Prova Finale (Progetto di Reti Logiche)

Prof. Fabio Salice - Anno 2021/2022

POLITECNICO
MILANO 1863

Manuela Merlo (Codice Persona 10670533 - Matricola 936925)

Matteo Mormile (Codice Persona 10666565 - Matricola 934238)

Indice

1 Introduzione

1.1 Scopo del progetto

Lo scopo del progetto è di implementare un componente hardware, descritto in VHDL. L'obiettivo del modulo è, prese in ingresso una sequenza continua di W parole ognuna da 8 bit, restituire una sequenza di Z=W*2 parole mediante l'utilizzo di un codice convoluzionale con tasso di trasmissione 1/2 , ovvero un codice in cui per ogni bit in ingresso ne vengono generati due in uscita.

1.2 Specifiche generali

Il modulo legge da una memoria con indirizzamento al byte una sequenza di parole, ognuna delle quali viene serializzata generando un flusso continuo U da 1 bit. Su questo flusso viene applicato il codice convoluzionale $\frac{1}{2}$ mediante un convolutore, una macchina sequenziale sincrona con un clock globale e un segnale di reset con il seguente diagramma degli stati :

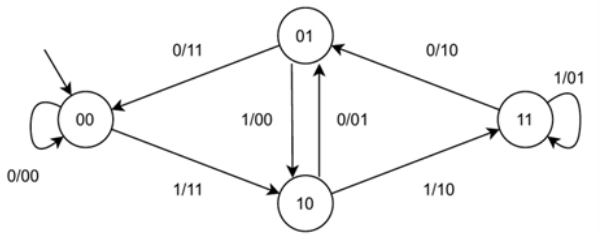


Figura 1: Convolutore

Il flusso di bit in uscita dal convolutore viene poi riparallelizzato dal modulo in parole da 8 bit che vengono salvate in memoria a partire dall'indirizzo 1000.

1.3 Esempio di elaborazione

Il seguente esempio, mostra il contenuto della memoria al termine dell'elaborazione di una sequenza di due parole :

W 10100010 01001011

Z 11010001 11001101 11110111 11010010

Contenuto	**Valore**	**Indirizzo**
Numero parole	2	0
Prima parola	162	1
Seconda parola	75	2
...	...	...
Codifica I prima parola	209	1000
Codifica II prima parola	205	1001
Codifica I seconda parola	247	1002
Codifica I seconda parola	210	1003

Per semplicità, riportiamo i valori in decimale anche se in memoria sono rappresentati usando una codifica binaria su 8 bit senza segno.

Esempio di codifica di una parola: 10100010

`Clock`	0	1	2	3	4	5	6	7
`input_encoder`	1	0	1	0	0	0	1	0
`output_encoder[0]`	1	0	0	0	1	0	1	0
`output_encoder[1]`	1	1	0	1	1	0	1	1

