POLITECNICO
MILANO 1863

Prova Finale (Progetto di Reti Logiche)

Prof. Fabio Salice - Anno 2021/2022

Roberto Giandomenico (Codice Persona - Matricola)

Indice

1 Introduzione

1.1 Scopo del progetto

Sia data una sequenza di parole da 8 bit ciascuna.
L'obiettivo del progetto è l'implementazione di un componente hardware descritto in VHDL che, letti i dati da memoria, sia in grado di applicarvi l'algoritmo di codifica convoluzionale con rapporto $\frac{1}{2}$ ed infine restituisca il risultato scrivendolo in memoria.

1.2 Descrizione generale

La memoria da cui il componente legge e su cui scrive i dati è sincrona, a indirizzamento a byte e con uno spazio degli indirizzi a 16 bit. Il componente, servendosi della memoria, segue in ordine i seguenti passaggi:

1. legge all'indirizzo di memoria `0` il numero di parole da codificare;
2. partendo dall'indirizzo `1`, legge e serializza le parole ottenendo una sequenza di bit;
3. su questa sequenza applica la codifica convoluzionale $\frac{1}{2}$ raddoppiando, di fatto, il numero di bit;
4. la nuova sequenza di bit viene infine scritta in memoria a partire dall'indirizzo `1000`.

Questa procedura viene ripetuta per tutte le parole presenti. Il numero massimo di parole da dover codificare è 255.
Inoltre il componente deve funzionare correttamente con un periodo di clock di almeno `100 ns`.

1.3 Codifica convoluzionale

Il codificatore convoluzionale con tasso di trasmissione $\frac{1}{2}$ è un componente che per ogni bit che riceve in ingresso genera la sua codifica in 2 bit. Si può rappresentare come una macchina a stati di Mealy con un bit per l'ingresso e 2 bit per l'uscita come in Figura 1.

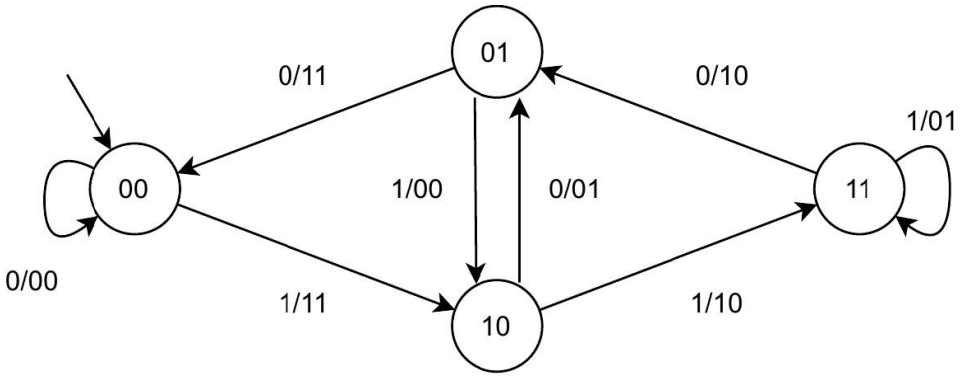

Figura 1: Codifica convoluzionale con rapporto $\frac{1}{2}$

1.4 Dati e memoria

Il componente deve leggere da una memoria con indirizzamento al byte. Si assume come prerequisito che il contenuto della memoria non verrà modificato durante l'esecuzione. La quantità di parole `W` da codificare è memorizzata all'indirizzo `0`. Il primo byte della sequenza è memorizzato all'indirizzo `1`. Lo stream di parole in uscita, invece, deve essere memorizzato a partire dall'indirizzo `1000`.
Si legge da una cella della RAM scrivendo l'indirizzo in `o_address` e abilitandone la lettura, cioè ponendo `o_en`=1 e `o_we`=0.
Si scrive in una cella della RAM scrivendo l'indirizzo in `o_address` e abilitandola in scrittura, cioè ponendo `o_en`=1 e `o_we`=1.
Come si può notare dalla Figura 2, l'ultimo indirizzo utilizzato della memoria è $1000 + 2 \times (W - 1) + 1$ che, dato $W_{max} = 255$ da specifica, può valere al massimo 1509.

