Progetto di Reti Logiche 2022

Andrea Sanguineti
(Codice Persona 10739788 - Matricola 936930)

Maggio 2022

Indice

1 Introduzione

1.1 Obiettivo

Lo scopo del progetto è quello di implementare un modulo hardware in grado di interfacciarsi con una memoria da cui leggere uno stream di dati e su cui salvarne in seguito la corrispettiva codifica convoluzionale 1/2.

1.2 Specifiche generali

Il modulo si interfaccia con una memoria (istanziata nei Test Bench) da cui preleva uno stream da codificare e su cui scriverà infine il risultato dell'elaborazione. I dati sulla memoria sono organizzati in questo modo:

- All'indirizzo 0 è memorizzato la lunghezza $\boldsymbol{N}$ (in Byte, $N \leq 255$) della sequenza da codificare.
- Dall'indirizzo 1 all'indirizzo N sono memorizzate le parole da codificare.
- Dall'indirizzo 1000 all'indirizzo $(2 \times N - 1) + 1000$ è memorizzato lo stream risultante dall'elaborazione *(di lunghezza* ***doppia*** *rispetto a quello elaborato).*

Indirizzo	Valore	
0	00000010 (2)	N = lunghezza in byte della sequenza da codificare
1	01110000 (112)	1° byte della sequenza da codificare
2	10100100 (164)	2° (ultimo) byte della sequenza da codificare
[...]	[...]	
1000	00111001 (57)	1° byte della sequenza elaborata
1001	10110000 (176)	
1002	11010001 (209)	
1003	11110111 (247)	(2*N)° byte della sequenza elaborata

Figura 1: Esempio di schema della memoria con $N = 2$ parole da elaborare

1.3 Protocollo comunicazione

La comunicazione tra il modulo e la memoria avviene secondo un protocollo così costruito:

- Prima della 1^a codifica viene **sempre** dato un segnale di ***i_rst = 1***.
- Per cominciare l'elaborazione devo portare a **1** il segnale ***i_start***.
- Una volta finita l'elaborazione viene portato a **1** il segnale ***o_done***.
- Soltanto dopo aver riportato a **0** il segnale ***i_start*** ritorno allo stato iniziale e posso cominciare una nuova elaborazione.

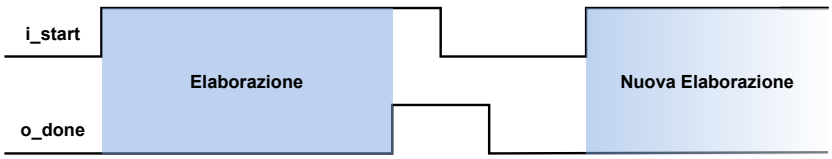

Figura 2: Protocollo di comunicazione

1.4 Codifica

Essendo un codice convoluzionale 1/2 questa codifica produrrà uno stream in uscita la cui lunghezza è il doppio di quella in entrata, generato secondo il diagramma a stati descritto nella **Figura 3**.

Ogni volta che viene ricevuto un segnale ***i_rst = 1*** o quando finisce l'elaborazione ritorna allo stato ***C0***, altrimenti procedo costruendo lo stream concatenando le coppie di bit in uscita, transizione dopo transizione.

Esempio:
La prima parola da codificare (all'indirizzo **1** della memoria) è W = **01110000**, leggendo i bit da sinistra verso destra la sua codifica avverrà passando per i seguenti stati: ***C0*** $\rightarrow$ ***C0*** $\rightarrow$ ***C2*** $\rightarrow$ ***C3*** $\rightarrow$ ***C3*** $\rightarrow$ ***C1*** $\rightarrow$ ***C0*** $\rightarrow$ ***C0*** $\rightarrow$ ***C0***
Il risultato dell'elaborazione sarà costituito dalle 2 parole Z_1 = **00111001** e Z_2 = **10110000** salvate poi negli indirizzi **1000** e **1001** della memoria.

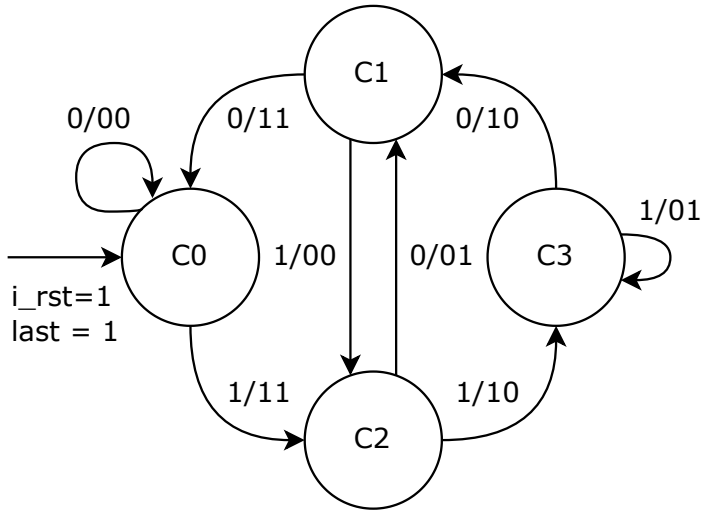

Figura 3: Diagramma a stati della codifica

1.5 Interfaccia del modulo

L'interfaccia del modulo è così definita:

