Progetto di Reti Logiche

Matteo Arrigo

06/03/2024

1 Introduzione

Il presente progetto è stato svolto come prova finale del corso di Reti Logiche del terzo anno di Ingegneria Informatica al Politecnico di Milano.

Il compito è progettare un'architettura hardware che soddisfi la specifica funzionale assegnata, sia in pre-sintesi sia in post-sintesi, avendo come unico vincolo non funzionale quello di assicurarsi che i ritardi di propagazione dei segnali non superino il periodo di clock assegnato (20 ns).

L'interfaccia dell'architettura 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;
```

Il sistema inoltre si interfaccia con una memoria indirizzata al byte con 2^{16} parole da 1 Byte.

Le operazioni da fare in sequenza, una volta iniziate, dipendono dal valore `ADD` e `K` dati dagli ingressi `i_add` e `i_k`. Si deve iniziare a leggere la memoria dall'indirizzo `ADD` per un totale di 2K parole, divise in posizioni pari (da dove iniziamo a leggere), che indicano dei valori significativi, e posizioni dispari, che indicano il valore di credibilità del valore significativo precedente. Il compito del sistema è rimpiazzare i valori significativi nulli con l'ultimo valore significativo non-nullo letto, e impostare il suo valore di credibilità in base a questo. Se viene letto un valore non-nullo, il suo valore di credibilità è impostato a `31`, mentre, per ogni valore significativo nullo letto di seguito, la credibilità viene decrementata. Se la credibilità raggiunge `0`, rimane a tale valore senza essere decrementata fino alla lettura del primo valore significativo non-nullo.

Se viene letto un valore significativo nullo all'inizio della sequenza delle operazioni, e finché rimane tale, viene lasciato nullo e impostata a `0` anche la sua credibilità. Per sequenze di operazioni successive e dopo ogni reset, la logica di funzionamento è sempre la stessa.

I segnali `o_mem_addr`, `i_mem_data`, `o_mem_data`, `o_mem_we`, `o_mem_en` servono ad interfacciarsi con la memoria, con gli ovvi significati suggeriti dai nomi, mentre `i_clk` è il segnale di clock che sincronizza tutti gli altri segnali. In particolare, escluso l'unico segnale asincrono `i_rst`, tutti i segnali sono da interpretare sul fronte di salita del clock. Gli altri segnali sono:

1. **`i_rst`**: segnale asincrono che resetta il sistema. Si può alzare `i_start` solo dopo aver abbassato `i_rst`. Il funzionamento del sistema prima della prima volta in cui `i_rst` viene portato alto non è specificato.

2. **i_start**: indica l'inizio delle operazioni. Rimane alto finché `o_done` non diventa '1', dopo di che si deve abbassare. Dopo che anche `o_done` viene abbassato, si può ricominciare un'altra sequenza di operazioni direttamente usando `i_start`, senza dover resettare nuovamente il sistema con `i_rst`.

3. **i_add**: indica il valore ADD della specifica funzionale.

4. **i_k**: indica il valore K della specifica funzionale.

5. **o_done**: segnala la fine della sequenza di operazioni quando diventa '1'. Una volta alzato non si deve più dialogare con la memoria, e si deve abbassare dopo che `i_start` è passato a '0'.

