

Progetto di Reti Logiche 2025

Diego Fernando Huaman Diego

(Codice Persona: 10891526 - Matricola: 215587)

Novembre 2025

Indice

1	Introduzione	3
1.1	Obiettivo	3
1.2	Specifiche generali	3
1.3	Protocollo di comunicazione	4
1.4	Interfaccia del modulo	5
2	Architettura	6
2.1	Segnali interni	6
2.1.1	Registri	6
2.1.2	Segnali interni del filtro differenziale	6
2.1.3	Altri segnali	7
2.2	Descrizione processi	7
2.2.1	Processo del Registro base_addr	7
2.2.2	Processo del Registro count	8
2.2.3	Processo del Registro k	8
2.2.4	Processo del Registro s	8
2.2.5	Processo del Registro a scorrimento C	8
2.2.6	Processo del Registro a scorrimento K	9
2.2.7	Processo per Calcolo rk_sel	9
2.2.8	Processo per Attivazione Load	9
2.2.9	Processo per Scelta Operazione	9
2.2.10	Processo di Saturazione Valore Normalizzato	9
2.2.11	Processo Stato corrente	9
2.2.12	Processo Funzione Stato Prossimo	10
2.2.13	Processo Funzione di Uscita	10
2.3	Descrizione Stati	10
2.3.1	Stato S0 - Reset	10
2.3.2	Stato S1 - Inizializzazione Registri	10
2.3.3	Stato S2 - Lettura	11
2.3.4	Stato S3 - Attesa del Dato	11
2.3.5	Stato S4 - Stabilizzazione Risultati	11
2.3.6	Stato S5 - Scrittura	12
2.3.7	Stato S6 - Termine Computazione	12

3	Risultati sperimentali	13
3.1	Report Di Sintesi	13
3.2	Simulazioni	14
3.2.1	Risultato fuori banda	14
3.2.2	Correttezza del filtro di ordine 3 in presenza di estremi non nulli	14
3.2.3	Somma di prodotti molto grandi	15
3.2.4	Sequenza massima di valori nel campo K ($k = 32759$)	16
3.2.5	Comportamento all'ultimo indirizzo disponibile in memoria . . .	16
3.2.6	Reset asincrono	17
3.2.7	Elaborazione multipla	17
4	Conclusioni	19

1 Introduzione

1.1 Obiettivo

Lo scopo del progetto è quello di implementare un modulo hardware (in VHDL) in grado di interfacciarsi con una memoria. In particolare, il sistema:

1. legge i primi byte per configurare il filtro differenziale;
2. legge, un byte alla volta, un messaggio in memoria costituito da una sequenza di K parole W , il cui valore appartiene all'intervallo $[-128, 127]$ ed è espresso in complemento a 2;
3. applica il filtro differenziale;
4. salva la sequenza di K parole R in memoria. I valori di R fuori dall'intervallo $[-128, 127]$ sono saturati al limite definito dall'intervallo.

1.2 Specifiche generali

Tutti i dati da utilizzare sono memorizzati sequenzialmente a partire da un indirizzo specificato (ADD). I dati iniziali sono i seguenti:

- **17 (2+1+14) byte**, utilizzati per configurare il filtro differenziale. In particolare:
 - **K1** e **K2**, due byte, indicano la lunghezza della sequenza K dei dati W . **K1** è il byte più significativo, mentre **K2** è quello meno significativo. Si ipotizza che il minimo valore che può assumere $K1K2$ è 7.
 - **S**, un byte, definisce quale filtro utilizzare tra quello di ordine 3 e quello di ordine 5. In particolare:
 - * se il bit meno significativo è 0, si utilizza il filtro di ordine 3;
 - * altrimenti, si utilizza il filtro di ordine 5.
 - **Da C1 a C14**, quattordici byte, si memorizzano i coefficienti dei due filtri. In particolare:
 - * i primi 7 byte (C1..C7) definiscono i coefficienti del filtro di ordine 3;
 - * gli ultimi 7 byte (C8..C14), invece, definiscono i coefficienti del filtro di ordine 5.
- **K byte** da elaborare, che contengono valori da -128 a +127.

Il filtro calcola la seguente funzione:

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

I valori di l e n variano in base all'ordine del filtro e sono i seguenti:

Parametro	Ordine 3	Ordine 5
l	2	3
n	12	60

La sequenza K del risultato R viene riscritta a partire dall'indirizzo $\text{ADD}+17+K$, cioè alla fine della sequenza K dei valori W. Quindi, la memoria utilizzata dal modulo sarà strutturata nel seguente modo:

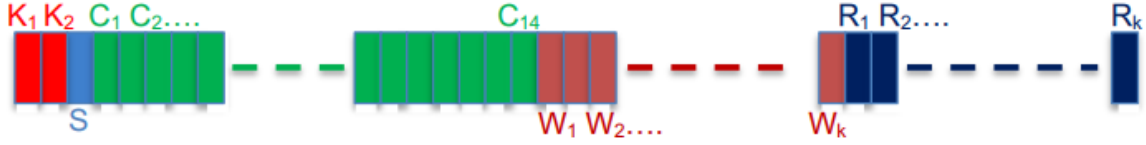


Figura 1: Sequenza di byte da gestire in memoria

I valori fuori dalla sequenza sono da considerare pari a 0. Inoltre, ogni valore ottenuto deve essere normalizzato dividendolo per il coefficiente n . La divisione deve essere approssimata nel seguente modo:

$$\frac{1}{12} \approx \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \frac{1}{1024}, \quad \frac{1}{60} \approx \frac{1}{64} + \frac{1}{1024}$$

Si osserva che ogni shift a destra equivale a una divisione per due e produce un troncamento verso -1 per numeri negativi e verso 0 per i numeri positivi. Quindi, solo quando il valore da shiftare è negativo, si aggiunge +1 al risultato del singolo shift.