- ***i_clk***: segnale di *CLOCK* generato dal Test Bench.
- ***i_rst***: segnale di *RESET*, se **1** riporta in modo asincrono il componente allo stato iniziale.
- ***i_start***: segnale di *START*, va portato a **1** per cominciare l'elaborazione e a **0** in seguito della ricezione del segnale ***done = 1***.
- ***i_data***: segnale da **8 bit** in uscita dalla memoria verso il modulo.
- ***o_address***: indirizzo di memoria da **16 bit** in uscita verso la memoria.
- ***o_done***: segnale che comunica la fine dell'elaborazione.
- ***o_en***: segnale di *ENABLE*, se **1** è possibile comunicare con la memoria.
- ***o_we***: segnale di *WRITE-ENABLE*, deve essere **1** per poter scrivere sulla memoria e **0** per leggere.
- ***o_data***: segnale da **8 bit** in uscita dal modulo che la memoria andrà a scrivere se ***o_we = 1***.

2 Architettura

2.1 Segnali interni

- ***reg0***: Registro a **8 bit** dove salvo la lunghezza **N** *(in Byte)* della sequenza totale da elaborare ($0 \leq \mathbf{N} \leq 255$)
- ***reg1***: Registro a **8 bit** dove salvo la lunghezza **M** *(in Byte)* della sequenza già elaborata ($0 \leq \mathbf{M} \leq \mathbf{N} \leq 255$)
- ***reg2***: Registro a **8 bit** dove salvo la parola da elaborare.
- ***r0_load***: Se ***1*** posso caricare dati in ***reg0***.
- ***r1_load***: Se ***1*** posso caricare dati in ***reg1***.
- ***r2_load***: Se ***1*** posso caricare dati in ***reg2***.
- ***r1_sel***: Seleziona qual è il nuovo dato da caricare in ***reg1***:
  (Se ***0*** $\rightarrow$ *"00000000"*, se ***1*** $\rightarrow M + 1$)
- ***r2_sel***: Seleziona qual è il nuovo dato da caricare in ***reg2***:
  (Se ***0*** $\rightarrow i\_data$, se ***1*** $\rightarrow reg2 << 4$)
- ***addr_sel***: Seleziona l'indirizzo da mandare in output verso la memoria.
- ***compute***: Se ***1*** scatena l'elaborazione del dato presente nei **4 bit** più significativi di ***reg2*** e manda in output il risultato su ***o_data***.
- ***last***: Se ***1*** non ci sono altre parole da elaborare (***reg0*** = ***reg1***)
- ***done***: Se ***1*** segnala la fine dell'elaborazione
- ***cur_state***: Segnale che indica lo **stato corrente** *(S0 - S7)* della macchina a stati.
- ***next_state***: Segnale che indica il **prossimo stato** *(S0 - S7)* verso cui deve fare la transizione la macchina a stati.
- ***cur_compute_state***: Segnale su cui salvo l'ultimo stato *(C0 - C3)* in cui è rimasta la macchina a stati relativa all'**elaborazione** della codifica.

2.2 Descrizione processi

Il componente è interamente descritto in un singolo modulo con **10 processi**. Per quelli relativi ai registri ***reg0*** e ***reg2*** ho deciso di sfruttare il fronte di **discesa** del clock in modo tale da non introdurre uno stato intermedio a causa del **delay** in fase di lettura dell'output della memoria *(**2 ns** in ritardo rispetto al fronte di salita)*.
Il registro ***reg1*** invece non legge valori da memoria quindi non è un problema aggiornarlo sul **fronte di salita**.
In caso di *RESET* ***(i_rst = 1)*** tutti i segnali vengono ripristinati ai loro valori di **default** in modo asincrono dai corrispettivi processi.

2.2.1 Reg0 Process

Se ***r0_load = 1***, aggiorna sul **fronte di discesa** del clock il dato in ***reg0*** col segnale in ingresso nel componente dalla porta ***i_data*** *(in questo caso la lunghezza della sequenza da elaborare)*.