Un esempio di una possibile sequenza di tali operazioni è dato, con ADD=10 e K=10 da:

Addr	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Prima	0	0	128	0	64	0	0	0	0	0	100	0	1	0	0	0	1	0	23	0	0	0
Dopo	0	0	128	31	64	31	64	30	64	29	100	31	1	31	1	30	1	31	23	31	0	0

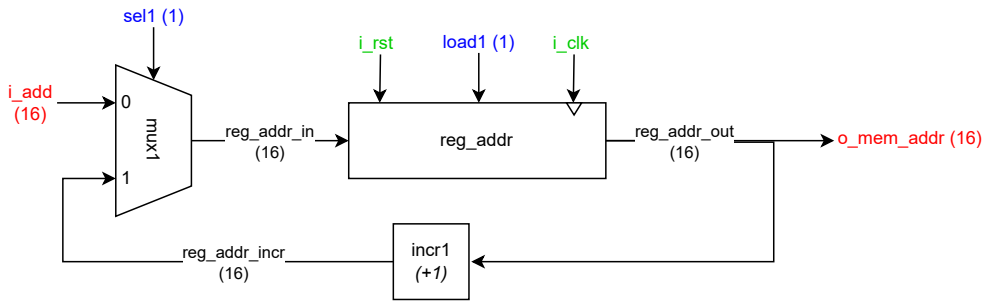
2 Architettura

Di seguito è descritta l'architettura progettata e implementata per il progetto. Il nome dei segnali fa riferimento al nome dato nel codice VHDL dell'architettura principale. Nel complesso, l'architettura è divisa in 3 moduli indipendenti tra loro e una macchina a stati che gestisce i segnali di controllo. I segnali di input/output direttamente gestiti dall'architettura sono stati evidenziati in rosso, mentre i segnali di controllo gestiti dalla FSM (quindi le uscite della FSM) sono state evidenziate in blu. I segnali verdi sono quello di reset e di clock, presenti nei registri, mentre il resto dei segnali non evidenziati sono segnali interni (corrispondenti a `signal` in VHDL). Sono stati indicati tra parentesi il numero di bit dei segnali.

I moduli architetturali di base usati sono (con riferimento al nome delle rispettive entity date in VHDL):

- **mux**: un MUX standard con 2 ingressi da 16 bit, 1 uscita da 16 bit, e un bit di selezione.
- **reg16**: un registro sincrono (flip-flop) da 16 bit, con anche un segnale di reset asincrono e un segnale di load che abilita alla scrittura quando è alto.
- **addr16**: un sommatore da 16 bit implementato tramite un 16 full-adder (usato effettivamente solo come incrementatore o decrementatore).

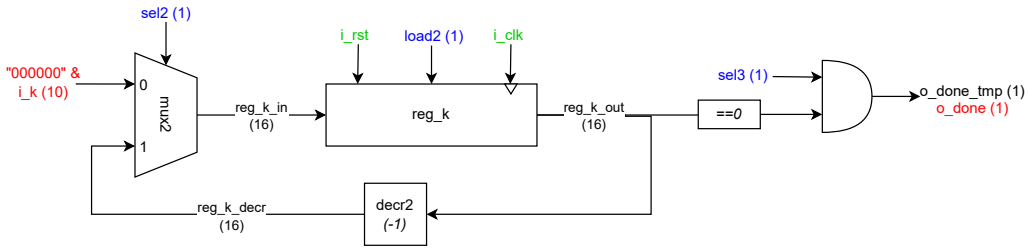
2.1 Modulo 1: Gestore dell'indirizzo di memoria



Il bit `sel1` manda in ingresso registro di `reg_addr` il segnale `i_add` (per l'inizializzazione) o l'uscita del registro stessa dopo che è stata incrementata (quindi l'uscita del modulo `incr1`, che esegue l'operazione +1). La scrittura del registro `reg_addr` è governata dal bit `load1`. L'uscita del registro rappresenta l'uscita `o_mem_addr` di tutta l'architettura.

L'implementazione in VHDL è totalmente Structural.