1.3 Protocollo di comunicazione

La comunicazione tra il modulo e la memoria avviene secondo un protocollo definito nel seguente modo:

- Prima del primo `i_start = 1` (e prima di richiedere il corretto funzionamento del metodo), viene sempre dato il segnale di RESET (`i_rst = 1`).
- Quando `i_rst = 0`, il modulo inizia l'elaborazione quando `i_start = 1`. Il segnale di START rimane alto fino a quando il segnale di DONE è basso.
- Una volta terminata la computazione, il modulo pone `o_done = 1`. Il segnale di DONE rimane alto fino a quando il segnale di START non si abbassa.
- Una successiva elaborazione con `i_start = 1` non deve attendere il reset del modulo.
- Ogni volta che viene dato il segnale di RESET (`i_rst = 1`), il modulo deve essere re-inizializzato.

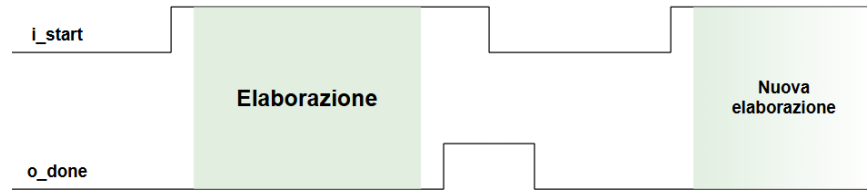


Figura 2: Protocollo di comunicazione

1.4 Interfaccia del modulo

L'interfaccia del modulo hardware è definita nel seguente modo:

- **i_clk**: segnale di CLOCK in ingresso generato dal TestBench.
- **i_rst**: segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START.
- **i_start**: segnale di START generato dal TestBench.
- **i_add**: vettore ADD generato dal TestBench che rappresenta l'indirizzo da cui parte la sequenza da elaborare.
- **o_done**: segnale di DONE in uscita che comunica la fine dell'elaborazione.
- **o_mem_addr**: vettore di uscita che manda l'indirizzo alla memoria.
- **i_mem_data**: vettore che arriva dalla memoria e che contiene il dato in seguito ad una richiesta di lettura.
- **o_mem_data**: vettore che va verso la memoria e che contiene il dato che viene successivamente scritto.
- **o_mem_en**: segnale di ENABLE da inviare alla memoria per poter comunicare (sia in lettura che in scrittura).
- **o_mem_we**: segnale di WRITE ENABLE che definisce una scrittura su memoria se è alto, altrimenti una lettura da memoria.

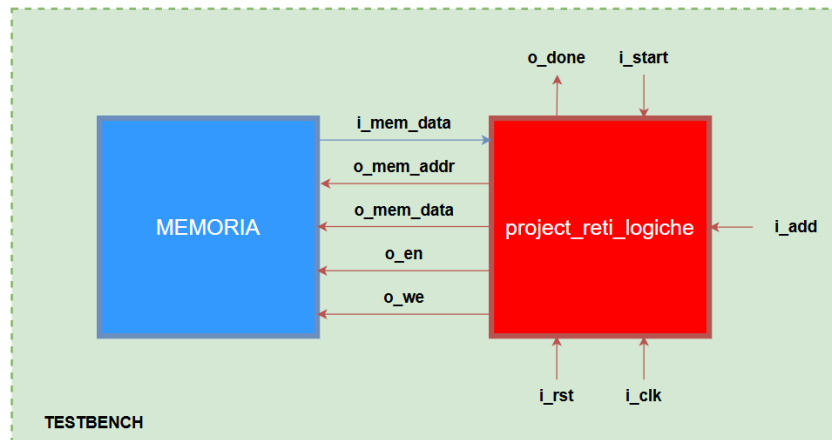


Figura 3: Interfaccia del modulo dal punto di vista grafico

2 Architettura

2.1 Segnali interni

2.1.1 Registri

I registri utilizzati dal modulo hardware sono i seguenti:

- **base_addr**: 16 bit, contiene l'indirizzo inserito in **i_add**.
- **count**: 16 bit, contiene il numero di letture svolte.
- **k**: 16 bit, contiene il valore di K1K2.
- **s**: 1 bit, contiene il bit meno significativo del campo S.
- **c2..c6**, **c7..c14**: (ciascuno) 8 bit, contiene il byte del campo C in posizione i .
- **k1..k7**: (ciascuno) 8 bit, contiene il byte del campo K che contribuisce al calcolo della funzione differenziale e che copre attualmente la posizione $i = 1..7$.

Importante: Dal punto di vista hardware, i registri **c2..c14** e **k1..k7** sono organizzati come uno shift register SIPO. Il segnale di ingresso è caricato rispettivamente in **c14** e **k7**.

2.1.2 Segnali interni del filtro differenziale

I segnali interni utilizzati dal filtro differenziale sono i seguenti:

- **cc1..cc7**: (ciascuno) 8 bit, contiene il coefficiente in posizione $i = 1..7$ da usare nel filtro differenziale.
- **p1..p7**: (ciascuno) 19 bit, contiene il prodotto $cc_i * k_i$. Il numero di bit utilizzati è tale che la somma di tutti i prodotti non generi mai overflow.
- **sum**: 19 bit, contiene la somma di tutti i prodotti.
- **ovf**: 19 bit, è il valore da sommare ad ogni shift destro. Il bit meno significativo è il bit più significativo di **sum**, mentre gli altri sono posti a 0.
- **res_o3**: 19 bit, è il risultato ottenuto con la normalizzazione ordine 3.
- **res_o5**: 19 bit, è il risultato ottenuto con la normalizzazione ordine 5.
- **res**: 19 bit, è il segnale di uscita del MUX che seleziona il risultato con la normalizzazione richiesta.