2.2.2 Reg1 Process

Se ***r1_load = 1***, aggiorna sul **fronte di salita** del clock il dato in ***reg1*** in funzione del segnale ***r1_sel:***

se $0 \rightarrow$ **reg1** $= 0$, se $1 \rightarrow$ ***reg1*** $= reg1 + 1$.

All'inizio di una nuova elaborazione porta ***reg1*** a **0** e poi lo incrementa man mano contando il numero di byte della sequenza che ha finito di elaborare fino a raggiungere il valore presente in ***reg0*** ($M = N \Rightarrow$ ***last = 1***)

2.2.3 Reg2 Process

Se ***r2_load = 1***, aggiorna sul **fronte di discesa** del clock il dato in ***reg2*** in funzione del segnale ***r2_sel:***

se $0 \rightarrow$ ***reg2*** $= i\_data$, se $1 \rightarrow$ ***reg2*** $= reg2 << 4$.

Siccome la codifica della parola salvata in ***reg2*** produce un risultato a **16 bit** e la memoria è **indirizzata al Byte**, la codifica avverrà in **2 fasi**: elaboro prima i primi **4 bit** più significativi e salvo in memoria gli **8 bit** di risultato, poi faccio un ***left shift*** di **4 bit** su ***reg2*** per continuare a codificare con lo stesso processo (*Compute Process*) anche i **4 bit** meno significativi e salvo infine gli ultimi **8 bit** di risultato.

2.2.4 Cur_State Process

Sul **fronte di salita** del clock assegna a ***cur_state*** il valore presente in ***next_state*** passando effettivamente allo stato successivo.

2.2.5 Next_State Process

Processo che si occupa di precalcolare in funzione dei segnali ***curr_state***, ***i_start*** e ***last*** lo stato futuro del modulo come descritto in **Figura 4**.

2.2.6 O_Done Process

Sul **fronte di discesa** del clock assegna a ***o_done*** il valore presente in ***done***. Si è reso necessario per evitare di mandare in uscita eventuali **glitch** presenti *(in timing)* sul segnale ***done*** poiché il Test Bench va a leggere il segnale ***o_done*** in modo asincrono e in tal caso andrebbe a terminare la simulazione erroneamente.

2.2.7 O_Address Process

Calcola e assegna a ***o_address*** il giusto valore in funzione di ***addr_sel***:

- Se ***00***: $0 \rightarrow$ lunghezza **N** della sequenza da leggere
- Se ***01***: $(M+1) \in [1, 255] \rightarrow$ valore da elaborare
- Se ***10***: $(2 \times M + 1000) \in \{1000, 1002, ..., 1508\} \rightarrow 1^a$ parola da scrivere
- Se ***11***: $(2 \times M + 1001) \in \{1001, 1003, ..., 1509\} \rightarrow 2^a$ parola da scrivere

2.2.8 Signals Process

In funzione del segnale ***cur_state*** (e quindi dello stato corrente del modulo) va ad impostare il giusto valore per ogni segnale del componente.

2.2.9 Compute Process

Si occupa dell'**elaborazione** del dato letto dalla memoria, lavorando quindi su ***reg2***, se ***compute = 1***, va a leggere i suoi **4 bit** più significativi uno dopo l'altro tramite un ciclo for, simulando la macchina a stati descritta in **Figura 3**. Ad ogni bit letto va ad aggiungere nelle posizioni ***x*** e ***x - 1*** della variabile ***res*** (il risultato) i **2 bit** in output della corrispettiva transizione.
x è una variabile interna al processo pari a $\mathbf{i \times 2 - 7}$, dove ***i*** è l'***indice*** del ciclo for che va da 7 a 4 (gli indici dei **4 bit** più significativi di ***reg2***), quindi ***x*** assumerà i valori 7, 5, 3 e 1 e insieme a ***x - 1*** andrà a coprire tutti gli indici degli ***8 bit*** della variabile ***res***.
Una volta terminata la costruzione di ***res*** il suo valore viene assegnato al segnale ***o_data*** in uscita verso la memoria.
Siccome per l'elaborazione del dato successivo serve riprendere dall'ultimo stato in cui il modulo si era fermato, il processo si occupa anche di aggiornare il segnale ***curr_compute_state*** in funzione delle transizioni fatte durante il calcolo e di riportarlo a ***C0*** in seguito a un segnale di ***i_rst*** o ***last = 1***.
Questo processo verrà usato ***2 volte*** per ogni dato della sequenza letto, la seconda volta genererà altri **8 bit** in uscita lavorando sui **4** meno significativi presenti in ***reg2***: per farlo verrà eseguito un ***left shift*** di **4 bit** su ***reg2*** così da non dover cambiare il comportamento del processo.