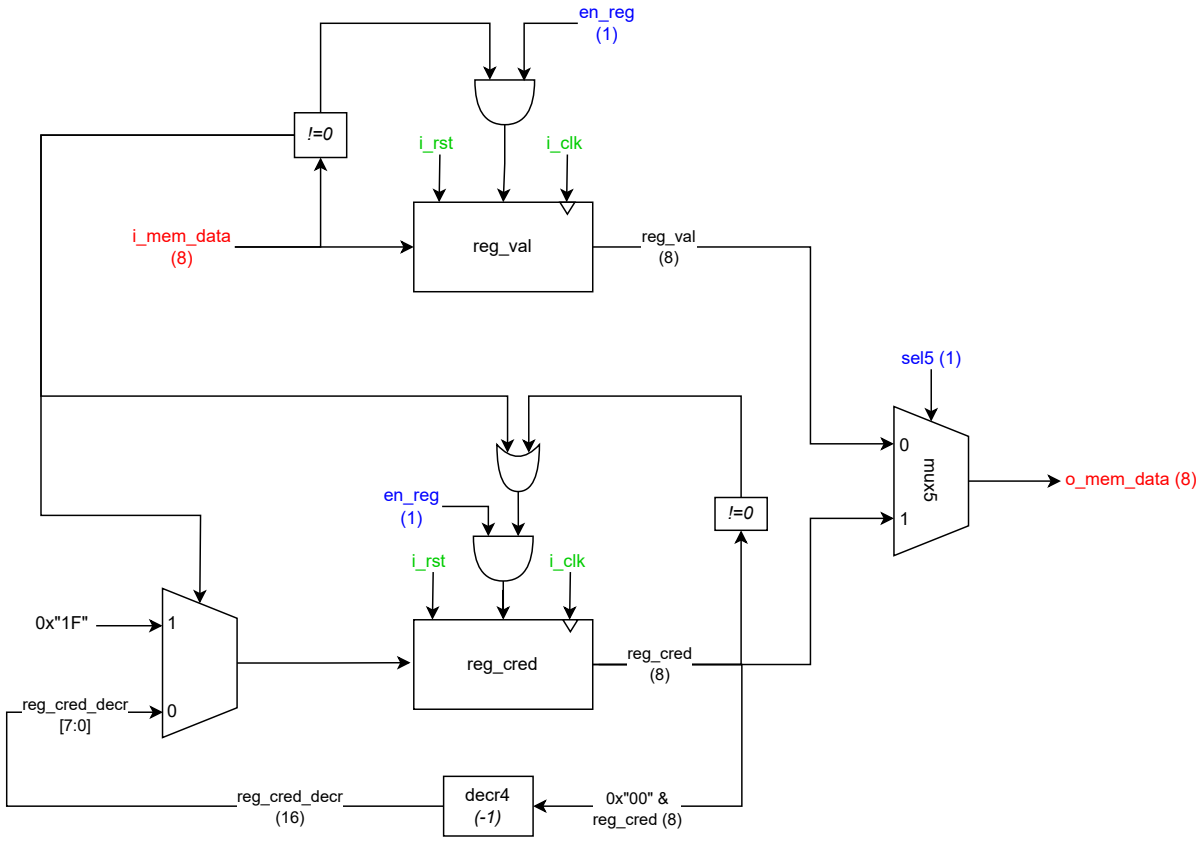
2.2 Modulo 2: Gestore del numero di iterazioni



Il bit `sel2` manda in ingresso al registro `reg_k` il segnale `i_k` (per l'inizializzazione) o l'uscita del registro stessa dopo che è stata decrementata (quindi l'uscita del modulo `decr2`, che esegue l'operazione -1). La scrittura del registro `reg_k` è governata dal bit `load2`. Il modulo `==0` ha in uscita '1' se `reg_k_out` è nullo, '0' altrimenti, e, in AND col bit `sel3`, definisce l'uscita `o_done`.

L'implementazione in VHDL è mista, quasi tutta Structural esclusa una parte Behavioural (quindi un `process`) per definire l'uscita `o_done` in funzione di `reg_k_out` e `sel3`. Il segnale ausiliario `o_done_tmp` serve per permette la lettura di `o_done` (come ingresso delle FSM), che altrimenti sarebbe solo un'uscita (non leggibile) dell'architettura.