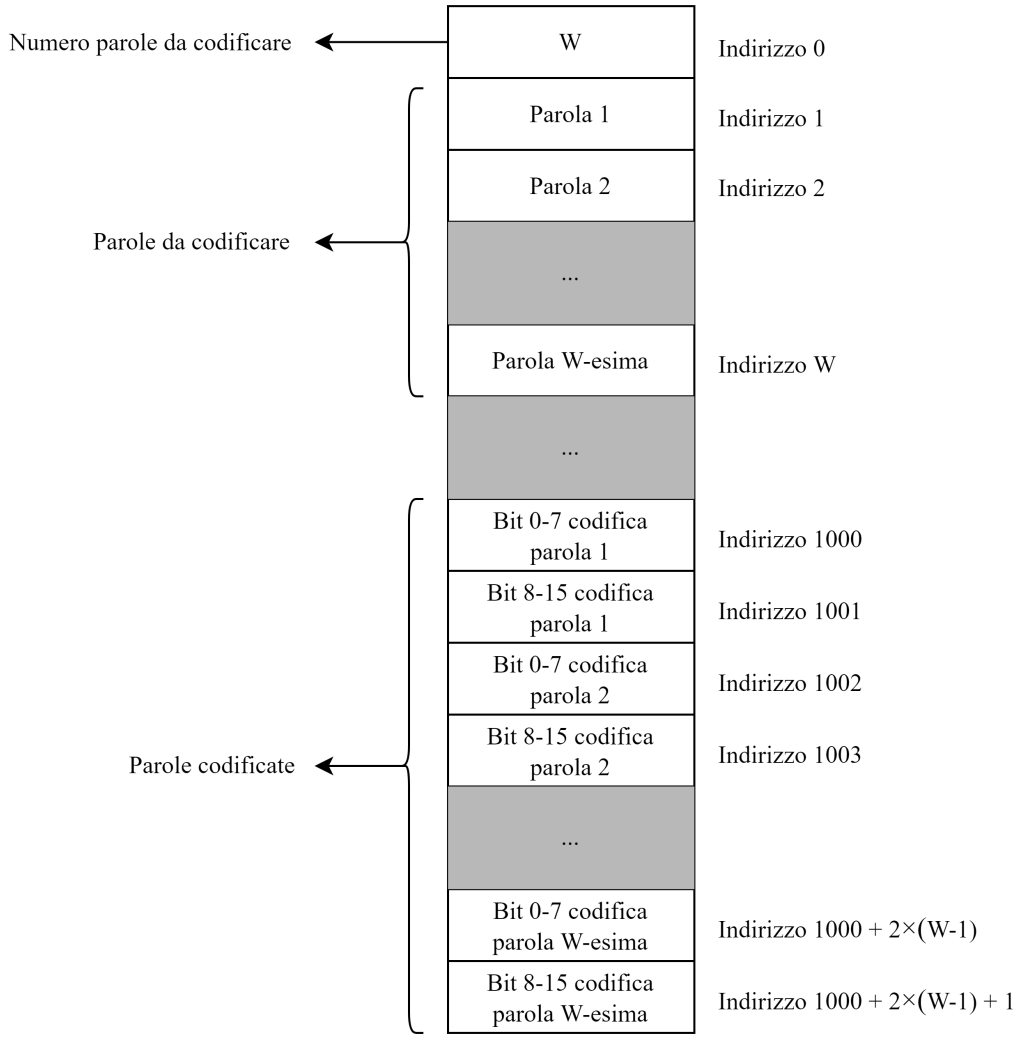

Figura 2: Indirizzi della RAM rilevanti

1.5 Esempio

Il componente legge dalla memoria (Tabella 1) all'indirizzo `0` il numero di parole totale da codificare. Successivamente partendo dall'indirizzo `1` legge la prima parola, la quale viene serializzata e il singolo bit viene dato in ingresso al codificatore convoluzionale come in Tabella 3. Quest'ultimo produce due bit in ogni istante `t`, i quali vengono scritti in memoria a partire dall'indirizzo `1000`. Una volta codificata la prima parola il codificatore continua con le successive fino ad averle codificate tutte.
Il contenuto della memoria alla fine della codifica è riportato in Tabella 2.