Esempio: elaborazione di ***01110000*** partendo dallo stato C0 $\rightarrow Z_1, Z_2$

Calcolo Z_1					
i	**reg2**	**reg2(i)**	**curr_compute_state**	**x**	**res**
7	***011***10000	*0*	$C0 \rightarrow C0$	7	***00***000000
6	***011***10000	*1*	$C0 \rightarrow C2$	5	00***11***0000
5	***011***10000	*1*	$C2 \rightarrow C3$	3	0011***10***00
4	***011***10000	*1*	$C3 \rightarrow C3$	1	00111***01***
Calcolo Z_2					
i	**reg2**	**reg2(i)**	**curr_compute_state**	**x**	**res**
7	***0000***0000	*0*	$C3 \rightarrow C1$	7	***10***000000
6	***0000***0000	*0*	$C1 \rightarrow C0$	5	10***11***0000
5	***0000***0000	*0*	$C0 \rightarrow C0$	3	1011***00***00
4	***0000***0000	*0*	$C0 \rightarrow C0$	1	10110***00***

2.3 Descrizione Stati

Il modulo è implementato come un automa a stati finiti composto da **8 stati** che vanno da ***S0*** a ***S7*** descritti in seguito:

2.3.1 S0 State

É lo stato iniziale dove il modulo torna ogni volta che viene ricevuto un segnale di ***i_rst = 1*** e ci resta finché non viene letto ***i_start = 1***, cioè quando viene richiesta una **nuova** elaborazione.

2.3.2 S1 State

É appena stata richiesta una nuova elaborazione quindi per prima cosa si occupa di mandare un segnale di ***o_en = 1*** alla memoria per **abilitarla** e avere pronto su ***i_data*** il valore letto all'indirizzo **0**.
Imposta ***r1_load = 1*** per impostare ***reg1*** a **0** *(r1_sel = 0)* e ripristinare il contatore delle ***parole elaborate*** *(M = 0)*. Questi effetti avranno luogo a partire dal fronte di clock di salita successivo, quindi in *S2*.

2.3.3 S2 State

Imposta ***r0_load = 1*** in modo tale che ***reg0*** si aggiorni con il valore letto da ***i_data*** contenente la ***lunghezza della sequenza da elaborare (N)*** sul successivo fronte di discesa del clock (altrimenti l'aggiornamento del dato avverrebbe prima dell'effettivo arrivo del segnale della memoria che ha un ritardo di ***2 ns*** rispetto al fronte di salita)
Imposta ***addr_sel*** a ***01*** ($\Rightarrow$ ***o_address*** $= 00000001 = M+1$) per preparare la memoria a leggere il valore presente all'indirizzo **1** nello stato successivo.
In questo stato viene gestita anche la situazione in cui la ***lunghezza della sequenza (N)*** sia ***0*** poiché in questo caso ***reg0*** $=$ ***reg1*** $\Rightarrow$ ***last=1*** e quindi ***next_state*** sarà ***S7*** invece che ***S3*** terminando direttamente l'elaborazione.

2.3.4 S3 State

Imposta ***r2_sel = 1*** in modo tale che ***reg2*** si aggiorni sul successivo fronte di discesa del clock con il valore presente all'indirizzo ***M + 1*** della memoria, cioè il prossimo dato da **elaborare**, e imposta anche ***compute = 1*** cosicché *S4* cominci l'elaborazione dei sui primi **4 bit**.

2.3.5 S4 State

Comincia l'elaborazione dei **4 bit** più significativi presenti in ***reg2***, come descritto nel ***Compute Process***, al termine della quale verranno presentati su ***o_data*** i primi **8 bit** di risultato.
Imposta inoltre ***o_we = 1*** e ***addr_sel*** a ***10*** ($\Rightarrow$ ***o_address*** $= 2 \times M + 1000$) così da avere nel prossimo stato la memoria pronta all'indirizzo giusto su cui **scrivere** questo primo risultato.
Mantiene ***compute = 1*** per continuare l'elaborazione di ***reg2*** su *S5*, per far ciò però deve lavorare sui **4 bit** meno significativi quindi, per non cambiare il funzionamento del *Compute Process*, imposta ***r2_load = 1***, ***r2_sel = 1*** effettuando un *left shift* di **reg2** e portando i **4 bit** su cui dovrà lavorare nella stessa posizione in cui erano gli altri **4** già elaborati.

