Prova Finale (Progetto di Reti Logiche)

Politecnico di Milano
Anno Accademico 2021/2022

Prof. Fabio Salice

Filippo Fini
Codice persona XXXX, Matricola XXXX

Francesca Grimaldi
Codice persona XXXX, Matricola XXXX

Indice

1 Introduzione

Il Progetto finale di Reti Logiche (per l'A.A. 2021/2022) ha come scopo quello di descrivere in linguaggio VHDL e successivamente sintetizzare tramite VIVADO, un modulo hardware che applichi un codice convoluzionale con tasso di trasmissione $\frac{1}{2}$ ai dati letti da memoria, ed in seguito ne scriva il risultato nella memoria stessa. Il componente deve dunque essere in grado di interfacciarsi correttamente con la memoria per la lettura e la scrittura dei dati, e di manipolare in modo appropriato l'input per ottenere un risultato che rispecchi la codifica convoluzionale.

1.1 Descrizione generale

Il modulo riceve in ingresso una sequenza di parole da 8 bit ciascuna. Ognuna di queste parole viene serializzata, ottenendo i singoli bit ai quali è poi applicato il codice convoluzionale $\frac{1}{2}$. Come rappresentato nella Figura 1, ad ogni bit in ingresso (U_k) ne corrispondono 2 in uscita (p_{1k},p_{2k}).

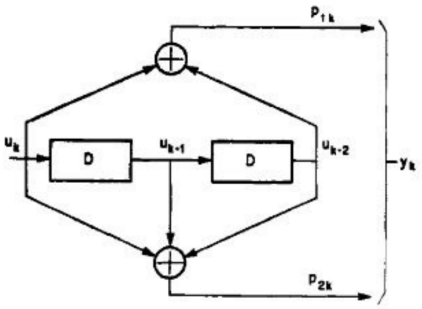


Figura 1: Codificatore convoluzionale con tasso di trasmissione $\frac{1}{2}$

Tutti i bit di uscita vengono concatenati per generare un flusso continuo da 1 bit, e l'output finale sarà dato dalla parallelizzazione, su 8 bit, di questo flusso continuo.

1.2 Serializzazione

Il primo passo per l'applicazione dell'algoritmo è quello di serializzare l'input. Dato un flusso di W parole da 8 bit ciascuna, si vuole isolare volta per volta un bit. Questo passo verrà ripetuto $8 * W$ volte, e il bit dell'iterazione corrente andrà a costituire l'input per il convolutore.

1.3 Codificatore convoluzionale

Il convolutore è una macchina sequenziale sincrona, dotata di clock globale e segnale di reset, che ad un bit in ingresso (U_k) ne fa corrispondere 2 in uscita

(p_{1k}, p_{2k}). In Figura 2 è possibile vederne la rappresentazione come macchina di Mealy (notazione U_k/p_{1k},p_{2k}).

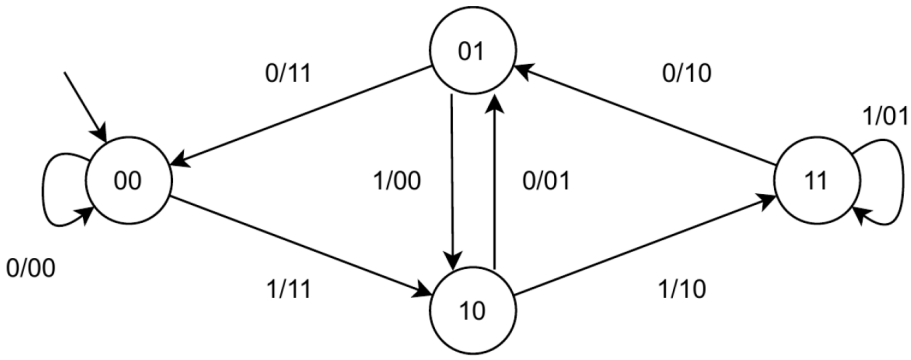


Figura 2: Convolutore

Il suo stato iniziale, che è anche quello al quale si torna dopo aver ricevuto un segnale di `RESET`, è lo stato `00`. Concatenando i 2 bit di uscita, si crea un flusso continuo Y da 1 bit, che sarà lungo $8 * W * 2$. Va sottolineato che il convolutore non viene riportato allo stato `00` all'inizio di ogni parola, ma solo quando riceve un segnale di `RESET`.