2.1.3 Altri segnali

Il modulo hardware utilizza anche i seguenti segnali interni:

- **init_count**: 1 bit. Quando posto a 1, inizializza il valore del contatore a 0x0000.
- **addr_read**: 16 bit, contiene l'indirizzo di memoria dove svolgere la prossima lettura.
- **addr_write**: 16 bit, contiene l'indirizzo di memoria dove svolgere la prossima scrittura.
- **ra_load**: 1 bit. Quando posto a 1, si carica l'ingresso nel registro **base_addr**.
- **c_load**: 1 bit. Quando posto a 1, si incrementa di 1 il valore del registro **count**.
- **rlength_load**: 1 bit. Quando posto a 1, si carica l'ingresso nel registro **k**.
- **rs_load**: 1 bit. Quando posto a 1, si carica il bit meno significativo dell'ingresso nel registro **s**.
- **rc_load**: 1 bit. Quando posto a 1, si carica l'ingresso nello shift register **C**.
- **rk_load**: 1 bit. Quando posto a 1, si carica l'ingresso nello shift register **K**.
- **o_sel**: 1 bit, è l'ingresso di selezione del MUX che seleziona l'indirizzo da caricare in **o_mem_addr**.
- **rk_sel**: 1 bit, è l'ingresso di selezione del MUX che seleziona il valore da caricare nello shift register **K** tra quello presente in **i_mem_data** e il valore 0x00.
- **in_k**: 8 bit, contiene il valore da caricare nello shift register **K**.
- **load_en**: 1 bit. Abilita il circuito che definisce il segnale di **LOAD** da attivare.
- **wait_res**: 1 bit. Quando posto a 1, il valore calcolato dal filtro differenziale è valido.
- **o_end**: 1 bit. Quando posto a 1, la computazione è terminata.

2.2 Descrizione processi

Il componente è realizzato con 13 processi, di cui 6 sono utilizzati per i registri, 3 per la FSM e 4 per implementare la specifica. Tutti i registri aggiornano il loro valore sul fronte di salita del **CLOCK** solo se il loro segnale di **LOAD** è alto.

Inoltre, in caso di **RESET** (**i_rst** = 1), tutti i segnali vengono riportati al loro valore di default in modo asincrono.

2.2.1 Processo del Registro **base_addr**

Quando **ra_load** = 1, il processo aggiorna il contenuto del registro **base_addr** sul fronte di salita del **CLOCK**. Il valore è utilizzato per determinare i successivi indirizzi da utilizzare per la lettura e la scrittura.

2.2.2 Processo del Registro count

Quando `c_load = 1`, il processo aggiorna sul fronte di salita del CLOCK il contenuto del contatore, incrementandolo di 1. Il contatore è utilizzato per diverse operazioni, in particolare:

- per calcolare i successivi indirizzi di lettura e scrittura;
- come condizione per determinare il segnale di LOAD da attivare;
- come condizione per determinare il valore dei segnali di controllo `wait_res`, `rk_sel` e `o_end`.

2.2.3 Processo del Registro k

Quando `rlength_load = 1`, il processo aggiorna sul fronte di salita del CLOCK il contenuto del registro `k`. In particolare:

- si copia il byte più significativo in quello meno significativo;
- si carica l'ingresso nel byte più significativo.

Il valore di `k` è utilizzato:

- per calcolare il successivo indirizzo di scrittura;
- come condizione per determinare il valore dei segnali di controllo `wait_res`, `rk_sel` e `o_end`.

2.2.4 Processo del Registro s

Quando `rs_load = 1`, il processo aggiorna sul fronte di salita del CLOCK il contenuto del registro `s`. Il valore di `s` è utilizzato come ingresso di selezione per i seguenti MUX:

- MUX che seleziona i coefficienti da utilizzare per il filtro differenziale (`s = 0` per ordine 3, `s = 1` per ordine 5);
- MUX che seleziona il risultato da inviare alla saturazione tra quello ottenuto con normalizzazione 12 (`s = 0`) e quello ottenuto con normalizzazione 60 (`s = 1`).

2.2.5 Processo del Registro a scorrimento C

Quando `rc_load = 1`, il processo aggiorna sul fronte di salita del CLOCK il contenuto dello shift register `C`. Nello shift register, i coefficienti `c1` e `c7` del filtro di ordine 3 non sono caricati perché il modulo non li utilizza attivamente. Per evitare distinzioni di progettazione dovute dall'ordine del filtro, il filtro differenziale pone a `0x00` i coefficienti del filtro di ordine 3 in posizione 1 e 7.

2.2.6 Processo del Registro a scorrimento K

Quando `rk_load = 1`, il processo aggiorna sul fronte di salita del `CLOCK` il contenuto dello shift register `K`. Visto che lo shift register è SIPO, ogni volta che si inserisce un nuovo valore, si perde il valore caricato per primo. Questa struttura è adatta a rappresentare il calcolo con il filtro differenziale se si carica il nuovo valore nel registro associato all'ultimo dei coefficienti utilizzati, cioè `k7`.

2.2.7 Processo per Calcolo `rk_sel`

Sulla base del valore del contatore, il processo inserisce nel segnale `rk_sel`:

- il valore 0, se ci sono ancora dati non letti nello stream;
- il valore 1, altrimenti.

2.2.8 Processo per Attivazione Load

Quando `load_en = 1`, il processo attiva la struttura dove caricare il dato al successivo ciclo di `CLOCK`. Per fare ciò, alza uno dei segnali di `LOAD` associati ai vari campi sulla base del valore di `count`.

2.2.9 Processo per Scelta Operazione

Sulla base del valore del contatore, il processo determina l'operazione da svolgere tramite i segnali `wait_res` e `o_end`. In particolare:

- per i primi 20 valori, la successiva operazione è una **lettura**: quindi, `wait_res = 0` e `o_end = 0`;
- dopo aver memorizzato l'ultimo calcolo, la successiva operazione è la comunicazione di **termine computazione**: quindi, `o_end = 1`. Solo in questo caso, il valore di `wait_res` è indifferente;
- negli altri casi, la successiva operazione è una **scrittura**: quindi, `wait_res = 1` e `o_end = 0`.

2.2.10 Processo di Saturazione Valore Normalizzato

Il processo satura a 8 bit il risultato ottenuto dal filtro differenziale (a 19 bit) e lo inserisce nell'uscita `o_mem_data`.

2.2.11 Processo Stato corrente

Il processo aggiorna lo stato corrente del modulo hardware nel seguente modo:

- se `i_rst = 1`, lo stato corrente è quello di reset `S0`;
- altrimenti, ad ogni ciclo di `CLOCK`, lo stato corrente è quello contenuto in `next_state` all'istante precedente.