2.3.6 S5 State

Scrive in memoria all'indirizzo $2 \times M + 1000$ i primi ***8 bit*** di risultato e comincia l'elaborazione degli altri **4 bit** di **reg2** al medesimo modo di *S4*.
Mantiene **o_we = 1** per scrivere in memoria durante *S6* gli ultimi **8 bit** dell'elaborazione del dato inizialmente presente in **reg2** ma impostando questa volta ***addr_sel*** a ***11*** ($\Rightarrow$ ***o_address*** $= 2 \times M + 1001$).
Imposta inoltre ***r1_load = 1***, ***r1_sel = 1*** così da incrementare sul prossimo fronte di salita del clock il contatore *(M)* del numero delle ***parole elaborate***.

2.3.7 S6 State

Scrive in memoria all'indirizzo $2 \times M + 1001$ gli ultimi ***8 bit*** di risultato e verifica se il nuovo valore incrementato presente in ***reg1*** *(M)* è uguale a quello in ***reg0*** *(N, la lunghezza* ***totale*** *della sequenza)*. In tal caso vuol dire che l'elaborazione è arrivata al **termine** e imposta ***last = 1*** procedendo verso *S7*, altrimenti torna a *S3* impostando la memoria all'indirizzo M+1 tramite ***addr_sel = 01*** per leggere la nuova parola da elaborare.

2.3.8 S7 State

L'elaborazione è **terminata**, manda in output su ***o_done*** il segnale ***done***, **disattiva** la memoria impostando ***o_en = 0*** e resta in attesa che il segnale ***i_start*** diventi **0** per passare a *S0* e ricominciare da capo aspettando che venga richiesta una **nuova** elaborazione.