2.3 Modulo 3: Gestore dell'interazione con la memoria



Si possono individuare la gestione di 3 parti principali:

- **Registro reg_val**: registro che immagazzina l'ultimo valore non nullo non letto. È abilitato in scrittura

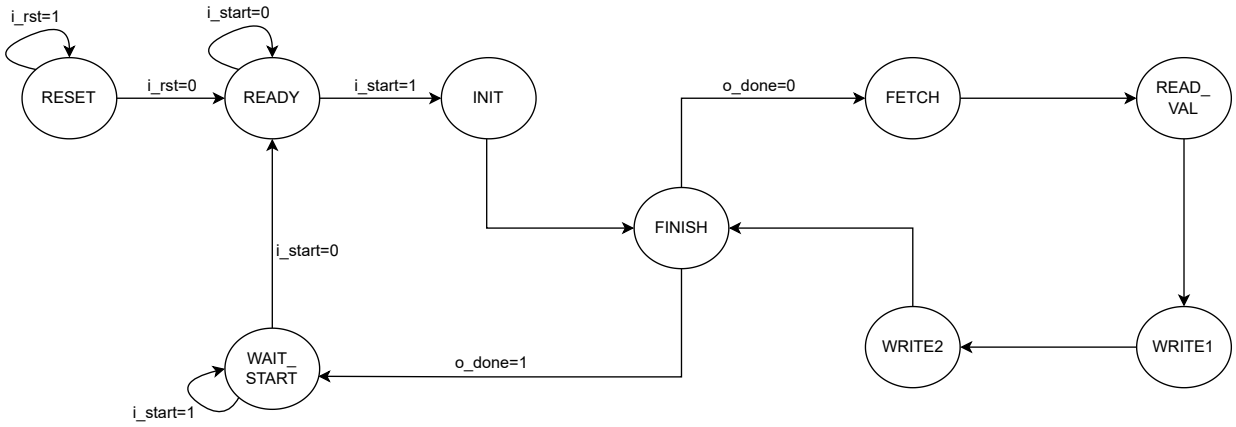
se il bit `en_reg` è alto e il valore `i_mem_data` è non-nullo (serve per memorizzare il valore se è non-nullo, altrimenti viene lasciato in memoria l'ultimo non-nullo).

- **Registro reg_cred**: L'ingresso è il valore 31 (base 10) o la sua uscita decrementata (quindi l'uscita di `decr4`, che esegue l'operazione -1). La scelta dell'ingresso dipende dalla nullità del segnale `i_mem_data` (se il valore letto è non-nullo, impostiamo 31 come valore di credibilità standard, altrimenti decrementiamo l'attuale credibilità). La scrittura del registro avviene se `en_reg` è alto e `i_mem_data` o l'attuale valore di credibilità sono non-nulli (in particolare dobbiamo controllare `reg_cred!=x"00"` per non decrementare ulteriormente la credibilitàquando viene raggiunto il valore nullo).

- **mux5**: sceglie di mandare in uscita (`o_mem_data`) l'uscita del registro `reg_val` o `reg_cred` in base al bit `sel5`.

L'implementazione in VHDL è quasi completamente Behavioural, escluso l'uso dello stile Structural per gestire solo il modulo `decr4`.

2.4 FSM



	load1	sel1	load2	sel2	sel3	en_reg	sel5	o_mem_en	o_mem_we
RESET	0	1	0	1	0	0	0	0	0
READY	0	1	0	1	0	0	0	0	0
INIT	1	0	1	0	0	0	0	0	0
FETCH	0	1	0	1	0	0	0	1	0
READ_VAL	0	1	0	1	0	1	0	1	0
WRITE1	1	1	0	1	0	0	0	1	1
WRITE2	1	1	1	1	0	0	1	1	1
FINISH	0	1	0	1	1	0	0	0	0
WAIT_START	0	1	0	1	1	0	0	0	0

La macchina a stati gestisce i segnali di controllo passando per 9 stati. È stata progettata come macchina di Moore, quindi le uscite si possono indicare su una tabella a parte in funzione solo dello stato corrente. Nella tabella sono stati evidenziate in rosso le uscite diverse da quelle di default per come è stata scritta la FSM nel codice VHDL.