2.2.12 Processo Funzione Stato Prossimo

Il processo definisce la funzione stato prossimo δ del modulo hardware.

2.2.13 Processo Funzione di Uscita

Il processo definisce la funzione di uscita λ del modulo hardware.

2.3 Descrizione Stati

Il modulo hardware è realizzato con una Macchina a Stati Finiti (FSM - Finite State Machine) di Moore, composta da **7 stati**. La struttura della FSM è la seguente:

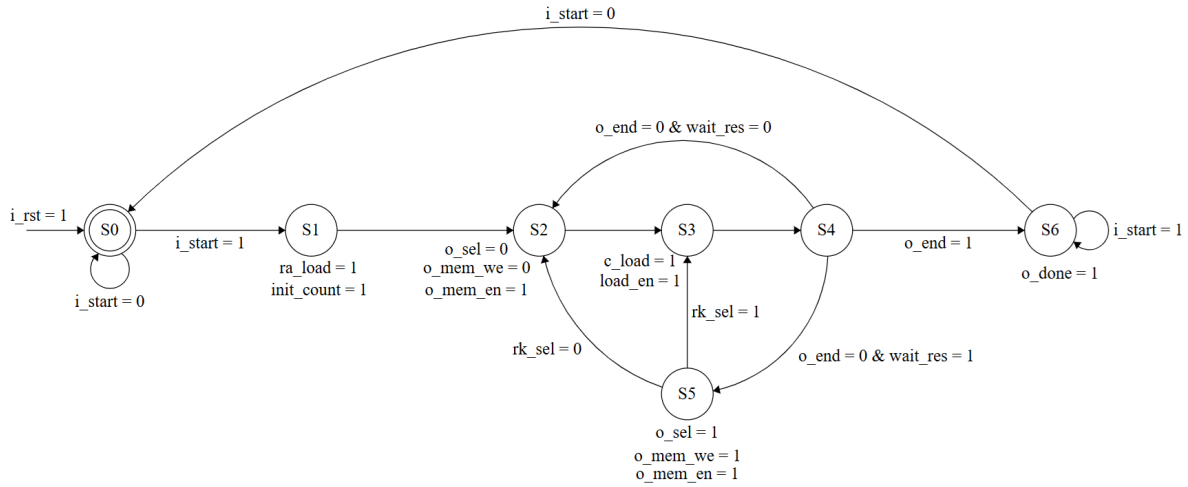


Figura 4: Macchina a Stati Finiti

2.3.1 Stato S0 - Reset

Lo stato S0 è lo stato di reset, dove il modulo si porta una volta ricevuto $i_rst = 1$ e rimane fino a quando non riceve la prima richiesta di elaborazione, cioè $i_start = 0$.

2.3.2 Stato S1 - Inizializzazione Registri

Quando il modulo riceve la richiesta di elaborazione ($i_start = 1$), l'ingresso i_addr ha un valore valido e il modulo passa dallo stato S0 allo stato S1. Lo stato S1 si occupa di inizializzare il modulo prima di svolgere la prima lettura. In particolare, nello stato S1, si pongono:

- **init_count = 1:** si inizializza il contatore a 0x0000. L'inizializzazione del contatore porta a inizializzare al loro valore di default (0) anche i segnali di controllo rk_sel , o_end e $wait_res$;
- **ra_load = 1:** si carica l'indirizzo in i_addr nel registro **base_addr**.

Importante: I registri relativi ai campi K1K2, S, C e K non sono portati al loro valore di default perché verranno sovrascritti con i nuovi valori prima del primo calcolo valido.

2.3.3 Stato S2 - Lettura

Lo stato S2 svolge l'operazione di lettura dalla memoria all'indirizzo inserito in `o_mem_addr`. Nello stato S2, si pongono:

- `o_sel = 0`: si carica in `o_mem_addr` l'indirizzo da cui leggere il dato (`addr_read`). Questo indirizzo è ottenuto nel seguente modo:

$$addr_read = base_addr + count;$$

- `o_mem_we = 0` e `o_mem_en = 1`: si abilita l'operazione di lettura all'indirizzo in `addr_read`.

2.3.4 Stato S3 - Attesa del Dato

Una volta inviata la richiesta di lettura, il modulo deve aspettare un ciclo di CLOCK (rappresentato come uno stato) prima di utilizzare il dato. Nello stato S3, si pongono:

- `c_load = 1`: allo stato successivo (S4), si incrementa il contatore di 1;
- `load_en = 1`: si attiva il segnale di LOAD della struttura in cui verrà caricato il dato.

2.3.5 Stato S4 - Stabilizzazione Risultati

Una volta ricevuto il dato dalla memoria, il modulo passa dallo stato S3 allo stato S4. Nello stato S4:

- si carica il dato letto dalla memoria nella struttura a registri opportuna;
- si stabilizzano il valore del contatore appena incrementato e il valore dell'uscita calcolato con il circuito combinatorio che realizza la funzione differenziale;
- si determina l'operazione da svolgere al ciclo di CLOCK successivo.

Lo stato successivo è determinato dai segnali `o_end` e `wait_res`. In particolare:

- se `o_end = 1`, la computazione è terminata: quindi, lo stato successivo è quello di Termine Computazione S6;
- altrimenti:
 - se `wait_res = 0`, la prossima operazione è la lettura di un nuovo dato dalla memoria: quindi, lo stato successivo è quello di Lettura S2;
 - se `wait_res = 1`, la prossima operazione è la scrittura sulla memoria: quindi, lo stato successivo è quello di Scrittura S5.

2.3.6 Stato S5 - Scrittura

Lo stato S5 svolge l'operazione di scrittura del risultato sulla memoria all'indirizzo inserito in `o_mem_addr`. Nello stato S2, si pongono:

- `o_sel = 1`: si carica in `o_mem_addr` l'indirizzo su cui scrivere il risultato (`addr_write`). Questo indirizzo è ottenuto nel seguente modo:

$$addr_write = add_read + (k - 4) = (base_addr + count) + (k - 4)$$

Il risultato dell'operazione $k-4$ può generare underflow prima della seconda lettura dall'inizio dell'elaborazione, in quanto il valore di K1K2 nello stream di dati non è ancora caricato interamente. Dal primo calcolo della funzione differenziale, però, il modulo non è affetto da questo problema, in quanto il campo K1K2 è completamente memorizzato e $k > 4$ per ipotesi.