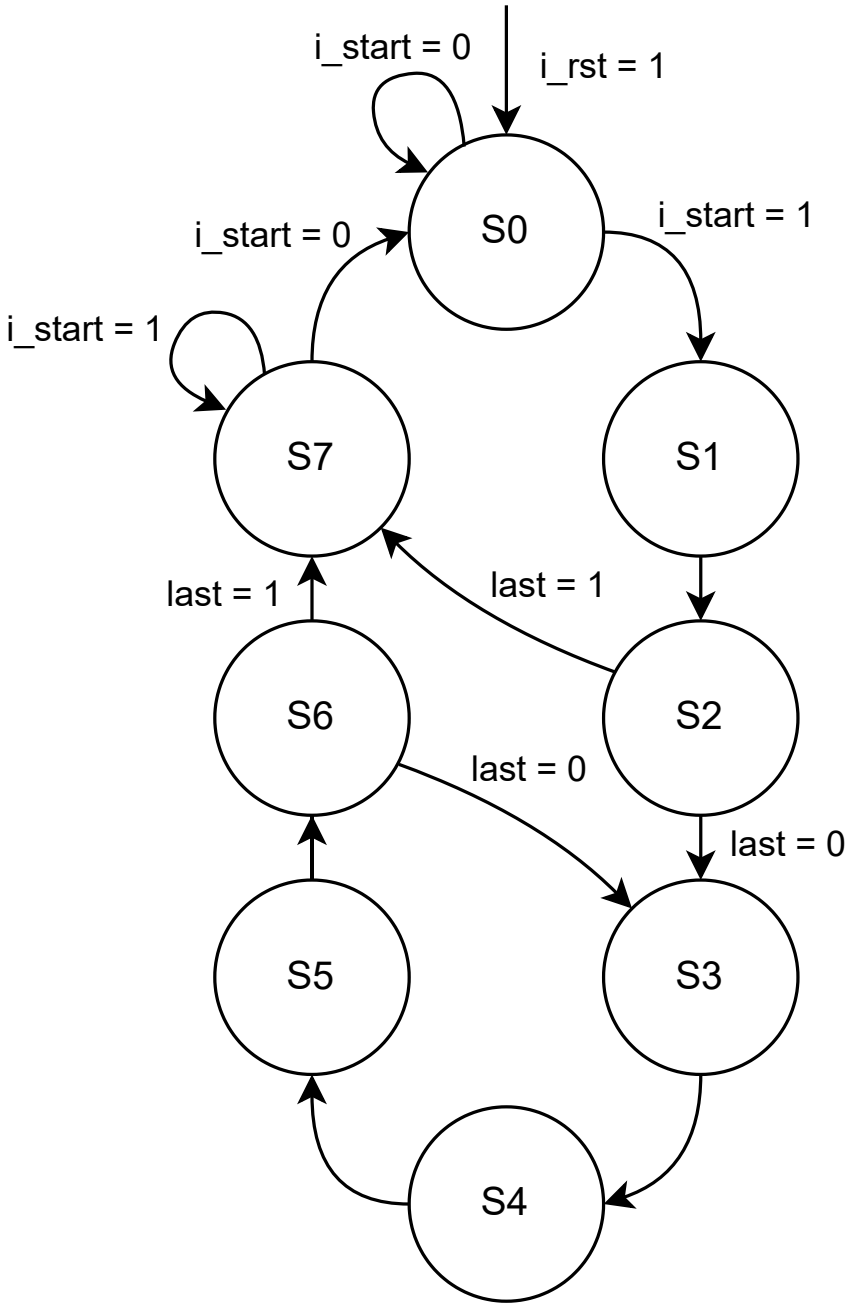

Figura 4: Macchina a stati

3 Risultati Sperimentali

3.1 Report Di Sintesi

Nella **Tabella 1** è mostrato il **Timing Report** da cui è possibile notare che il **WNS** *(Worst Negative Slack)*, calcolato in funzione di un clock di periodo pari a **100 ns**, è di circa **46 ns**.
Questo è dovuto al fatto che il modulo lavora su **entrambi** i fronti di clock e vuol dire che nel caso **peggiore** il ritardo complessivo è di circa **4 ns** ogni semi-periodo ($50\,\text{ns} - 46\,\text{ns} = 4\,\text{ns}$) ed è quindi possibile aumentare la frequenza del clock fino ad arrivare a un periodo di ***8.02 ns*** senza andare incontro a problemi di timing.

Setup		**Hold**		**Pulse Width**	
Wost Negative Slack (WNS):	45.992 ns	Worst Hold Slack (WHS):	0.155 ns	Worst Pulse Width Slack (WPWS):	49.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	62	Total Number of Endpoints:	62	Total Number of Endpoints:	39

Tabella 1: Timing Report

Nella **Tabella 2** è invece possibile vedere che il modulo **non** fa uso di alcun *Latch*, mentre i *Flip Flop* utilizzati sono **38** e le LUTs *(Look Up Tables)* **56**.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	56	0	134600	0.04
LUT as Logic	56	0	134600	0.04
LUT as Memory	0	0	46200	0.00
Slice Registers	38	0	269200	0.01
Register as Flip Flop	38	0	269200	0.01
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Tabella 2: Slice Logic

3.2 Simulazioni

Per verificare il corretto funzionamento del modulo sono stati eseguiti dei test sui seguenti **corner case**:

3.2.1 Sequenza nulla

Per gestire il caso in cui la lunghezza della sequenza da elaborare sia **0** c'è una transizione che dallo stato ***S2*** salta direttamente a ***S7*** nel caso in cui ***last = 1*** (vero in questo caso poiché $N = M = 0$ e quindi ***reg0*** = ***reg1***). Senza di questa il modulo avrebbe continuato verso ***S3*** cominciando erroneamente l'elaborazione della parola all'indirizzo **1** della memoria per poi incrementare ***reg1*** rendendo impossibile il verificarsi di ***reg0*** = ***reg1*** (a meno di overflow di ***reg1***) e l'uscita dal loop $S3 \rightarrow S4 \rightarrow S5 \rightarrow S6$.