Gli stati sono:

RESET	Stato di reset della FSM, in cui rimane finché `i_rst` non ritorna basso
READY	Attendiamo che `i_start` sia portato alto, per iniziare la sequenza di operazioni
INIT	Impostiamo i valori iniziali dei registri `reg_addr` e `reg_k` per iniziare la sequenza di operazioni
FETCH	Abilitiamo la lettura in memoria per impostare `i_mem_data` con il valore significativo da considerare per questa iterazione
READ_VAL	In base a `i_mem_data`, impostiamo i giusti valori per questa iterazione nei registri `reg_val` e `reg_cred`
WRITE1	Scriviamo in memoria il valore di `reg_val`. Incrementiamo `reg_addr` per prepararci alla scrittura del valore di credibilità
WRITE2	Scriviamo in memoria il valore di `reg_cred`. Incrementiamo `reg_addr` per prepararci alla prossima iterazione Decrementiamo `reg_k` per segnare l'iterazione come avvenuta
FINISH	Stato corrispondente alla fine di un'iterazione, in cui non dialoghiamo con la memoria. Se `i_k` non è diventato 0, iniziamo una nuova iterazione da FETCH. Altrimenti, passiamo a WAIT_START
WAIT_START	Aspettiamo finché `i_start` non ritorna basso

L'implementazione in VHDL è la classica implementazione vista anche a lezione, con 2 processi distinti per gestire la funzione di stato prossimo e di uscita.

3 Risultati Sperimentali

3.1 Sintesi

L'architettura descritta, una volta implementata in VHDL, si può sintetizzare e può essere simulata come se fosse su una vera FPGA (quella usata per la sintesi è quella proposta, l'Artix-7 FPGA xc7a200tfbg484-1), tramite la simulazione funzionale post-sintesi. In particolare possiamo accertarci di alcuni aspetti, basandoci sulla simulazione post-sintesi del testbench dato come esempio per il progetto.

Usando il comando TCL `report_utilization`, si può notare come non siano stati sintetizzati latch, come voluto. Sono anzi generati 52 flip-flop, che il tool di sintesi ha probabilmente usato per gestire i 16 bit dei registri `reg_addr`, `reg_k`, gli 8 bit dei registri `reg_val`, `reg_cred` e 4 bit per gestire i 9 stati della FSM.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	66	0	0	134600	0.05
LUT as Logic	66	0	0	134600	0.05
LUT as Memory	0	0	0	46200	0.00
Slice Registers	52	0	0	269200	0.02
Register as Flip Flop	52	0	0	269200	0.02
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

Inoltre tramite il comando `report_timing`, essendo presente un contraint che impone ai ritardi dei segnali di non superare il periodo di clock di 20ns, si può notare come lo slack time (tempo all'interno del periodo di clock in cui i segnali sono stabili, senza che nessuna operazione sia effettuata) sia ampiamente positivo (16.213 ns). Quindi l'architettura proposta non sembra avere problemi per quanto riguarda i ritardi dei segnali.