1.4 Interfaccia del componente

Il componente 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;
```

Nello specifico:

- il nome del modulo è `project_reti_logiche`;
- `i_clk` è il segnale di `CLOCK` in ingresso generato dal TestBench;
- `i_rst` è il segnale di `RESET` che inizializza la macchina pronta per ricevere il primo segnale di `START`;
- `i_start` è il segnale di `START` generato dal TestBench;
- `i_data` è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- `o_address` è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- `o_done` è il segnale che comunica la fine dell'elaborazione e la scrittura dell'output in memoria;
- `o_en` è il segnale di `ENABLE` da inviare alla memoria per poter comunicare sia in lettura che in scrittura;
- `o_we` è il segnale di `WRITE ENABLE` per abilitare la scrittura in memoria;
- `o_data` è il segnale (vettore) di uscita dal componente verso la memoria.

Componente e memoria vengono poi collegati sfruttando i pin in uscita `o_address`, `o_en`, `o_we` e il pin in ingresso `i_data`.

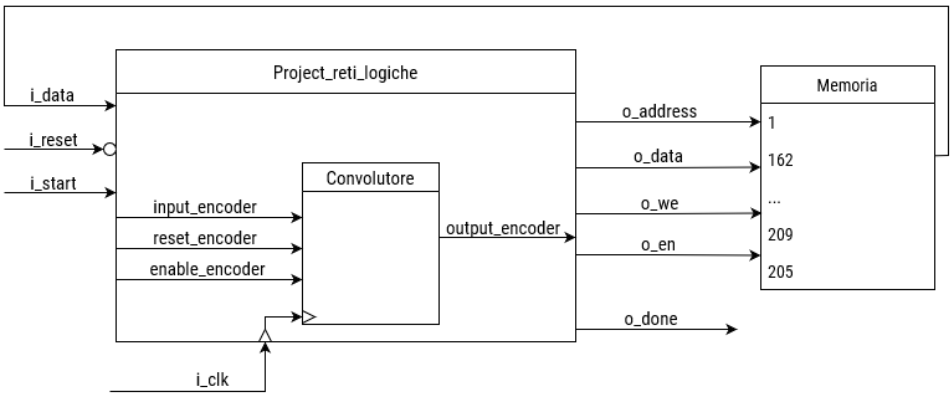
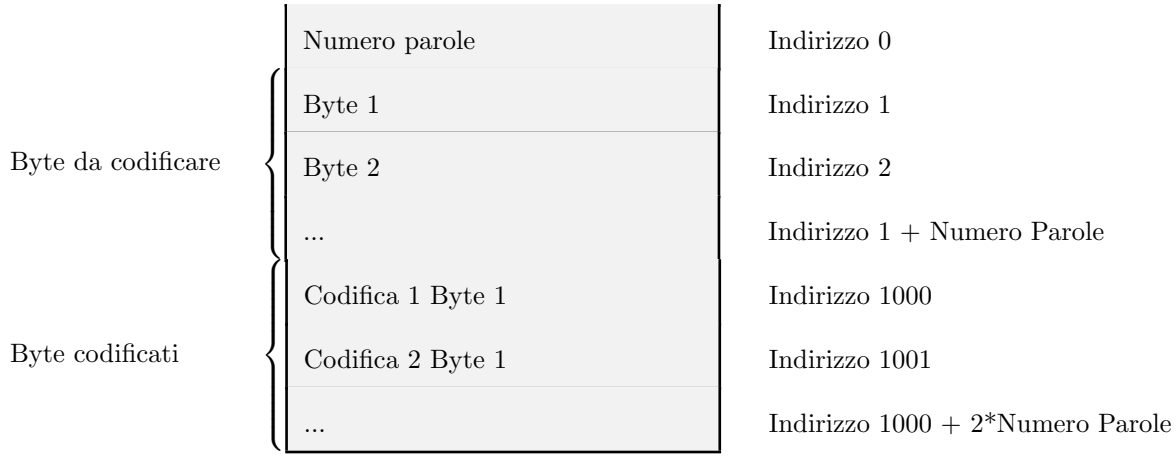


Figura 2: Interazione memoria - modulo

2 Dati e descrizione memoria

Il modulo da implementare si interfaccia con una memoria con indirizzamento al byte da cui legge uno stream di parole avente dimensione massima 255 byte.



In particolare:

- L'indirizzo 0 contiene il numero di parole W da codificare;
- Gli indirizzi $[1;1+W]$ contengono le sequenze di byte da codificare;
- Gli indirizzi $[1000;1000+2*W]$ contengono le sequenze di byte codificati.

2.1 Protocollo di inizio e fine computazione

Quando il segnale di ingresso `i_start` viene portato a 1, il componente progettato inizia la computazione spostandosi nello stato `START` in cui inizia a richiedere dati in memoria. Il segnale `i_start` rimmarrà alto fino a quando `o_done` non verrà portato alto.
Dopo aver scritto l'intero risultato in memoria, il componente alza a 1 il segnale di `o_done`, per segnalare la fine dell'elaborazione. Il segnale `o_done` rimane alto fino a quando il segnale di `i_start` non viene riportato a 0; dopodichè anche `o_done` viene abbassato.

2.2 Protocollo di accesso alla memoria

Durante la computazione è possibile accedere in memoria all'indirizzo indicato in `o_address` sia in lettura che in scrittura. In particolare:

- Se `o_en`=1 e `o_we`=0 viene letto dalla memoria il dato contenuto nel segnale `i_data`
- Se `o_en`=1 e `o_we`=1 viene scritto in memoria il dato contenuto nel segnale `o_data`

3 Architettura