- `o_mem_we = 1` e `o_mem_en = 1`: si abilita l'operazione di scrittura sull'indirizzo in `addr_write`.

Lo stato successivo è determinato dal segnale `k_sel`. In particolare:

- se `k_sel = 0`, cioè ci sono ancora dati da leggere nello stream, si passa allo stato di Lettura S2;
- se `k_sel = 1`, invece, si passa allo stato di Attesa del Dato S3.

Il modulo è progettato in modo tale da svolgere solo le letture e le scritture necessarie.

2.3.7 Stato S6 - Termine Computazione

Una volta terminata la computazione, il modulo hardware passa dallo stato S4 allo stato S6, in cui si pone `o_done = 1`. A questo punto, il modulo:

- rimane nello stato S6 fino a quando `i_start = 1`;
- passa allo stato di Reset S0 una volta che `i_start = 0`.

3 Risultati sperimentali

3.1 Report Di Sintesi

I Report di Sintesi sono ottenuti considerando la FPGA xc7k70tfbv676-1 della famiglia Kintex-7.

La Tabella 1 mostra le informazioni ottenute dal Timing Report generato con il comando `report_timing` in *Post-Synthesis Functional*. Il modulo ha uno Slack Time pari a **17.023 ns**, margine che soddisfa ampiamente il vincolo di periodo di 20 ns.

Il percorso più lungo è quello dal registro `count` al registro `c10` dello shift register `C`. Esso attraversa **tre livelli logici** (un LUT4 e due LUT6) e impiega un tempo pari a **2.555 ns**, che è distribuito tra:

- **Logic Delay** (ritardo speso dentro la logica): 0.528 ns (20.665%);
- **Route Delay** (ritardo speso nei collegamenti fisici): 2.027 ns (79.335%).

Per evitare problemi di timing, quindi, il periodo di clock deve essere di almeno 3 ns.

Parameter	Value
Requirements	20.000 ns
Data Path Delay	2.555 ns
Logic	0.528 ns (20.665%)
Route	2.027 ns (79.335%)
Logic Levels	3 (LUT4 = 1 LUT6 = 2)
Critical Path	count_reg[7]/C → c10_reg[0]/CE
Slack time	17.023 ns (Required Time - Arrival Time)

Tabella 1: Informazioni generali

Parameter	Value
Clock Rise Destination	20.000 ns
Destination Clock Delay (DCD)	1.860 ns
Clock Pessimism Removal (CPR)	0.112 ns
Clock Uncertainty (CU)	-0.035 ns
Setup Time (FDCE Setup_C_CE)	-0.242 ns
Required Time	21.695 ns

Tabella 2: Calcolo di Required Time

Parameter	Value
Data Path Delay (Logic + Route)	2.555 ns
Source Clock Delay (SCD)	2.117 ns
Arrival Time	4.672 ns

Tabella 3: Calcolo di Arrival Time

La Tabella 2 mostra le informazioni ottenute dallo Slice Logic con il comando `report_utilization` in *Post-Synthesis Functional*. Si osserva che il modulo non implementa alcun *Latch*, ma utilizza un numero di *Flip Flop* pari a **207** e un numero di *LUT* pari a **863**.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	863	0	41000	2.10
LUT as Logic	863	0	41000	2.10
LUT as Memory	0	0	13400	0.00
Slice Registeres	207	0	82000	0.25
Register as Flip Flop	207	0	82000	0.25
Register as Latch	0	0	82000	0.00
F7 Muxes	0	0	20500	0.00
F8 Muxes	0	0	10250	0.00

Tabella 4: Slice Logic

3.2 Simulazioni

3.2.1 Risultato fuori banda

Questo test verifica che i risultati fuori banda siano saturati a 8 bit, cioè all'interno dell'intervallo $[-127, 128]$. La saturazione è possibile tramite un processo applicato al risultato normalizzato del filtro differenziale. Si osserva che l'operazione è svolta correttamente sia nel caso di saturazione positiva (valore maggiore di $+127$) sia nel caso di saturazione negativa (valore minore di -128).

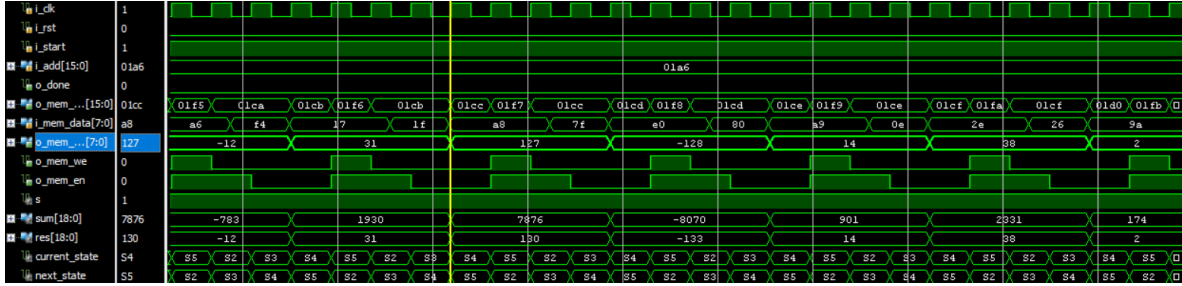


Figura 5: Saturazione positiva e negativa

3.2.2 Correttezza del filtro di ordine 3 in presenza di estremi non nulli

Per definizione, il filtro di ordine 3 considera solo i coefficienti che hanno distanza al più $l = 2$ dal suo elemento centrale: quindi, il valore dei coefficienti `c1` e `c7` è indifferente. Il modulo gestisce questo caso leggendo i coefficienti ma non memorizzandoli, in quanto non sarebbero utilizzati attivamente. L'ottimizzazione di questa parte porta a un'inutile complicazione del circuito.

