem_regi;
    when 13 => c_bytes(10)  <= mem_regi;
    when 14 => c_bytes(11)  <= mem_regi;
    when 15 => c_bytes(12)  <= mem_regi;
    when 16 => c_bytes(13)  <= mem_regi;
    when others => null;
end case;
if header_count < HEADER_BYTES_I-1 then
    header_count <= header_count + 1;
```

2.3 Calcolo

2.3.1 Selezione dei coefficienti

I coefficienti dell'header vengono selezionati a seconda del filtro richiesto e passati a coeffs_sel per poi essere usati nella computazione del calcolo, tenendo conto che il primo e ultimo coefficiente del filtro di ordine 3 sarà sempre 0.

```
coeffs_sel(0) <= to_signed(0,8)           when s(0)='0' else signed(
   c_bytes(7));
coeffs_sel(1) <= signed(c_bytes(1))       when s(0)='0' else signed(
   c_bytes(8));
coeffs_sel(2) <= signed(c_bytes(2))       when s(0)='0' else signed(
   c_bytes(9));
coeffs_sel(3) <= signed(c_bytes(3))       when s(0)='0' else signed(
   c_bytes(10));
coeffs_sel(4) <= signed(c_bytes(4))       when s(0)='0' else signed(
   c_bytes(11));
coeffs_sel(5) <= signed(c_bytes(5))       when s(0)='0' else signed(
   c_bytes(12));
coeffs_sel(6) <= to_signed(0,8)           when s(0)='0' else signed(
   c_bytes(13));
```

2.3.2 Finestra dati

Tramite lo scorrimento del registro x_window si portano i valori dei k campioni del payload da computare nei registri x, partendo dall'ingresso in x6. Lo scorrimento permette di usare i valori per il calcolo del risultato successivo, associandoli al nuovo coefficiente di moltiplicazione.

```
x0 := x_window(1);
x1 := x_window(2);
x2 := x_window(3);
x3 := x_window(4);
x4 := x_window(5);
x5 := x_window(6);

if window_counter <= K then
    x6 := signed(mem_regi);
else
     x6 := (others=>'0');
end if;

x_window <= (x0, x1, x2, x3, x4, x5, x6);
```

I primi tre cicli vengono eseguiti "a vuoto", senza effettuare calcoli in modo da caricare correttamente i primi elementi del payload.

2.3.3 Moltiplicazione e accumulazione

In questa fase ogni x viene moltiplicata per il coefficiente corretto e successivamente sommata in mac_accum a seguio di un resize necessario per poter rappresentare ogni possibile risultato.

```
prod0 := x0 * coeffs_sel(0);
prod1 := x1 * coeffs_sel(1);
prod2 := x2 * coeffs_sel(2);
prod3 := x3 * coeffs_sel(3);
prod4 := x4 * coeffs_sel(4);
prod5 := x5 * coeffs_sel(5);
prod6 := x6 * coeffs_sel(6);

mac_accum :=  resize(prod0, 20) +
            resize(prod1, 20) +
            resize(prod2, 20) +
            resize(prod3, 20) +
            resize(prod4, 20) +
            resize(prod5, 20) +
            resize(prod6, 20);
```

2.3.4 Normalizzazione e saturazione

Il calcolo dei risultati si conclude con la normalizzazione (/60 per filtro di ordine 5, /12 per il filtro di ordine 3) seguita dalla normalizzazione dei risultati che eccedono il range [-128,+127], in seguito viene indicato il dato che deve essere salvato in memoria: out_result che è stato assegnato all'uscita o_mem_data.

```
-- NORMALIZATION:
    if s(0) = '0' then
        -- order-3
        t0 := shift(mac_accum, 4);
        t1 := shift(mac_accum, 6);
        t2 := shift(mac_accum, 8);
        t3 := shift(mac_accum,10);
        norm_val := resize(t0, norm_val'length) +
                    resize(t1, norm_val'length) +
                    resize(t2, norm_val'length) +
                    resize(t3, norm_val'length);
    else
        - order-5
        t0 := shift(mac_accum, 6);
        t1 := shift(mac_accum,10);
        norm_val := resize(t0, norm_val'length) +
        resize(t1, norm_val'length);
    end if;