1.3.1 Esempio di codifica

Applicando, ad esempio, il codice convoluzionale $\frac{1}{2}$ al numero binario `01011101`, si avrebbe la seguente situazione:

k	0	1	2	3	4	5	6	7
U_k	0	1	0	1	1	1	0	1
p_{1k}	0	1	0	0	1	0	1	0
p_{2k}	0	1	1	0	0	1	0	0
next state	00	10	01	10	11	11	01	10

Tabella 1: Applicazione del codice convoluzionale al numero `01011101`

I 2 byte in uscita saranno: `00110100` e `10011000`.

1.4 Parallelizzazione

Scopo della parallelizzazione (o deserializzazione) dei bit è quello di tornare ad avere un flusso di parole da 8 bit ciascuna da poter scrivere in memoria. Ogni byte di Y rappresenta quindi una parola che farà parte del risultato finale, costituito complessivamente da $2 * W$ parole.

1.5 Dati e rappresentazione in memoria

Ogni parola (da 8 bit) può essere letta dalla memoria tramite indirizzamento al byte nel quale essa è memorizzata. Il byte all'indirizzo `0` rappresenta la lunghezza della sequenza da codificare, le cui parole si trovano tra l'indirizzo `1` e `999`. Dall'indirizzo `1000` in poi sono invece contenuti i byte della sequenza di uscita.

	Lunghezza input (W)	Indirizzo 0
Sequenza in input	Byte 1 da codificare	Indirizzo 1
	Byte 2 da codificare	Indirizzo 2
	...	
	Byte W da codificare	Indirizzo W
	...	
Sequenza in uscita	Byte 1 in uscita	Indirizzo 1000
	Byte 2 in uscita	Indirizzo 1001
	...	
	Byte 2W-1 in uscita	Indirizzo 1000+2W-2
	Byte 2W in uscita	Indirizzo 1000+2W-1
	...	

Poiché la lunghezza della sequenza di ingresso è rappresentata in binario su 8 bit, e poiché per codificare un numero decimale `n` in binario occorrono $log_2 n$ bit, il massimo numero di byte da codificare è dato da $2^8 - 1$, quindi `255`. Come conseguenza, gli indirizzi di memoria da `256` a `999` rimarranno sempre inutilizzati.

1.6 Funzionamento

Per prima cosa, il componente legge il byte all'indirizzo di memoria `0`, che rappresenta la lunghezza della sequenza di ingresso (W). Quindi passa all'indirizzo `1`, nel quale è contenuta la prima parola da codificare. La parola viene serializzata e ogni bit passa attraverso il convolutore. I 2 bit in uscita dal convolutore vengono concatenati ad ogni iterazione, ed in seguito alla loro parallelizzazione,

si scrivono i 2 byte in uscita agli indirizzi `1000` e `1001`. Dunque si passa a leggere la seconda parola da codificare, finché non si raggiunge la W-esima parola, i cui byte di uscita corrispondenti saranno scritti agli indirizzi `1000+2W-2` e `1000+2W-1`.

1.6.1 Esempio di funzionamento

Un esempio del funzionamento completo del modulo, nel caso in cui il flusso in input sia costituito da 2 sole parole, è descritto nelle seguenti tabelle.

Indirizzo	Contenuto
0	00000010
1	01011101
2	11000110
...	...
1000	...
1001	...
1002	...
1003	...
...	...

Tabella 2: Stato iniziale memoria

Indirizzo	Contenuto
0	00000010
1	01011101
2	11000110
...	...
1000	00110100
1001	10011000
1002	10011011
1003	00111010
...	...

Tabella 3: Stato finale memoria

In particolare è stato messo in evidenza lo stato della memoria prima e dopo l'elaborazione. Per quanto riguarda la codifica, si faccia sempre riferimento alla Tabella 1.