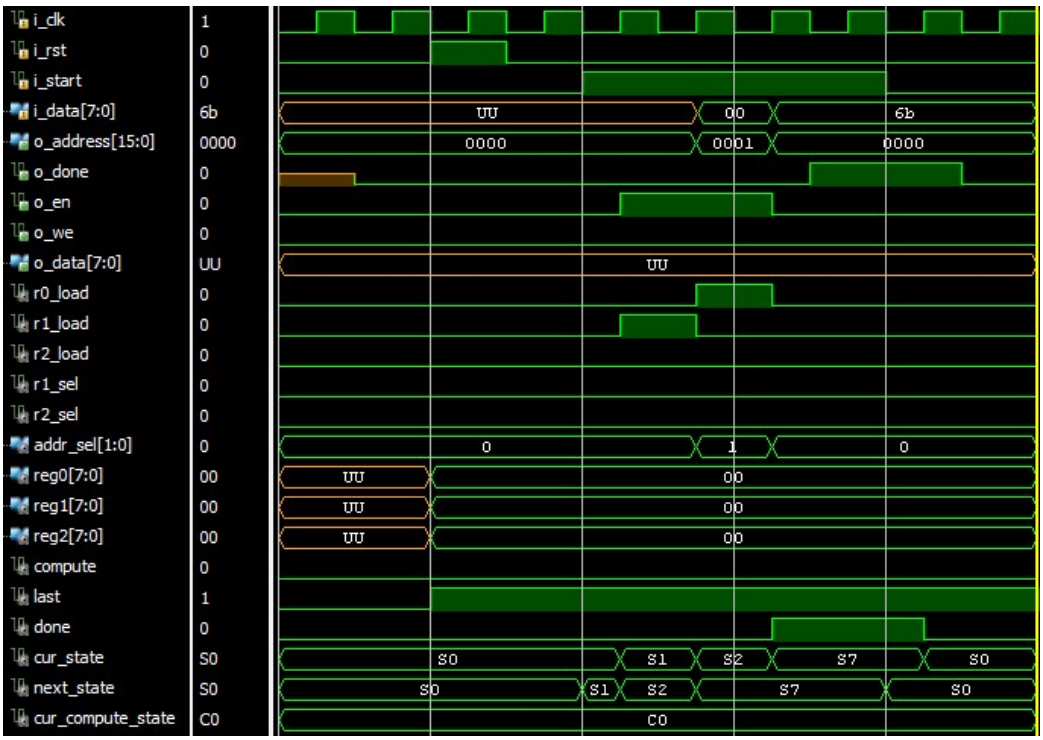

Figura 5: Simulazione con sequenza di lunghezza 0

3.2.2 Sequenza massima

Questo test va a verificare il corretto funzionamento del caso opposto al precedente, ovvero quando viene richiesto di elaborare una sequenza lunga **255 Byte** *(N = 255)*, ovvero il **massimo** possibile da specifica.
Utilizzando registri da **8 bit** sono in grado di memorizzare su ***reg0*** la lunghezza N di qualsiasi sequenza da 0 a 255 (2^8-1) e il problema non si pone nemmeno per ***reg1***, il contatore M delle parole elaborate, infatti $0 \leq M \leq N \leq 255$.

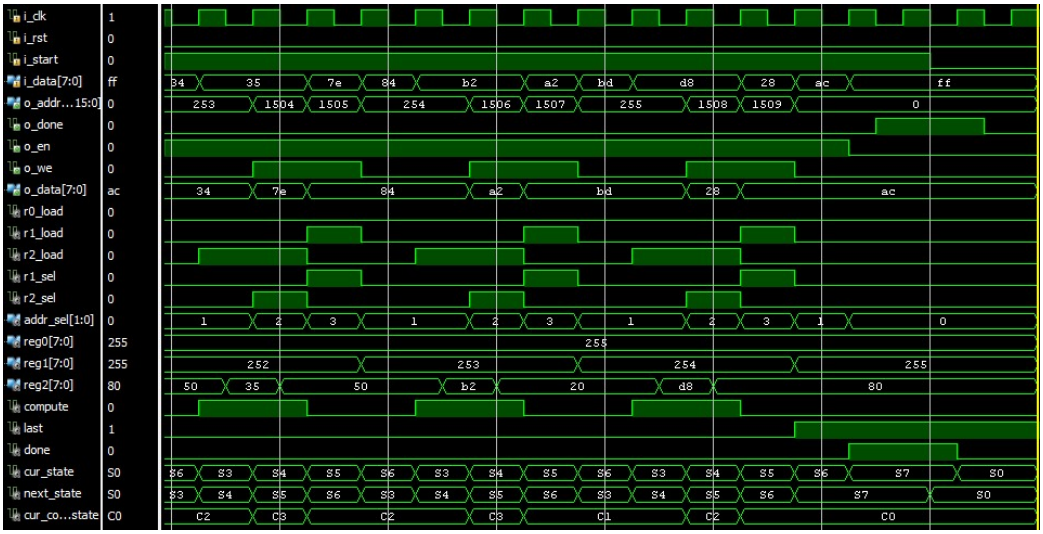

Figura 6: Parte finale della simulazione con sequenza di massima lunghezza

3.2.3 Reset asincrono

Questo test verifica che a seguito di un segnale di ***reset asincrono*** tutti i segnali tornino istantaneamente ai loro valori di default.
In **Figura 7** è possibile vedere come, a seguito del segnale di **reset**, la simulazione riparta subito da capo.