Per la progettazione del componente è stata scelta l'implementazione tramite una macchina a stati finiti (FSM) che dialoga con il convolutore. Quest'ultimo è a sua volta una macchina a stati finiti che rispecchia il grafo indicato nella specifica.

3.1 FSM

La macchina è composta da 6 stati. Qui di seguito è fornita una breve descrizione qualitativa del loro funzionamento.

3.1.1 RESET state

Stato iniziale e di default della macchina, in cui si attende il segnale di `i_start` e raggiungibile a partire da ogni altro stato in caso di ricezione di un segnale di reset.

3.1.2 READ_COUNTER state

Stato in cui la macchina legge dall'indirizzo 0 della memoria e assegna al segnale contatore il numero di parole da processare.

3.1.3 READ_WORD state

Stato che a seconda del valore del contatore :

Counter $>$ 0 : legge una parola dalla memoria e decrementa il contatore;

Counter $\leq$ 0 : porta la macchina nello stato `DONE` e ferma il convolutore.

3.1.4 ENCODING state

Definiti `bits_read` come il numero di bit letti a partire dal byte in ingresso dalla memoria e `bits_saved` come il numero di bit di cui si ha a disposizione l'ouput:

bits_read $<$ 8 : Serializza i bit del byte letto nello stato `READ` per darli in input al convolutore. Ferma il convolutore in caso contrario.

bits_read $>$ 0 : Riparallelizza i bit in uscita dal convolutore e ricrea lo stream di byte da salvare in memoria.

bits_saved = 8 : Attiva la memoria in modalità scrittura, definisce l'indirizzo e passa allo stato `WRITE`.

3.1.5 WRITE state

Stato in cui la macchina scrive in memoria il valore salvato in `Reg_OUT` e a seconda del valori di `bits_read`:

bits_read $<$ 8 : La macchina a stati finiti deve finire la codifica generando la seconda parola, dunque il modello continua la serializzazione della parola e torna allo stato `ENCODING`;

bits_read $\geq$ 8 : Attiva la memoria in modalità lettura, definisce l'indirizzo di memoria corretto, riporta il numero di bit letti a 0 e passa allo stato `READ_WORD`.

3.1.6 DONE state

Stato che determina la fine dell'eleborazione e riporta la macchina nello stato `RESET` quando viene abbassato il segnale `i_start`.

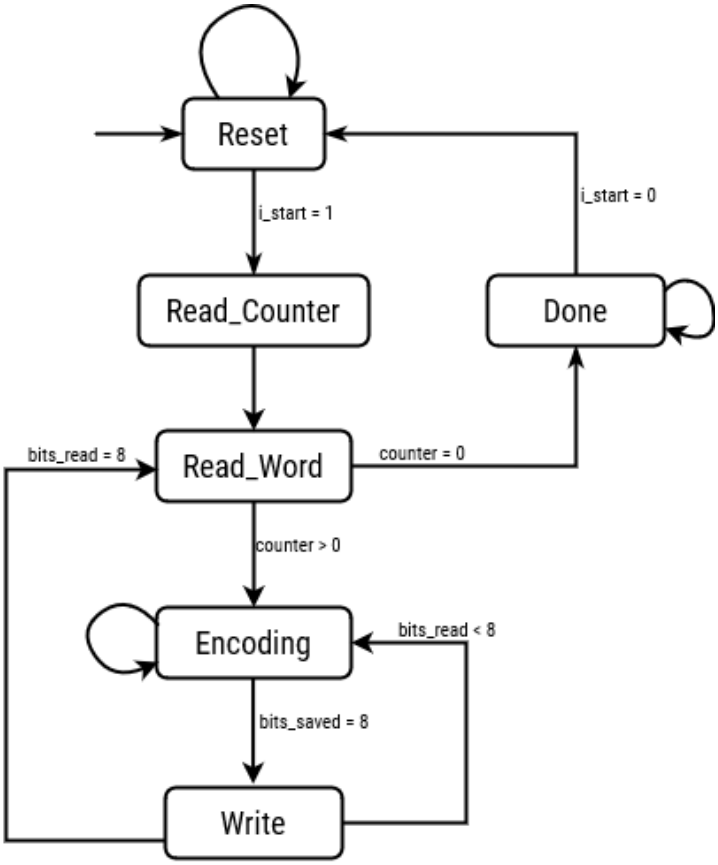


Figura 3: Macchia a Stati Finiti

3.2 Uscite di default

Determinati segnali di uscita non sono rilevanti in determinati casi. In particolare:

- se `i_start` = '0': tutte le uscite del componente non sono rilevanti
- se `o_en` = '1' e `o_we` = '0': l'uscita `o_data` non è rilevante

Quando le uscite non sono rilevanti assumono un valore di default.