Indirizzo	Contenuto
0	00000010
1	10100010
2	01001011

Tabella 1: Contenuto iniziale memoria

Indirizzo	Contenuto
1001	11010001
1002	11001101
1003	11110111
1004	11010010

Tabella 2: Contenuto finale memoria

t	0	1	2	3	4	5	6	7
Uk	1	0	1	0	0	0	1	0
P1k	1	0	0	0	1	0	1	0
P2k	1	1	0	1	1	0	1	1
t	8	9	10	11	12	13	14	15
Uk	0	1	0	0	1	0	1	1
P1k	1	1	0	1	1	0	0	1
P2k	1	1	1	1	1	1	0	0

Tabella 3: Codifica convoluzionale al tempo t

1.6 Interfaccia del componente

Il componente descritto presenta la seguente interfaccia:

```
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;
```

In particolare:

- `i_clk` è il segnale di `CLOCK` in ingresso generato dal Test Bench;
- `i_rst` è il segnale di `RESET` invocato sempre all'inizio della computazione e talvolta durante la codifica;
- `i_start` è il segnale di `START` che fa partire il processo di codifica;
- `i_data` è il vettore a 8 bit tramite cui è possibile leggere il dato da memoria;
- `o_address` è il vettore in uscita che serve ad indicare l'indirizzo di memoria desiderato;
- `o_done` è il segnale di notifica che comunica la fine dell'elaborazione di tutte le parole in uscita;
- `o_en` è il segnale di `ENABLE` che, quando posto ad `1`, permette di comunicare con la memoria;
- `o_we` è il segnale di `WRITE ENABLE` che, quando posto ad `1`, permette la scrittura in memoria; deve essere `0` in caso di sola lettura;
- `o_data` è il vettore contenente la stringa da salvare in memoria.

L'elaborazione inizia quando viene alzato il segnale `i_start` e termina quando il componente alza il segnale `o_done`. Solo successivamente il segnale `i_start` dovrà essere abbassato.

2 Architettura

Per la progettazione del componente è stata scelta l'implementazione tramite una macchina a stati finiti (FSM) di Mealy non completamente specificata. Infatti il valore delle uscite è funzione degli ingressi e per alcune transizioni è indifferente.

2.1 Macchina a stati

La macchina implementata (Figura 3) è composta dai seguenti 13 stati principali.

2.1.1 `IDLE` state

Stato iniziale e di default della macchina, in cui si attende il segnale di `i_start` e in cui si torna in caso di ricezione di un segnale di `reset`.

2.1.2 `W_SAVE` state

Stato in cui si legge e si salva il numero di parole da codificare, presente all'indirizzo `0`. Nel caso questo numero sia nullo, si alza il segnale `o_done`. In caso contrario si può salvare `W` nel registro ausiliario `words_num`.

2.1.3 `WORD_READ_SETUP` state

Stato in cui si pongono `o_en`=1 e `o_we`=0. L'array `o_address` sarà l'indirizzo della parola che si vuole leggere.

2.1.4 `WORD_READ` state

In questo stato si effettua la lettura della parola e la si salva nel registro `loaded_word`.

2.1.5 `ENCODER_STATE` state

Questo stato segue il codificatore convoluzionale durante il suo percorso e può assumere il valore di uno dei quattro sotto-stati `S00`, `S01`, `S10` o `S11`. In input riceve il bit specifico della parola da codificare.

2.1.6 `RECORD_ENCODED_BIT` state

Stato in cui si aggiorna il registro da 16 bit contenente le due parole finali con i due bit appena ricevuti in output dal codificatore convoluzionale.

2.1.7 `UPDATE_BIT_COUNTER` state

Stato in cui si aggiorna il numero di bit già letti e processati della singola parola.