2 Architettura

2.1 Interfaccia del componente

L'interfaccia del componente 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_data : in std_logic_vector(7 downto 0);
        o_address : out std_logic_vector(15 downto 0);
        o_done : out std_logic;
        o_en : out std_logic;
        o_we : out std_logic;
        o_data : out std_logic_vector (7 downto 0)
    );
end project_reti_logiche;
```

Più nello specifico:

`i_clk` è il segnale di `CLOCK`;

`i_rst` è il segnale di `RESET` (inizializza la macchina a dei valori di default);

`i_start` è il segnale di `START`;

`i_data` è il segnale che arriva in seguito ad una richiesta di lettura dalla memoria;

`o_address` è il segnale che trasmette l'indirizzo alla memoria;

`o_done` è il segnale di uscita che determina la fine dell'eleborazione;

`o_en` è il segnale di `ENABLE` da mandare alla memoria per poter leggere o scrivere;

`o_we` è il segnale di `WRITE ENABLE` da mandare alla memoria con valore `1` se si vuole scrivere e `0` se si vuole leggere da essa;

`o_data` è il segnale di uscita verso la memoria.

2.1.1 Inizio e fine computazione

Il modulo inizia l'elaborazione nel momento in cui assume valore `1` il segnale `i_start`, che rimane alto. Dopo aver scritto l'intero risultato in memoria, anche il segnale `o_done` viene portato ad `1`, per comunicare il termine della computazione. Questo segnale rimane alto finché `i_start` non torna a `0`. Dal momento in cui anche `o_done` viene riportato al valore logico basso, può avere inizio un'altra computazione, seguendo le stesse modalità. Un segnale di `RESET` precede sempre la prima computazione, e se `i_rst` dovesse valere `1` in qualche momento durante l'elaborazione, i segnali dovranno essere portati ai loro valori di default.

2.2 Macchina a stati

Si è deciso di implementare il componente tramite una macchina a stati finiti, la cui rappresentazione è la seguente:

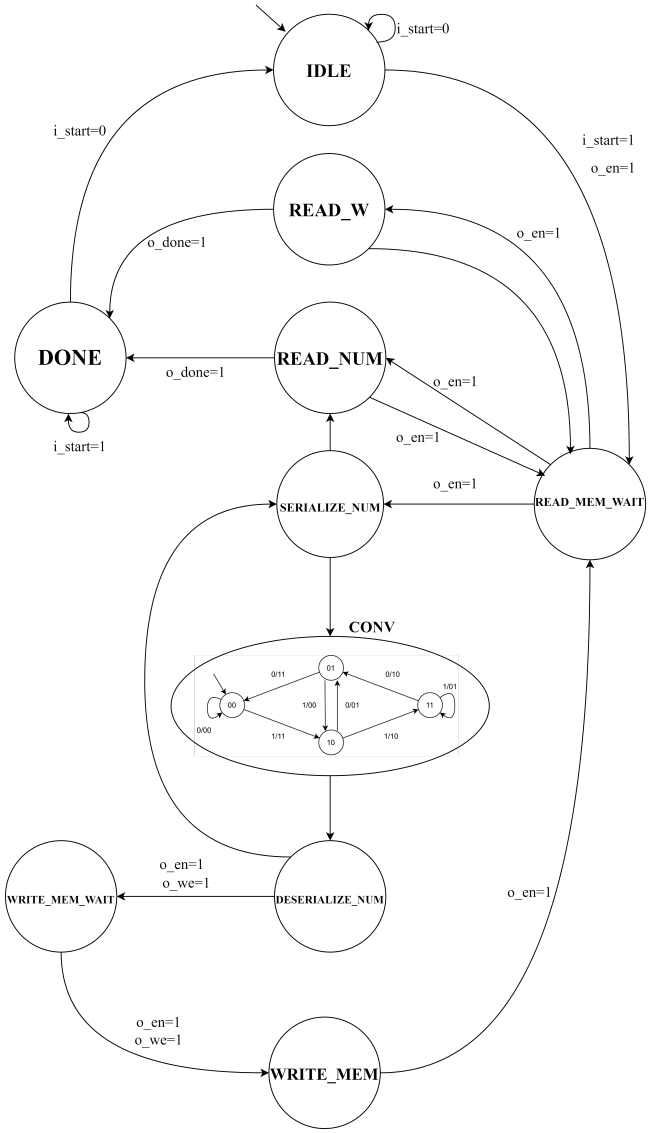


Figura 3: Macchina a stati finiti