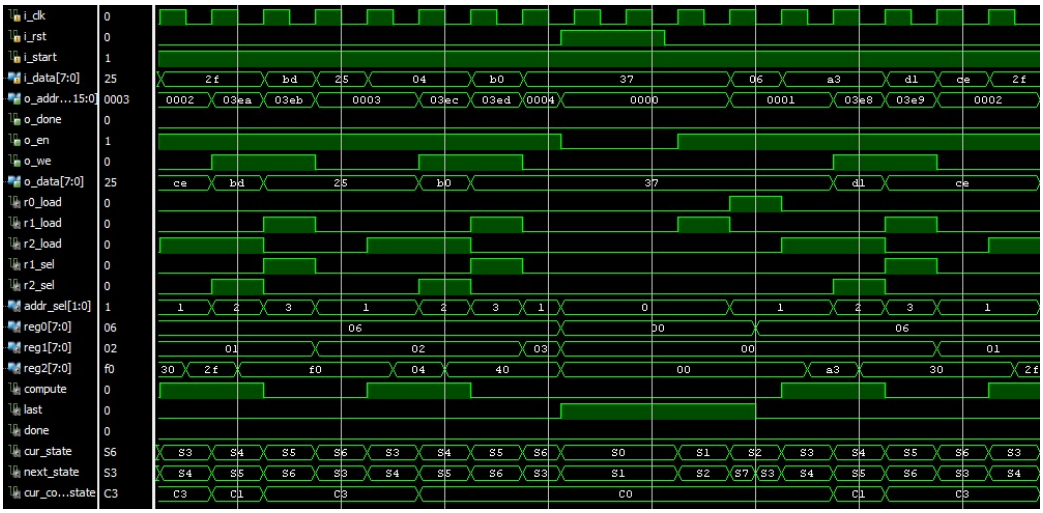

Figura 7: Simulazione con reset asincrono

3.2.4 Elaborazione multipla

Siccome una volta terminata un'elaborazione deve essere possibile poterne richiedere un'altra **senza** resettare il modulo, bisogna verificare anche questo caso. In particolare, prima di eseguire questo test, ***cur_compute_state*** non veniva reimpostato a *C0*, problema ora gestito **correttamente** tramite l'aggiunta di ***last = 1*** come sua condizione di reset (controllato in modo **sincrono** perché in timing presenta spesso degli *spike*).

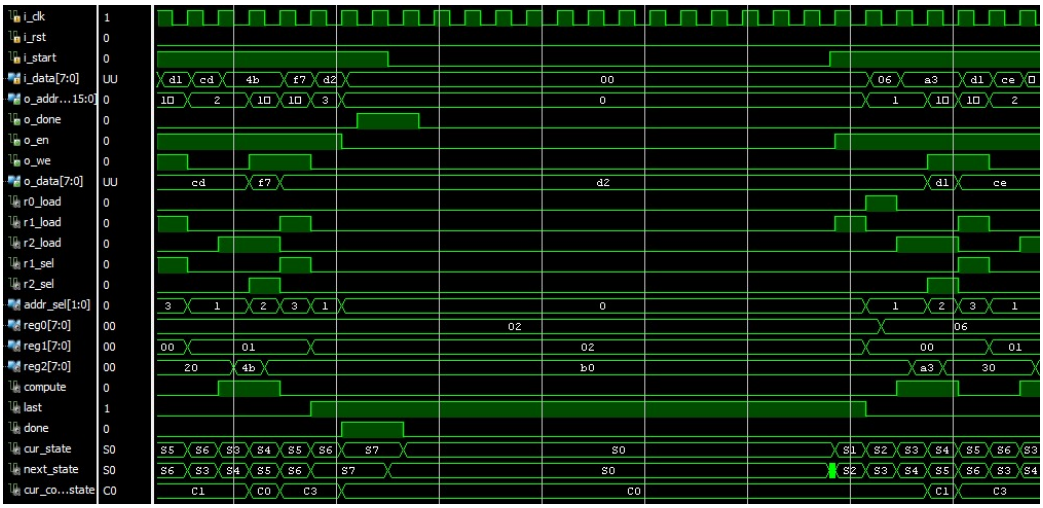

Figura 8: Passaggio a nuova elaborazione

3.2.5 Altri test

Per testare a fondo il modulo ho inoltre utilizzato un generatore per generare **10000** test, tutti passati correttamente in ***Post-Synthesis Timing.***

4 Conclusioni

Il modulo sintetizzato funziona correttamente con **tutti** i test provati sia in ***Behavioral*** che in ***Post-Synthesis*** *(functional e timing)* e le ottimizzazioni che ho effettuato vanno a ridurre quanto più mi è stato possibile il ***numero di stati***.

Avrei potuto ridurre il ***WNS*** introducendo più stati e sfruttando solo i **fronti di salita** del clock ma, siccome il componente da specifica funzionerà con un clock di 100 ns lasciando un ampio margine di tempo, ho preferito questa soluzione che riesce ad elaborare una parola ogni **4 cicli di clock**.

Si poteva inoltre ridurre la dimensione del registro ***reg2*** a **4 bit** risparmiando **4 Flip Flop** ma questo sarebbe costato una lettura da memoria in più e, per questo motivo, ho preferito fare una **singola** lettura salvando tutti gli **8 bit** anche se ne vengono utilizzati solo **4** alla volta.