2.1.8 `CHECK_WORD_END` state

Si controlla quanti bit sono stati letti e codificati. Non appena si è completata la lettura dei primi 4 bit, si è pronti per scrivere la prima parola in memoria (codificata in 8 bit). Se sono stati letti tutti gli 8 bit dell'attuale parola presa in input si è, di fatto, conclusa la codifica ed è possibile registrare anche la seconda parte della parola in memoria.

2.1.9 `WRITE_WORD_1` state

Stato in cui si pongono `o_en`=1 e `o_we`=1 in modo da poter registrare in memoria i primi 8 bit della codifica, cioè la prima parola in output all'indirizzo $1000 + 2 \times words\_counter$.

2.1.10 `WRITE_WORD_2` state

Stato in cui si pongono `o_en`=1 e `o_we`=1 in modo da poter registrare in memoria i successivi 8 bit della codifica, cioè la seconda parola in output all'indirizzo $1000 + 2 \times words\_counter + 1$.

2.1.11 `UPDATE_COUNTER` state

Si incrementa il contatore delle parole lette fino a questo momento in attesa di un nuovo confronto con il totale numero di parole da leggere.

2.1.12 `CHECK_END` state

Si verifica se tutte le parole sono già state codificate. In caso affermativo si pone `o_done`=1. In caso contrario si continua con la codifica delle successive parole.

2.1.13 `DONE` state

Stato per la terminazione di un'istanza di computazione ponendo il segnale `o_done`=1. Riporta allo stato di `IDLE` non appena riceve il segnale `i_start`=0.