Non è esplicitato il fatto che ad ogni ciclo di clock i segnali `o_done`, `o_en` e `o_we` vengono messi a `0`. Inoltre da ogni stato si torna ad `IDLE` se `i_rst` vale `1`.

Di seguito è fornita una breve descrizione per ognuno dei 10 stati di cui si compone la FSM.

`IDLE` : stato iniziale, nel quale viene riportata la macchina in caso di `RESET` o di terminazione della sequenza

`READ_MEM_WAIT` : stato di attesa necessario affinché avvenga il corretto aggiornamento dei dati provenienti dalla memoria

`READ_W` : stato nel quale viene letto il numero di parole che compongono la sequenza da codificare

`READ_NUM` : stato nel quale viene letta la parola da codificare

`SERIALIZE_NUM` : stato nel quale vengono serializzati i bit della parola letta nello stato `READ_NUM`

`CONV` : contiene un'altra macchina a stati finiti che rappresenta il convolutore. I suoi stati sono: `S_00`, `S_01`, `S_10`, `S_11`, come mostrato nella Figura 3

`DESERIALIZE_NUM` : stato nel quale vengono deserializzati i bit provenienti dal convolutore

`WRITE_MEM_WAIT` : stato di attesa necessario affinché si possano scrivere correttamente i dati in memoria

`WRITE_MEM` : stato nel quale avviene l'effettiva scrittura dei dati, e l'aggiornamento dell'indirizzo della memoria per l'operazione successiva

`DONE` : stato per la terminazione della computazione di una sequenza. Si rimane in questo stato fin quando `i_start` non torna a `0`, dopodiché si torna allo stato iniziale

2.2.1 Scelte implementative

L'intero componente è stato realizzato utilizzando due soli processi: `state_reg` e `cod_conv`.

`state_reg` : processo che gestisce gli aggiornamenti dei segnali ad ogni ciclo di clock e la routine di `RESET`

`cod_conv` : processo in cui è descritto il funzionamento del componente e dei vari stati della FSM

Come accennato precedentemente, lo stato `CONV`, che rappresenta il convolutore, è a sua volta una macchina a stati. Dal codice VHDL abbiamo infatti:

```
type state_type is (IDLE, READ_W, READ_NUM, READ_MEM_WAIT,
    SERIALIZE_NUM, CONV, DESERIALIZE_NUM, WRITE_MEM_WAIT,
    WRITE_MEM, DONE);
type state_type_conv is (S_00, S_01, S_10, S_11);
```

Si è optato per questa soluzione per mantenere il concetto di codificatore convoluzionale come macchina a stati a sé, rendendo la notazione più compatta ed evitando l'introduzione di altri 4 (o più) stati nella FSM principale.

2.3 Segnali interni

`current_state` : segnale che rappresenta lo stato corrente della macchina a stati finiti (Figura 3). Il suo valore di default è `IDLE`

`conv_current_state` : segnale che rappresenta lo stato corrente del convolutore, all'interno dello stato `CONV` della FSM principale. Il suo valore di default è `S_00`

`return_state` : segnale che tiene traccia dello stato successivo di `READ_MEM_WAIT`. Il suo valore di default è `IDLE`

`o_address_copy` : segnale che tiene memoria del valore di `o_address`. Il suo valore di default è `(others => '0')`

`words_number` : segnale che rappresenta il numero di parole della sequenza. Il suo valore di default è `0`

`count_read` : segnale che tiene conto del numero di parole già lette della sequenza. Il suo valore di default è `0`

`count_write` : segnale che tiene conto del numero di parole scritte in memoria. Il suo valore di default è `0`

`i` : contatore utile per la serializzazione della parola letta. Il suo valore iniziale è `7`, e viene decrementato di 1 ogni volta che avviene la serializzazione di un bit. Una volta completata la serializzazione di una parola, viene riportato a `7`

`signal_out` : segnale che contiene l'i-esimo bit della parola letta, serializzato. Il suo valore di default è `0`

`j` : contatore utile per la deserializzazione. Il suo valore iniziale è `7`, e viene decrementato di 2 ogni volta che vengono deserializzati due bit. Viene riportato a `7` dopo la scrittura di una parola in memoria

`new_out_1` : segnale che contiene il primo bit in uscita dal convolutore. Il suo valore di default è `0`

`new_out_2` : segnale che contiene il secondo bit in uscita dal convolutore. Il suo valore di default è `0`

`new_number` : segnale che rappresenta la parola da scrivere in memoria. Il suo valore di default è `(others => '0')`

Sono stati aggiunti inoltre dei segnali, denotati con suffisso `_next`, in abbinamento alla maggior parte dei segnali appena descritti. L'obiettivo è quello di evitare la sintesi di latch. I segnali di tipo `_next` aggiornano i relativi segnali ad ogni ciclo di clock.

3 Risultati sperimentali - Sintesi

In questa sezione sono riportate le informazioni più rilevanti raccolte durante la fase di sintesi.