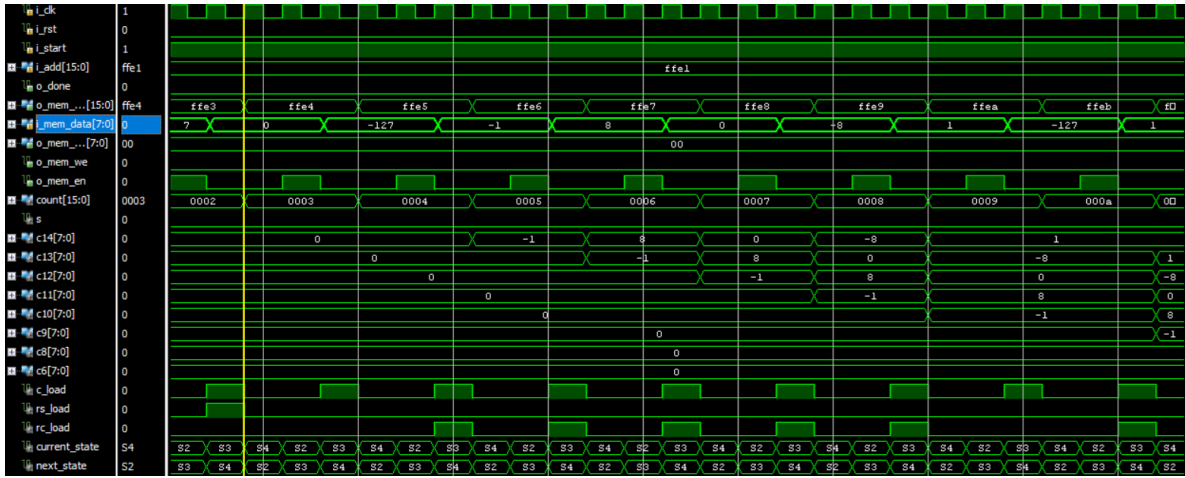


Figura 6: Comportamento del modulo nella lettura di c1 e c7 (con valore -127)

3.2.3 Somma di prodotti molto grandi

Questo test verifica che il valore restituito dalla funzione differenziale è corretto anche in presenza di prodotti molto grandi. Dal punto di vista hardware, il numero di bit dei prodotti ($p1..p7$) e del risultato (res_o3 , res_o5 , res) è tale da coprire il valore massimo e minimo ottenibile dal filtro differenziale senza normalizzazione ed evitare, quindi, errori di overflow ed errori durante lo shift a destra.

In questo modo, il risultato restituito prima della normalizzazione è corretto anche in presenza di prodotti molto grandi.

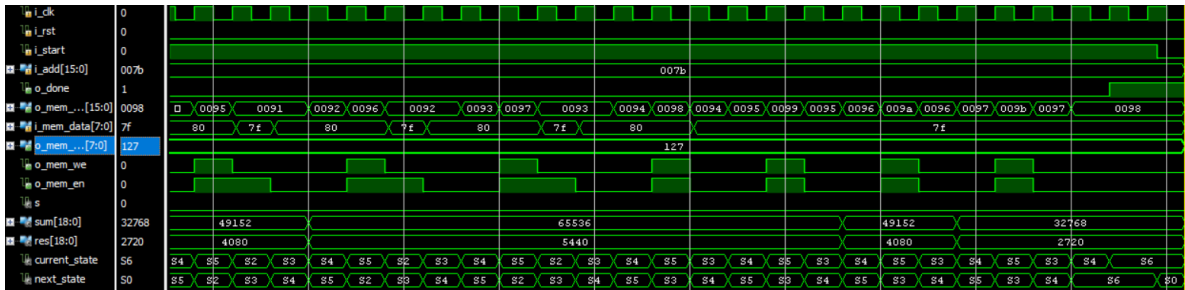


Figura 7: Valore massimo ottenibile con ordine 3

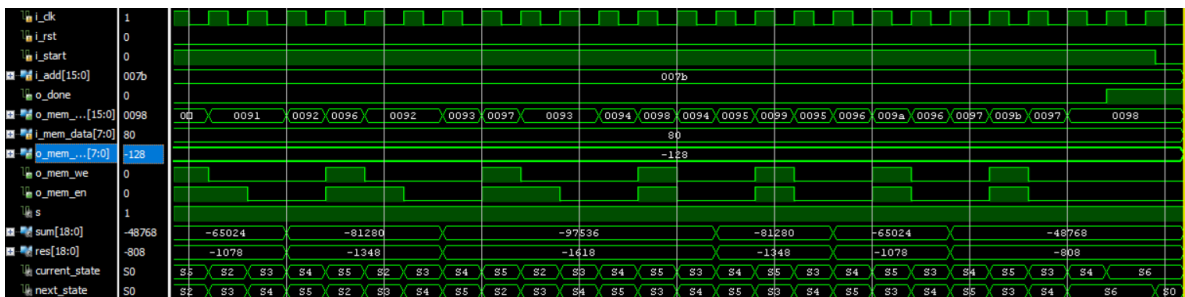


Figura 8: Valore minimo ottenibile con ordine 5

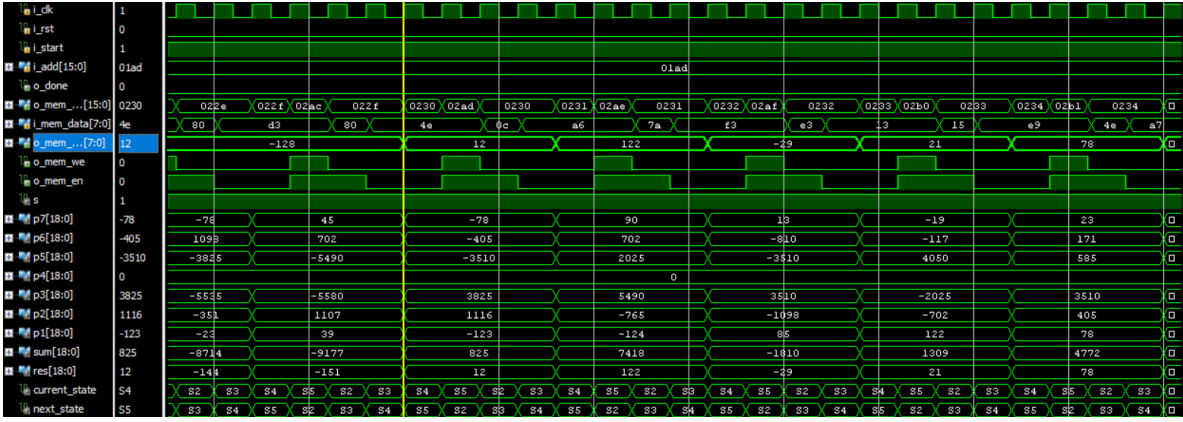


Figura 9: Somma di prodotti molto grandi

3.2.4 Sequenza massima di valori nel campo K ($k = 32759$)

Questo test verifica che il modulo hardware funziona correttamente anche nel caso di sequenza K massima possibile, che è pari a:

$$k_{max} = \left\lfloor \frac{65536 - 17}{2} \right\rfloor = 32759$$

Questo è possibile dal momento che il contatore è dimensionato in modo tale da contenere completamente un indirizzo di memoria. Si osserva che il contatore non è sovradimensionato in quanto, in queste condizioni, il valore massimo di `count` è pari a:

$$count_{max} = 17 + k_{max} + 3 + 1 = 32780 > 32767 = 2^{15} - 1$$

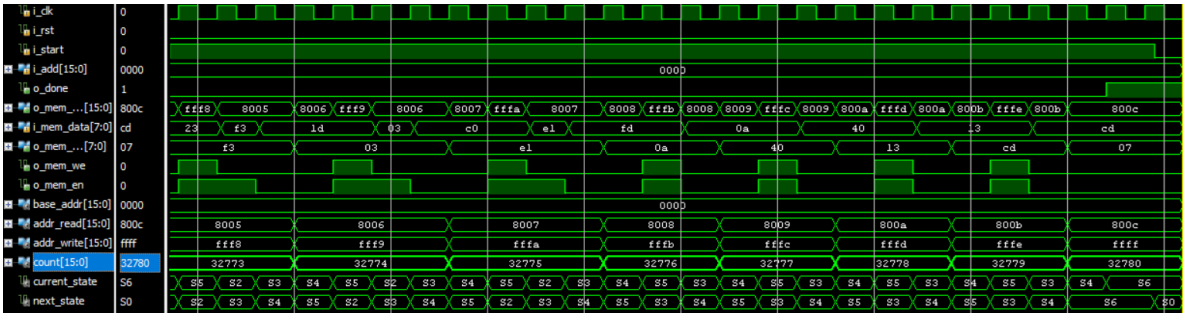


Figura 10: Parte finale della simulazione con sequenza K massima

3.2.5 Comportamento all'ultimo indirizzo disponibile in memoria

Questo test verifica che il modulo legge e scrive solo sull'area di memoria consentita. Questo caso è gestito nello stato S4, in cui si determina l'operazione da svolgere dopo aver caricato il nuovo dato nel registro opportuno.

Per come il modulo è implementato, se l'ultimo dato è scritto in `0xFFFF`, l'incremento del contatore causa un overflow sul segnale `add_write`, che assume valore nullo. Tuttavia, questo non rappresenta un problema nel funzionamento del circuito perché, a quel punto, il modulo si trova in stato di DONE.

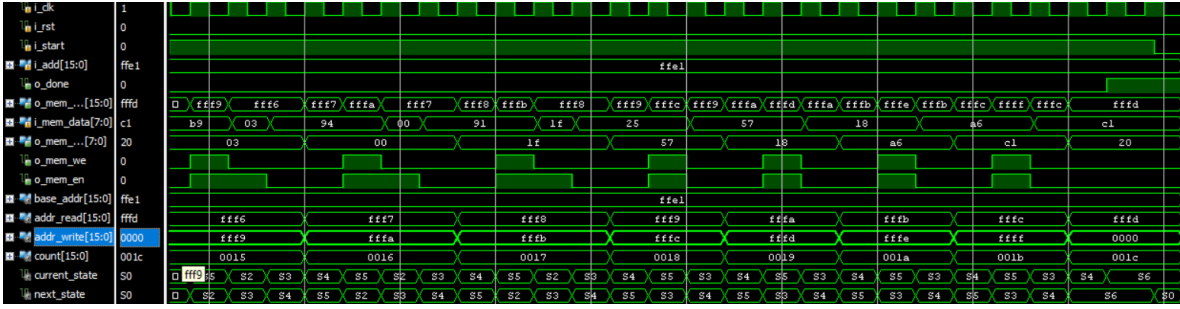


Figura 11: Parte finale della simulazione che termina sull'ultimo indirizzo di memoria

3.2.6 Reset asincrono

Questo test verifica che, quando il segnale di RESET si alza ($i_rst = 1$):

- tutti i registri e i segnali intermedi caricano il loro valore di default;
- il modulo si pone nello stato di reset S0 e, quindi, in attesa di una richiesta di elaborazione.

Nella Figura 9, è riportato solo il comportamento di alcuni registri. Gli altri seguono la stessa logica. Inoltre, si osserva che, una volta posto $i_start = 1$, l'elaborazione è eseguita normalmente.