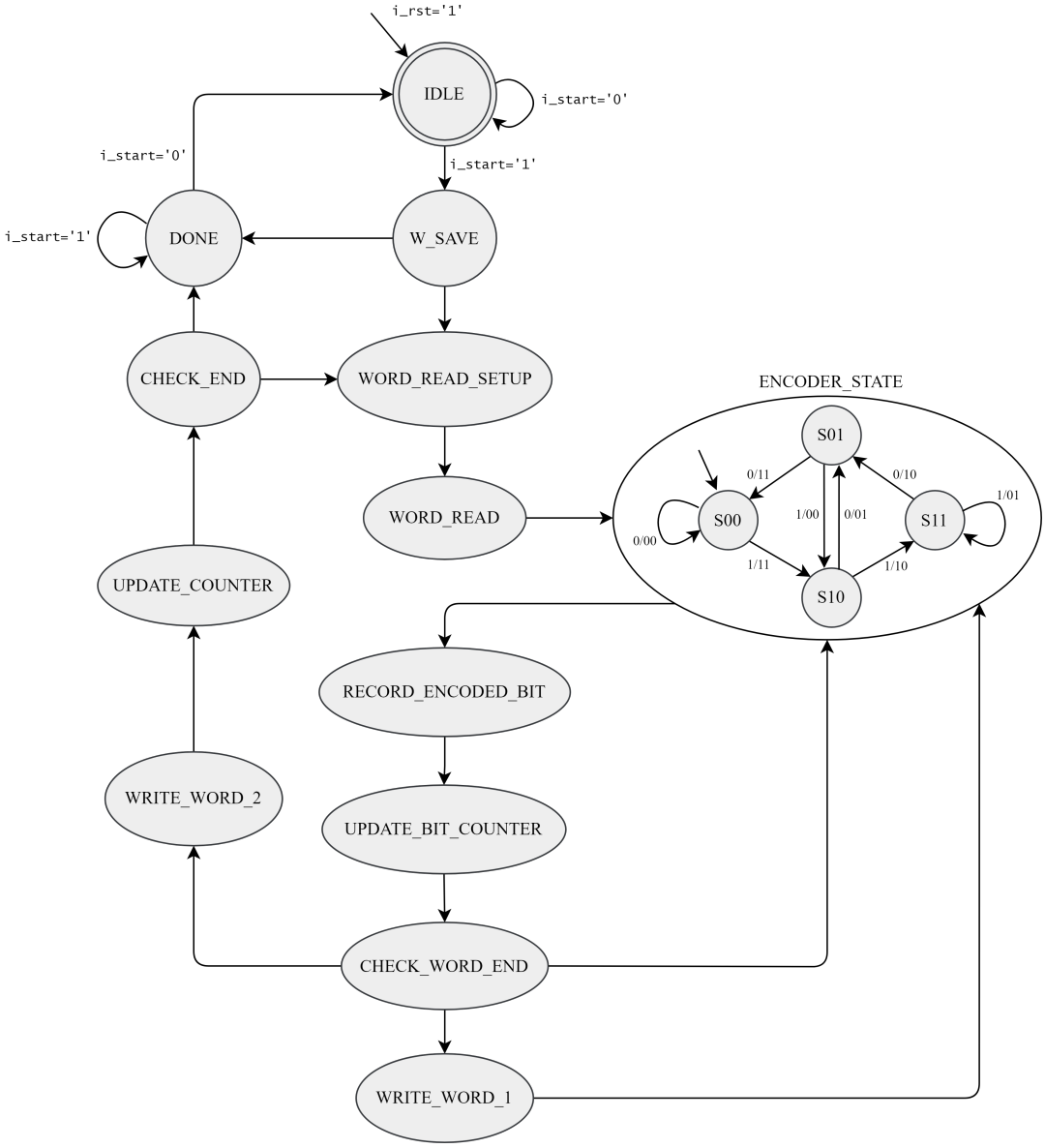

Figura 3: Macchina a stati finiti

2.2 Variables e signals

Sono qui riportati i signals utilizzati come ausilio per l'elaborazione dei dati.

- `words_num`: registro in cui si salva il valore di `W` dopo la sua lettura;
- `words_counter`: registro in cui si salva il numero delle parole lette e codificate fino a quel momento;
- `loaded_word`: registro in cui viene salvata la parola letta dalla memoria RAM e da codificare;
- `word_bit_counter`: registro che conta il numero bit già codificati della parola in input;
- `encoder_state`: memorizza l'ultimo stato visitato del codificatore convoluzionale;
- `encoder_output`: registro in cui si salva l'output su 2 bit del codificatore;
- `encoded_word`: registro in cui si salva, durante il processo di codifica, l'intera parola codificata su 16 bit.

3 Sintesi

3.1 Tool e FPGA

La sintesi e l'implementazione sono state fatte usando il software *VIVADO 2016.4*. La FPGA target utilizzata è la Artix-7 FPGA xc7a200tfbg484-1.

3.2 Report di utilizzo

Effettuando la sintesi risulta il seguente utilizzo:

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	94	0	0	134600	0.07
LUT as Logic	94	0	0	134600	0.07
LUT as Memory	0	0	0	46200	0.00
Slice Registers	86	0	0	269200	0.03
Register as Flip Flop	86	0	0	269200	0.03
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

Come si può notare dal report sull'utilizzo, il codice VHDL è stato scritto in modo da evitare l'inferenza di latch, con il preciso scopo di rendere l'intero componente sincronizzato sul fronte di salita del clock. Sono stati così evitati gli effetti di propagazioni indesiderate dei segnali.
Tutte le percentuali di utilizzo risultano essere molto minori del 100%, dunque il modulo implementato occupa solo una piccola parte della FPGA.

3.3 Report di timing

Dal report si ottiene:

```
Slack (MET) :     96.283ns  (required time - arrival time)
```

Analizzando il report sul timing possiamo notare che il *Worst Negative Slack (WNS)* è di `96.283ns`, il quale sta ad indicare che il componente potrebbe funzionare correttamente anche con periodi di clock molto più bassi. In particolare, il componente è in grado di funzionare ad una frequenza di clock f_{clk} maggiore di quella data da specifica che è pari a $\frac{1}{100ns} = 10$ MHz. Più precisamente, il minimo periodo utilizzabile risulta:

$$T_{min} = T_{clk} - WNS = 100ns - 96.283ns = 3.717ns$$

e quindi si ottiene la massima frequenza di funzionamento:

$$f_{max} = \frac{1}{T_{min}} = \frac{1}{3.717ns} \approx 269.03MHz$$

4 Testing del componente

Allo scopo di verificare il corretto funzionamento del componente sintetizzato, il codice è stato testato dapprima con i Test Bench forniti insieme alle specifiche e in seguito con test costruiti ad hoc aventi i seguenti obiettivi:

1. verificare i casi limite;
2. verificare il corretto funzionamento dei segnali;
3. verificare la solidità del componente.