3.1 Utilization

Nella stesura del codice VHDL si è cercato di ottimizzare in modo da mantenere al minimo l'area occupata dal componente, e al contempo evitarne comportamenti indesiderati.
Il componente risulta essere sintetizzabile e implementabile in modo corretto, con 103 LUT e 73 Flip Flop. Quanto si evince dal *Report Utilization* inserito qui di seguito, è innanzitutto il fatto che le percentuali di LUT e Flip Flop inferiti durante la sintesi sono entrambe inferiori all'1%. Dunque, come auspicato, solo una minima parte di tutta disponibilità dell'FPGA è effettivamente utilizzata.
In aggiunta, durante la sintesi non sono inferiti latch, situazione che impedisce propagazioni non volute di segnali.

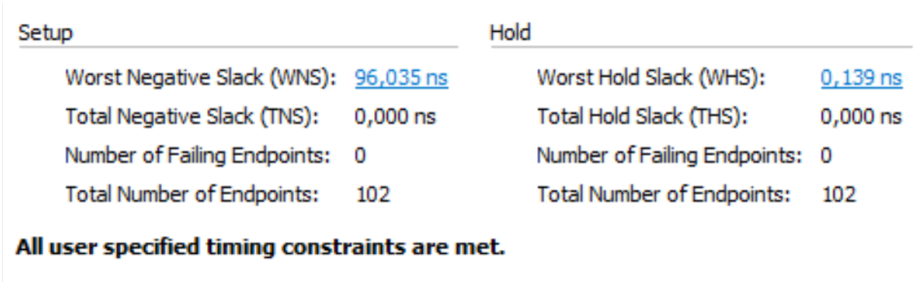
Site Type	Used	Fixed	Available	Util%
Slice LUTs*	103	0	134600	0.08
LUT as Logic	103	0	134600	0.08
LUT as Memory	0	0	46200	0.00
Slice Registers	73	0	269200	0.03
Register as Flip Flop	73	0	269200	0.03
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

3.2 Timing

Dalle specifiche, il componente deve essere in grado di operare con un periodo di clock $T_{clk} \leq 100ns$. Per verificare questo, è stato aggiunto un timing constraint con $T_{clk} = 100ns$ e duty cycle pari al 50%. Analizzando il *Report Timing Summary* in Figura 4, è chiaro che questa condizione è rispettata.
A livello teorico,

$$T_{clk,min} = T_{clk} - WNS$$

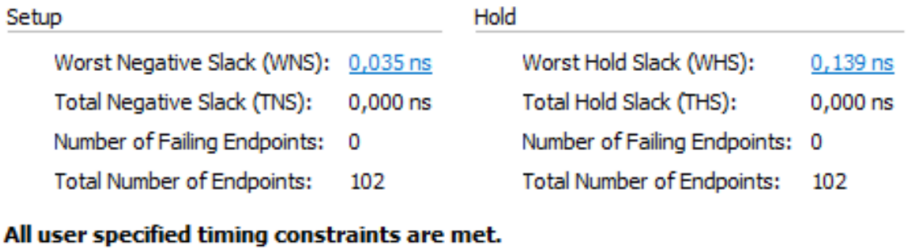
dove WNS è il Worst Negative Slack.
Imponendo vincoli sempre più stringenti sul periodo del clock, è stato constatato che $T_{clk} = 4ns$ è il minimo periodo per il quale il componente continua a funzionare correttamente. È possibile avere conferma di questo dalla Figura 5, nella quale il WNS è ancora positivo.



Setup		Hold	
Worst Negative Slack (WNS):	96,035 ns	Worst Hold Slack (WHS):	0,139 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	102	Total Number of Endpoints:	102

All user specified timing constraints are met.

Figura 4: *Report Timing Summary* ($T_{clk} = 100ns$)



Setup		Hold	
Worst Negative Slack (WNS):	0,035 ns	Worst Hold Slack (WHS):	0,139 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	102	Total Number of Endpoints:	102

All user specified timing constraints are met.

Figura 5: *Report Timing Summary* $T_{clk} = 4ns$

4 Risultati sperimentali - Simulazioni

Dopo la stesura del codice, questo è stato sottoposto ai testbench forniti insieme alle specifiche, al fine di accertare che il modulo operasse correttamente.
In particolare il focus è stato posto su:

1. casi limite;
2. funzionamento del `RESET`;
3. codifica continua;