```
Timing Report

Slack (MET) :             16.213ns  (required time - arrival time)
  Source:                 reg_cred_reg[4]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            reg_cred_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:            20.000ns  (clock rise@20.000ns - clock rise@0.000ns)
  Data Path Delay:        3.405ns  (logic 1.019ns (29.927%)  route 2.386ns (70.073%))
  Logic Levels:           3  (LUT4=1 LUT5=1 LUT6=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
```

3.2 Simulazione

3.2.1 Testbench 1

K = 36, ADD = 100

Addr	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
Prima	33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dopo	33	31	33	30	33	29	33	28	33	27	33	26	33	25	33	24	33	23	33	22
Addr	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139
Prima	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dopo	33	21	33	20	33	19	33	18	33	17	33	16	33	15	33	14	33	13	33	12
Addr	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
Prima	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dopo	33	11	33	10	33	9	33	8	33	7	33	6	33	5	33	4	33	3	33	2
Addr	160	161	162	163	164	165	166	167	168	169	170	171								
Prima	0	0	0	0	0	0	0	0	1	1	0	1								
Dopo	33	1	33	0	33	0	33	0	1	31	1	30								

Questo testbench Si occupa principalmente di testare il corretto decremento di `reg_k`, che parte da 31 (col primo valore non-nullo 33 all'indirizzo 100) e viene decrementato fino a 0, valore in cui rimane per 3 iterazioni. Inoltre si vede anche che valori non nulli agli indirizzi destinati alle credibilità non danno problemi. Comunque, essendo il primo testbench presentato, vengono correttamente testati il comportamento generale, la gestione del comportamento basato sui segnali `i_start` e `i_rst`, e la corretta gestione dei segnale `o_done`, `o_mem_en`, `o_mem_we` (per cui ci sono delle corrispondenti `assert` nel testbench)

3.2.2 Testbench 2

K = 7, ADD = 65522

Addr	65522	65523	65524	65525	65526	65527	65528	65529	65530	65531	65532	65533	65534	65535
Prima	0	0	0	1	66	0	66	0	66	0	0	0	65	0
Dopo	0	0	0	0	66	31	66	31	66	31	66	30	65	31

Questo testbench testa il caso il cui si trovano valori significativi iniziali nulli. C'è anche il caso particolare in cui più valori significativi non-nulli consecutivi sono uguali (per cui, anche se alla fine ricompare lo stesso valore, la credibilità rimane 31). Inoltre è verificato il caso limite in cui si raggiunge la fine della memoria.

3.2.3 Testbench 3

K = 0, ADD = 100

Addr	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
Prima	33	0	33	0																
Dopo	33	0	33	0																

Questo testbench controlla il caso limite in cui K=0, per cui non ci devono essere iterazioni. (In questo caso, la prima volta che la FSM arriva allo stato FINISH, o_done è già alto e quindi si va subito nello stato WAIT_START, senza fare iterazioni).

3.2.4 Testbench 4

ADD1 = 100, K1 = 5; ADD2 = 110, K2 = 5

Addr	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
Prima	255	1	52	0	11	0	92	0	22	0	2	0	0	0	12	0	39	0	31	0
Dopo1	255	31	52	31	11	31	92	31	22	31	2	0	0	0	12	0	39	0	31	0
Dopo2	255	31	52	31	11	31	92	31	22	31	2	31	2	30	12	31	39	31	31	31

Questo testbench controlla la corretta gestione di 2 sequenze di operazioni consecutive, senza sovrapposizione dei segmenti di memoria gestiti. La seconda sequenza è fatta partire senza prima resettare la macchina.

3.2.5 Testbench 5

ADD1 = 100, K1 = 8; ADD2 = 104, K2 = 8

Addr	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
Prima	33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dopo1	33	31	33	30	33	29	33	28	33	27	33	26	33	25	33	24	0	0	0	0
Dopo2	33	31	33	30	33	31	33	31	33	31	33	31	33	31	33	31	33	30	33	29

Questo testbench controlla la corretta gestione di 2 sequenza di operazioni consecutive, con sovrapposizione parziale dei segmenti di memoria gestiti. La seconda sequenza è fatta partire dopo il reset della macchina

4 Conclusioni

I testbench proposti (che comunque non sono gli unici ad essere stati usati per la fase di testing) sembrano coprire tutti i casi di interesse, senza essere ridondanti. Dato il fatto che l'architettura si comporta come aspettato sia in pre- sia in post-sintesi, è ragionevole assumere che la stessa sia stata correttamente progettata ed implementata.

Sono conscio che l'implementazione in VHDL proposta non è la più agevole da scrivere o la più immediata da leggere, essendoci diverse parti Structural e sommatori che potevano essere lasciati da implementare a VHDL, ma ho scritto l'implementazione così come è volutamente, in modo da esplorare quanto più possibile VHDL stesso secondo i metodi esposti in aula.