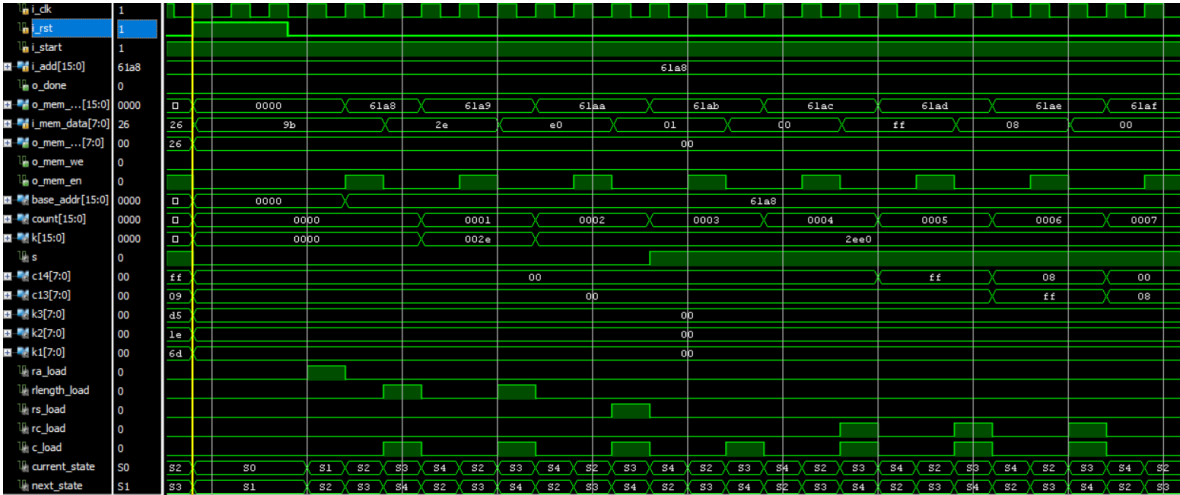


Figura 12: Simulazione con reset asincrono

3.2.7 Elaborazione multipla

Questo test verifica che il modulo riceve correttamente un'altra richiesta di elaborazione senza fare prima il reset. Questo caso è gestito tramite il segnale `init_count`, che inizializza il contenuto del contatore a 0x0000 prima di leggere il primo dato dalla memoria.

Gli altri registri non sono inizializzati perché saranno sovrascritti prima del primo calcolo valido, compreso il contenuto dello shift register K. In particolare, per come il modulo è implementato, gli ultimi tre registri dello shift register K memorizzano i

valori letti dallo stream precedente, mentre i rimanenti memorizzano il valore 0x00. Quando una nuova elaborazione è iniziata, prima del primo calcolo valido si caricano nello shift register quattro valori, i quali quindi tolgono dalla memorizzazione il *junk* dell'elaborazione precedente. Inoltre, i segnali intermedi sono posti al loro valore di default una volta resettato il contatore.

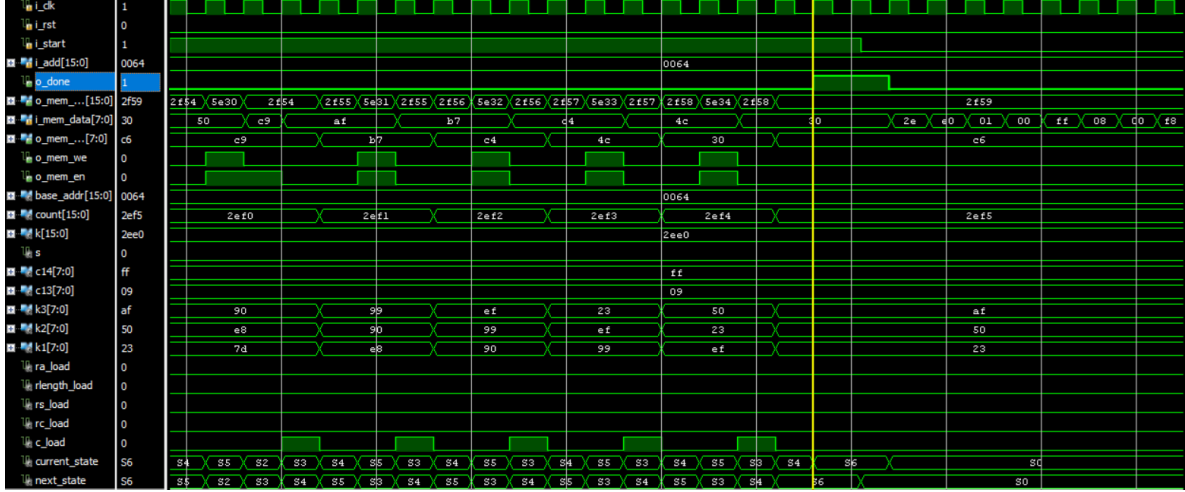


Figura 13: Comportamento di alcuni segnali nell'intorno di $o_done = 1$

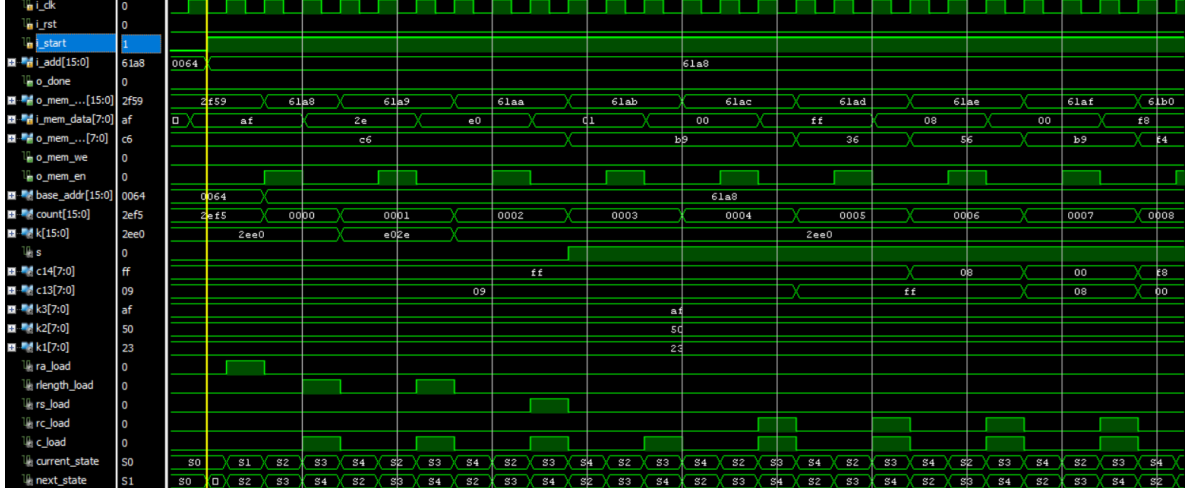


Figura 14: Comportamento dei segnali di Figura 11 all'inizio di un'altra elaborazione