4. stress test.

Nelle successive immagini si nota che il valore del segnale di uscita dalla RAM, prima di essere utilizzato, e quindi prima di essere aggiornato al valore corretto, subisce delle variazioni. Queste però non influiscono sul giusto funzionamento del componente.

4.1 Test casi limite

4.1.1 Minimo

Viene testato il caso limite in cui il numero di parole da codificare è uguale a 0.
Risultati delle simulazioni:

- *Behavioral*: 850 000 ps
- *Post Synthesis*: 850 100 ps

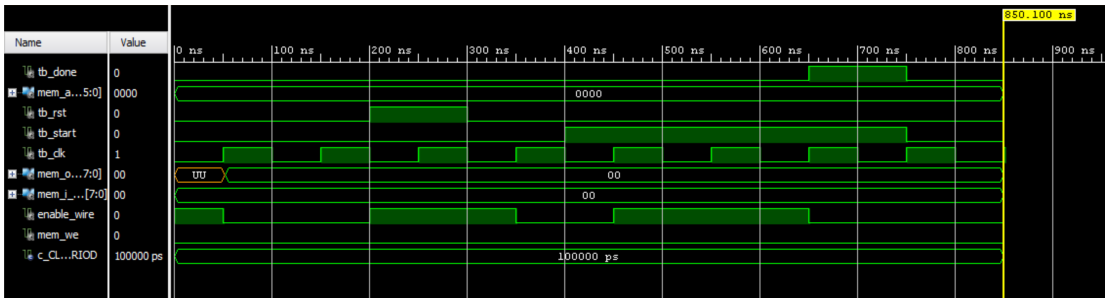


Figura 6: Simulazione *Post Synthesis* - minima sequenza in ingresso

4.1.2 Massimo

È testato il caso limite in cui il numero di parole da codificare è massimo, e quindi pari a 255.
Risultati delle simulazioni:

- *Behavioral*: 842 550 000 ps
- *Post Synthesis*: 842 550 100 ps

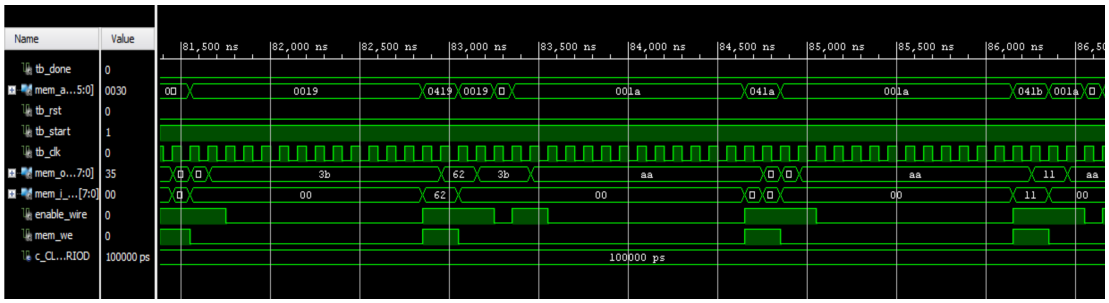


Figura 7: Simulazione *Post Synthesis* - massima sequenza in ingresso

4.2 Test reset

In questo test viene effettuato un `RESET` asincrono.
Risultati delle simulazioni:

- *Behavioral*: 22 750 000 ps
- *Post Synthesis*: 22 750 100 ps

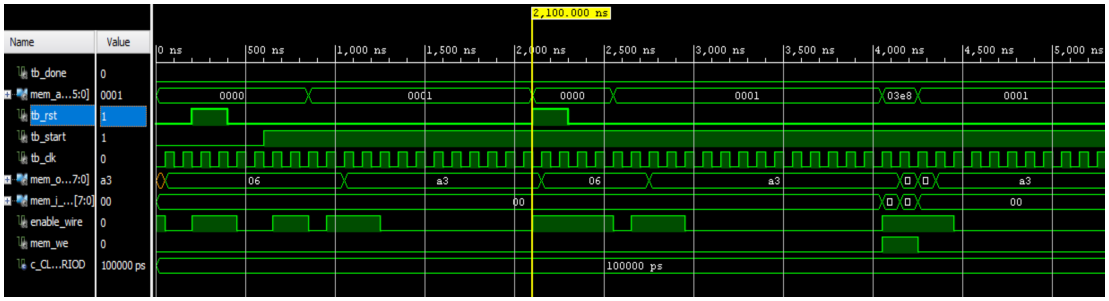


Figura 8: Simulazione *Post Synthesis* - funzionamento del `RESET`

4.3 Test codifica continua

Tre sequenze sono codificate di seguito, senza ricezione del segnale di `RESET` tra l'una e l'altra.
Risultati delle simulazioni:

- *Behavioral*: 52 350 000 ps
- *Post Synthesis*: 52 350 100 ps

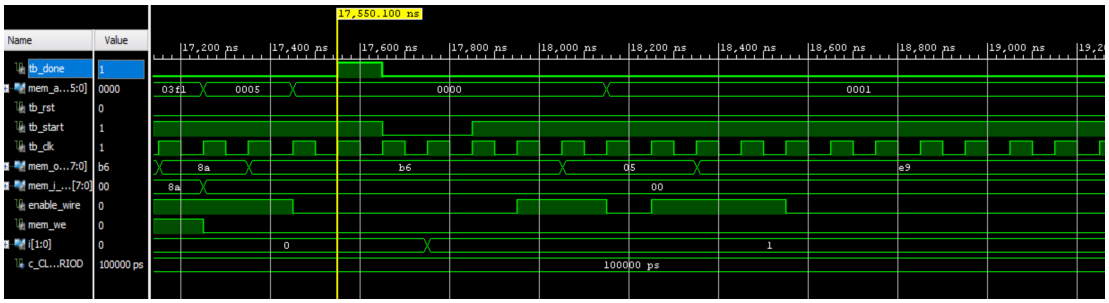


Figura 9: Simulazione *Post Synthesis* - codifica continua

4.4 Stress test

Il componente è stato sottoposto anche ad uno stress test creato utilizzando un generatore di test casuali scritto in Python, che prevedeva circa 4000 sequenze composte da un numero di parole variabile, compreso tra 0 e 255.

5 Conclusioni

Una prima versione del codice prevedeva che il segnale `o_en`, che permette la comunicazione con la memoria, rimanesse sempre ad `1`, e che da ogni stato si passasse sempre attraverso uno stato di wait prima di raggiungere il successivo. Successive fasi di ottimizzazione hanno fatto in modo che `o_en` venisse portato ad `1` solo quando necessario, e che si evitassero inutili passaggi attraverso stati di wait, riducendo notevolmente anche i tempi di elaborazione. Si è cercato di ridurre al minimo l'utilizzo di LUT e Flip Flop, nonché di stati della FSM e segnali interni. Tutti i test eseguiti hanno avuto esito positivo, sia in *Behavioral* che in *Post Synthesis*.
Dunque il componente risulta essere in grado di applicare il codice convoluzionale $\frac{1}{2}$ a qualsiasi sequenza in ingresso, operando in maniera conforme alle specifiche.