4.1 Test e casi limite

Le modalità di simulazione eseguite sono due: *Behavioural* e *Post-Synthesis Functional*. Di seguito sono riportati i casi di test più significativi.

4.1.1 Sequenza minima

Nel caso della sequenza minima viene scritto zero all'indirizzo `0`. In questo modo il componente, come in Figura 4, dopo aver letto che il numero di parole da elaborare è zero, alza il segnale `o_done` e termina la codifica.

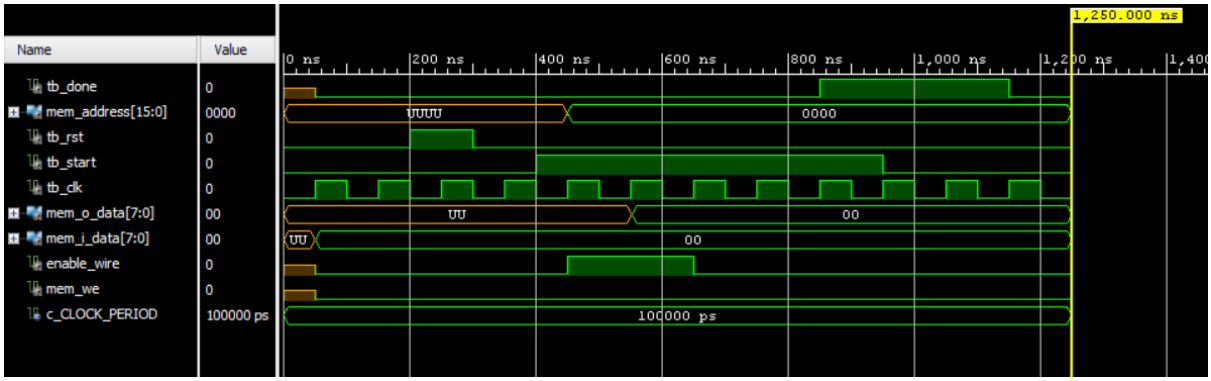

Figura 4: Simulazione sequenza minima

4.1.2 Sequenza massima

Nel caso di sequenza massima viene inserito 255 all'indirizzo `0`. Come mostrato in Figura 5, il componente riesce a codificare tutte le parole compresa l'ultima.

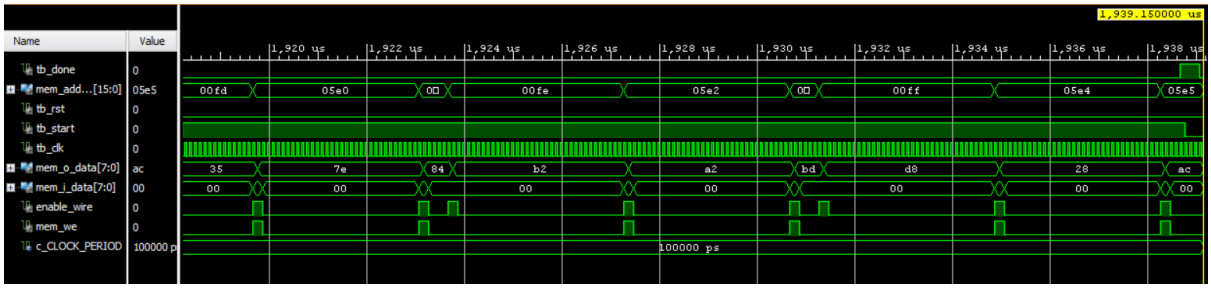

Figura 5: Estratto simulazione sequenza massima

4.1.3 Reset asincrono

Il componente riceve durante la codifica un segnale di `reset` e, come in Figura 6, riesce a ricominciare la codifica correttamente dopo aver ricevuto un nuovo segnale di `start`.