Di seguito riportiamo i valori di default assegnati alle uscite e le motivazioni delle nostre scelte.

Nome	Default	Motivazione
`o_address`	others ⇒ '0'	L'indirizzo 0 è il primo indirizzo di lettura
`o_en`	0	Non accedo alla memoria se non necessario
`o_we`	0	Il primo accesso alla memoria è in lettura
`o_data`	others ⇒ '0'	Scelta arbitraria
`o_done`	others ⇒ '0'	All'inizio dell'elaborazione deve valere 0

3.3 Variabili, signals e valori di default

Di seguito sono riportati i signals e le variabili principali utilizzate e i rispettivi valori di default assegnati.
Nota: Nel progetto, per i signals che fanno riferimento a contatori, è stato aggiungo un signal con la notazione `_next` utilizzato per separare in modo più chiaro la logica combinatoria da quella sincrona.

Nome	Uso	Default	Motivazione
`current_state`	Stato attuale della FSM	Reset	Stato iniziale
`next_state`	Stato prossimo della FSM	Reset	Scelta arbitraria
`current_state_encoder`	Stato attuale del Convolutore	s00	Stato iniziale
`next_state_encoder`	Stato prossimo del Convolutore	s00	Scelta arbitraria
`input_econder`	Input Convolutore	0	Scelta arbitraria
`enable_encoder`	Segnale di enable del convolutore	0	Il convolutore all'inizio del processo non deve elaborare alcun input
`output_encoder`	Output del Convolutore	00	Scelta arbitraria
`reset_encoder`	Segnale di Reset del convolutore	0	Il convolutore non deve andare nello stato di Reset
`address_IN`	Indirizzo di lettura	others ⇒ 0	Primo indirizzo di lettura
`address_OUT`	Indirizzo di scrittura	0000001111101000	Primo indirizzo di scrittura
`counter`	Contiene il numero di parole da processare	others ⇒ 0	Scelta arbitraria
`bits_read`	Contiene il numero di bit serializzati a partire dalla parola letta da memoria	others ⇒ 0	Numero di bit letti iniziale
`bits_saved`	Contiene il numero di bit processati a partire dalla parola letta da memoria	others ⇒ 0	Numero di bit processati inizialmente
`reg_IN`	Registro di input	others ⇒ 0	Scelta arbitraria
`reg_OUT`	Registro di Output	others ⇒ 0	Scelta arbitraria

3.4 Scelte implementative

Nella realizzazione del componente come FSM si è deciso di utilizzare quattro soli processi, due per ogni macchina implementata: `project_comb`, `project_sync`, `encoder_proc` e `encoder_sync`.

Modulo Pincipale è composto da due processi:

- Il process `project_comb` calcola i valori successivi dei signals sulla base di quelli attuali e sullo stato della macchina.
- Il process `project_sync` si occupa di aggiornare ad ogni ciclo di clock i valori attuali portandoli a quelli successivi (calcolati dal processo combinatorio).

Convolutore è composto da due processi:

- Il process `encoder_comb` calcola i valori successivi di output e stato in base allo stato attuale e al valore fornitogli in ingresso dalla macchina principale precedentemente descritta;
- Il process `encoder_sync` si occupa di aggiornare lo stato e l'uscita.

4 Sintesi

4.1 Area occupata e Timing

Per la sintesi del nostro componente abbiamo utilizzato la seguente FPGA: xc7a200tlffv1156-2L, appartenente alla famiglia Artix 7 prodotta dall'azienda Xilinx.
Vivado ha calcolato l'utilizzo di 168 lookuptable e 70 registri flip flop.
Il codice VHDL è stato sviluppato in modo tale da evitare l'inserimento di latch durante la fase di

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	168	0	134600	0.12
LUT as Logic	168	0	134600	0.12
LUT as Memory	0	0	46200	0.00
Slice Registers	70	0	269200	0.03
Register as Flip Flop	70	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

Figura 4: *Report Utilization*

sintesi con lo scopo di rendere l'intero componente sincronizzato sul fronte di salita del Clock.
Per quanto riguarda il *Timing*, con uno slack di 96.002ns il componente soddisfa il vincolo sulla frequenza minima $T_{Clock} \leq 100$ ns .

4.2 Codifica degli stati

Il *Synthesis Report* di Vivado contiene, tra le varie informazioni, il numero di bit utilizzati per ciascun registro. In particolare, per il registro contenente lo stato attuale `current_state` sono stati utilizzati 3 bit e una codifica binaria naturale, più efficiente dal punto di vista spaziale di quella one-hot.

State	New Encoding
reset	000
read_counter	001
read_word	010
encoding	011
write	100
done	101
iSTATE	110

Figura 5: *Synthesis Report*

4.3 Warnings post synthesis

Quasi tutti warning generati dal tool di sintesi durante lo sviluppo sono stati risolti. Tra questi anche i più "comuni" come i warning per latch inferiti e warning per segnali presenti nel processo ma non inseriti nella sensivity list.
Il solo warning presente nella versione finale del componente è sollevato dal registro `bits_saved` che essendo un contatore con passo pari (+2), non cambierà mai il valore del bit in posizione 0.
Per chiarezza di codice si è scelto di non modificare il contatore.
Una possibile soluzione sarebbe stata trasformare l'elemento in un contatore di coppie_salvate da 0 a 4 con passo 1.

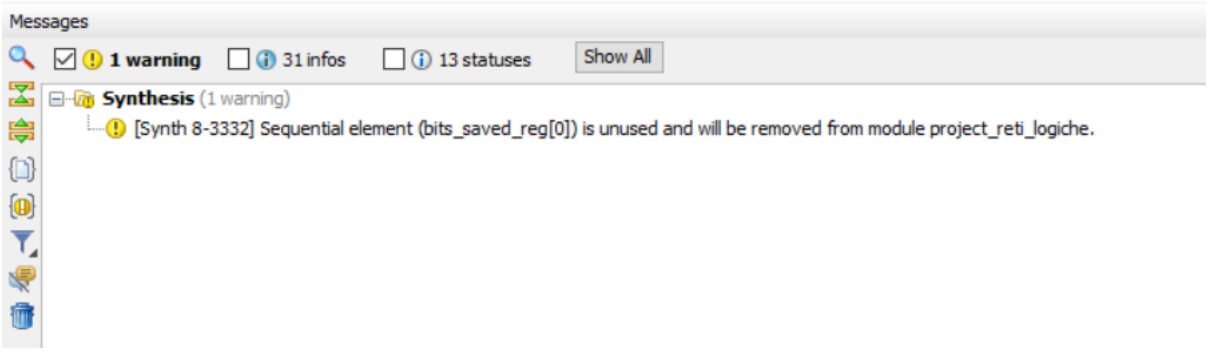


Figura 6: *Warning*

5 Simulazioni

5.1 Obiettivo

Al fine di verificare il corretto funzionamento del componente sintetizzato, il codice è stato testato dapprima con il test bench fornito insieme alle specifiche e in seguito con test realizzati ad hoc per:

1. Verificare casi limite
2. verificare il corretto funzionamento in risposta ai segnali
3. Coprire il maggior numero di cammmini possibile

5.2 Test

5.2.1 Casi Limite

Per testare i casi limite, il componente e stato sottoposto ai seguenti casi di test:

1. **Contatore = 0**
   - Behavioral Simulation :750 ns
   - Post Synthesis Fuctional Simulatioin: 751 ns

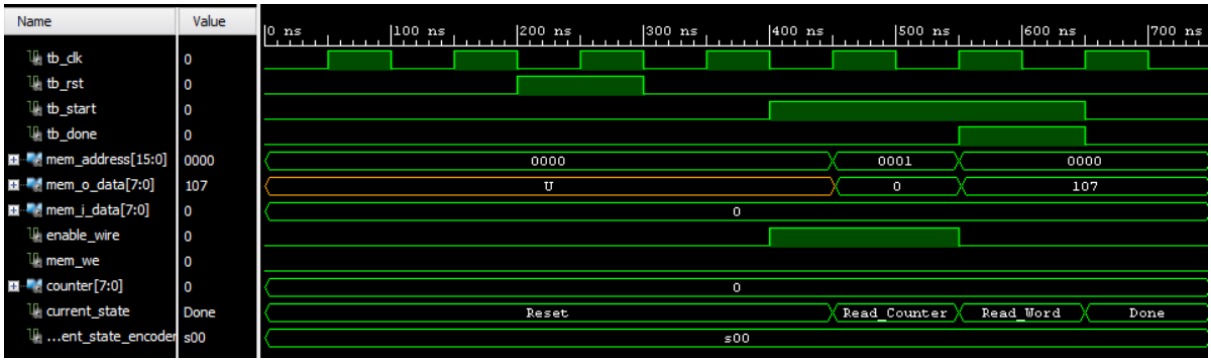


Figura 7: Caso limite : Contatore = 0

2. **Contatore = 255**
   - Behavioral Simulation : 306750 ns
   - Post Synthesis Fuctional Simulatioin: 306751 ns

5.2.2 Corretto funzionamento in risposta ai segnali

Per testare il corretto funzionamento del componente in risposta ai segnali, il componente e stato sottoposto ai seguenti casi di test:

1. **Reset asincrono**
   - Behavioral Simulation : 9859 ns
   - Post Synthesis Fuctional Simulatioin: 9860 ns
2. **Computazione multipla**
   - Behavioral Simulation : 2942.5 ns
   - Post Synthesis Fuctional Simulatioin: 2943.5 ns

5.2.3 Copertura dei cammini

Per aumentare la probabilità di trovare eventuali errori, il componente è stato sottoposto a numerosi test bench senza obiettivi specifici, generati utilizzando un applicativo scritto in python.

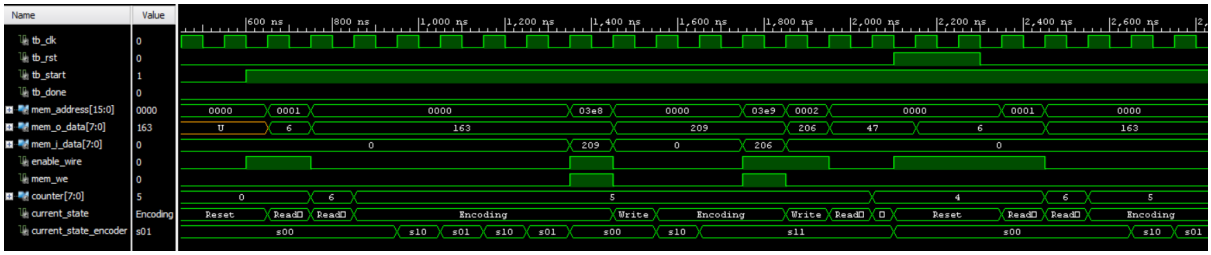


Figura 8: Reset asincrono

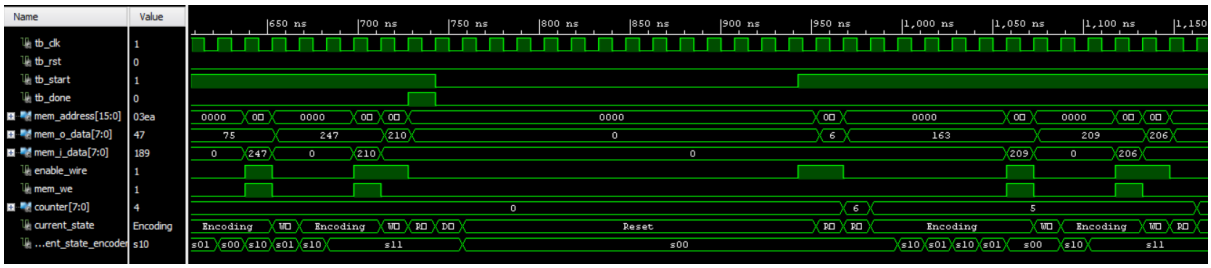


Figura 9: Computazione multipla

5.3 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 avessero influenzato il corretto funzionamento del componente.

6 Efficienza ed ottimizzazioni

Non essendo l'ottimizzazione spaziale particolarmente significativa, visto l'esiguo numero di risorse occupate, si è scelto di specificare il numero di bit salvati, ridondante rispetto al numero di bit letti, previlegiando una migliore lettura del codice.
Si è inoltre focalizzata l'attenzione sul miglioramento delle performance temporali.
In particolare si è scelto di fermare la convoluzione solo durante la fase di lettura e scrittura del secondo byte in memoria .

7 Conclusione

Il componente descritto nelle specifiche è stato sintetizzato con successo e tutte le simulazioni effettuate (sia *Behavioral* che *Functional Post-Synthesis*) terminano con esito positivo. Si può dunque affermare che il componente prodotto è in grado di interfacciarsi con una memoria avente indirizzamento al byte sfruttando il protocollo descritto nelle specifiche ed è in grado di applicare l'algoritmo di convoluzione per un numero di parole da 0 a 255.
Eventuali migliorie vanno nella direzione di ridurre il numero di LUT e FF inferiti dal tool di sintesi.