-- SATURATION
    if norm_val > to_signed(127, norm_val'length) then
        sat_result := to_signed(127, 8);
    elsif norm_val < to_signed(-128, norm_val'length) then
        sat_result := to_signed(-128, 8);
    else
        sat_result := resize(norm_val, 8);
    end if;

    out_result <= std_logic_vector(sat_result);
```

3 Risultati sperimentali

3.1 Utilization Report

Site Type	Used	Fixed	Prohibited	Available	Util%
CLB LUTs*	971	0	0	117120	0.83
LUT as Logic	971	0	0	117120	0.83
LUT as Memory	0	0	0	57600	0.00
CLB Registers	269	0	0	234240	0.11
Register as Flip Flop	269	0	0	234240	0.11
Register as Latch	0	0	0	234240	0.00
CARRY8	92	0	0	14640	0.63
F7 Muxes	0	0	0	58560	0.00
F8 Muxes	0	0	0	29280	0.00
F9 Muxes	0	0	0	14640	0.00

Dalla tabella si può osservare che il numero di Flip-Flop utilizzati nella sintesi è di 269 e che non sono stati utilizzati Latch.

3.2 Report Timing

Parametro	Valore
Slack (MET)	9.460 ns (required time - arrival time)
Source	k2_reg[1]/C (FDCE, clocked by clock, rise@0 ns fall@10 ns, period=20 ns)
Destination	out_result_reg[0]/D (FDCE, clocked by clock, rise@0 ns fall@10 ns, period=20 ns)
Path Group	clock
Path Type	Setup (Max at Slow Process Corner)
Requirement	20.000 ns (clock rise@20 ns - clock rise@0 ns)
Data Path Delay	10.402 ns (logic 4.593 ns = 44.155%, route 5.809 ns = 55.845%)
Logic Levels	34 (CARRY8=15, LUT1=1, LUT2=7, LUT3=1, LUT4=2, LUT5=4, LUT6=4)
Clock Path Skew	-0.145 ns (DCD - SCD + CPR)
Destination Clock Delay (DCD)	3.026 ns (23.026 - 20.000)
Source Clock Delay (SCD)	3.502 ns
Clock Pessimism Removal (CPR)	0.332 ns
Clock Uncertainty	0.035 ns $(((TSJ^2 + TIJ^2)^{1/2} + DJ)/2 + PE)$
Total System Jitter (TSJ)	0.071 ns
Total Input Jitter (TIJ)	0.000 ns
Discrete Jitter (DJ)	0.000 ns
Phase Error (PE)	0.000 ns

Dal report, in particolare dallo Slack, si osserva che il risultato viene computato 9,460 ns prima del limite imposto dal clock (20 ns), il segnale è quindi inviato correttamente in uscita senza problemi di tempo.

3.3 Test

I principali testbench utilizzati e il loro risultato sono stati (alcuni dei testi utilizzati provengono dal materiale fornito su weebeep dal professor Gianluca Palermo):

- Con dei test di base si è osservata il corretto funzionamento dei due filtri (di ordine 3 e 5) e la loro selezione
- tb_new ha permesso di verificare che gli estremi dei coefficienti del filtro di ordine 3 venissero ignorati, come da specifica
- testbench mark2 tramite start, reset multipli e consecutivi ha rivelato la necessità di aggiungere lo stato di BUS_HANDOVER per poter gestire casi limite che si possono verificare in tali condizioni, evitando anche eventuali problemi di disallineamento.

4 Conclusioni

4.1 Punti critici affrontati

Nella stesura del progetto VHDL sono state riscontrate criticità principalmente negli ambiti di :

- Complessità nell'allineamento della lettura da memoria sincrona con la scrittura nei registri dedicati ai valori dell'header.
- Complessità nell'allineamento della lettura da memoria sincrona con la scrittura nei registri dedicati ai valori dell'header.
- Difficoltà dell'inserimento nei corretti indirizzi di memoria dei risultati dei calcoli eseguiti dal componente sviluppato.
- Gestione del rischio di leggere dati da indirizzi di memoria non validi, con l'immissione di dati errati nel processo di calcolo.

4.2 Considerazioni finali

Il componente è stato verificato sia tramite testbench sviluppati autonomamente sia mediante i testbench forniti. I risultati ottenuti hanno confermato il corretto funzionamento del progetto in tutte le condizioni previste, si può quindi concludere che l'implementazione rispetta le specifiche richieste