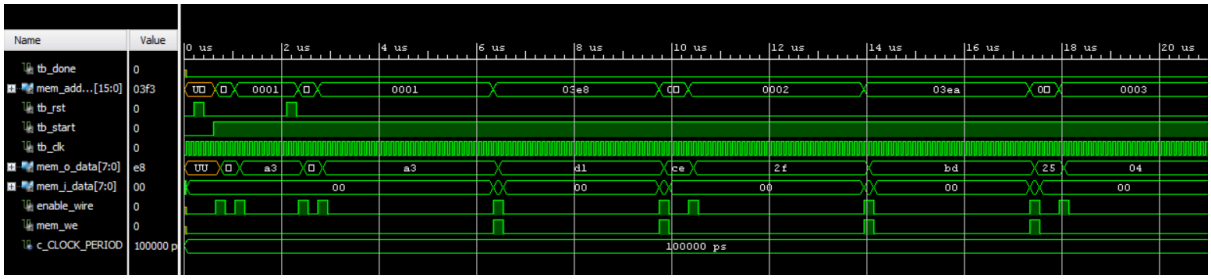

Figura 6: Estratto simulazione reset

4.1.4 Codifica continua

Viene testata la capacità del componente di codificare più sequenze una dopo l'altra senza ricevere un segnale di `reset`. Il componente, una volta terminata una sequenza, riesce ad iniziare correttamente la codifica di un nuovo flusso dopo aver ricevuto un nuovo segnale di `start`. Un estratto è mostrato in Figura 7.

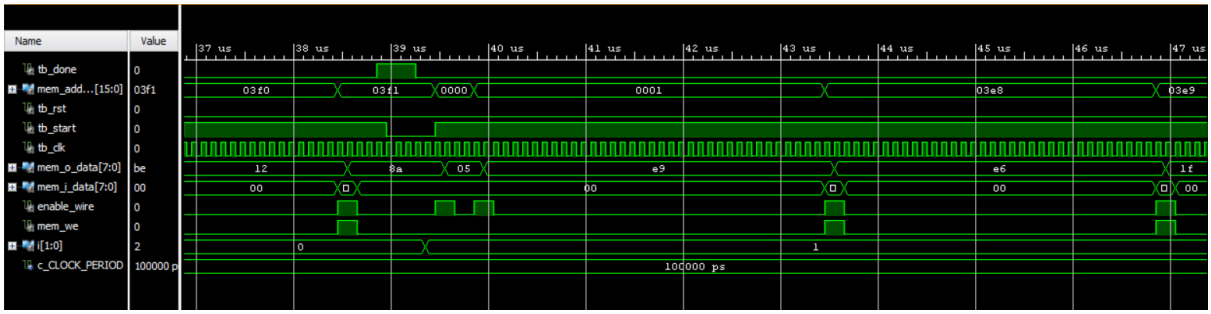

Figura 7: Estratto simulazione di codifica continua

4.1.5 Test Casuali

Per aumentare la probabilità di trovare eventuali errori, il componente è stato sottoposto a numerosi Test Bench casuali generati utilizzando un applicativo scritto in Python.
La correttezza di tali simulazioni è stata poi verificata tramite la lettura del file di log.

4.2 Osservazioni

Il componente supera correttamente tutti i test proposti in precedenza sia a livello *Behavioral* che *Post-Synthesis*. Quest'ultimo tipo di simulazione è stato essenziale per verificare che eventuali ottimizzazioni applicate dal tool in fase di sintesi non influenzino il corretto funzionamento del componente.

5 Conclusione

Il componente descritto nelle specifiche è stato sintetizzato con successo e tutte le simulazioni effettuate terminano con esito positivo.
La trasformazione della specifica in una macchina a stati finiti ha semplificato notevolmente la fase di architettura del modulo. Inoltre l'architettura è stata pensata per diminuire il più possibile l'overhead alla codifica utilizzando il minimo numero di stati possibili e minimizzando l'utilizzo di risorse, abbattendo così i tempi d'esecuzione del modulo e l'area occupata.
Dal punto di vista del design, si è scelto di utilizzare un'unica FSM senza componenti esterni, racchiudendo tutto in un solo processo.
Infine è possibile notare che il componente è in grado di funzionare ad una frequenza di clock molto più alta, all'incirca di 269.03 MHz.