Giovanni Paolino **Codice persona**: 10696774

Ilaria Paratici **Codice persona**: 10707097

Progetto Reti Logiche

Introduzione:

Il componente ha lo scopo di leggere da memoria un numero W di parole da 8 bit ciascuna, dove il numero W è salvato all'indirizzo di memoria 0 e le parole da leggere sono salvate in sequenza a partire dall'indirizzo di memoria 1, sulle parole lette viene applicato il codice di convoluzione ½ (ogni bit viene codificato con 2 bit) in modo da ottenere una sequenza di Z parole (dove Z=2W), composte da 8 bit ciascuna, le quali devono essere salvate in memoria in sequenza a partire dall'indirizzo di memoria 1000.

Per fare ciò abbiamo utilizzato due moduli: il modulo principale del progetto, implementato come macchina a stati, che legge e scrive in memoria e gestisce la sequenza delle operazioni, e un modulo secondario, anch'esso FSM, che si occupa della codifica e il cui funzionamento è attivato e disattivato tramite un segnale settato dal modulo principale.

Elaborazione:

Operazioni da svolgere dopo che il segnale i_start è andato ad 1.

1. Lettura da memoria del numero di parole da codificare, W, contenuto dall'indirizzo di memoria o_address = 0000 0000 0000 0000.
2. Controllo di W, se W=0 allora termina, altrimenti (se W>0) inizia la lettura delle parole.
3. Lettura di una parola da memoria ed incremento del numero di parole lette.
4. Passaggio della parola letta all' FSM codificatore, il quale codifica la parola letta tramite l'utilizzo del codice di convoluzione ½; questa operazione dà come risultato due parole da 8 bit ciascuna che vengono restituite come output dell'FSM codificatore.
5. Scrittura in serie delle due parole in memoria (la prima scritta all'indirizzo mille e le successive a seguire).
6. Controllo del numero di parole lette, nel caso in cui il numero delle parole lette sia pari a W il segnale di output o_done viene posto ad 1 e il processo termina, altrimenti si riprende dalla Lettura di una parola da memoria (punto 3).

Modello del modulo FSM principale del progetto:

Usando un approccio FSM si è giunti alla seguente modellizzazione:

STATI:

- STOP: stato di pronto in attesa del segnale i_start
- A: stati di preparazione della computazione con lettura del numero delle parole (W) e allineamento dell'offset
    - A0: stato di reset e settaggio dell'indirizzo di memoria o_address=0000 0000 0000 0000 per la lettura di W
    - A1: stato intermedio per attendere che la memoria fornisca l'output

    - A2: lettura effettiva di W dal segnale i_data
    - A3: incremento dell'offset rispetto a 0000 0000 0000 0000, che indica il numero della parola da leggere, nonché il suo indirizzo

- B: stato di lettura delle parole in serie: settaggio di o_address, o_we, o_en per leggere la parola e controllo di W (se W è a 0 non c'è bisogno di fare nessuna codifica e la computazione deve terminare andando nello stato D di terminazione).
- C: stati di computazione della parola letta e creazione delle due parole di output
    - C0: inizio elaborazione degli 8 bit che compongono la parola, passando in input al codificatore il bit numero 7 e attivando il codificatore
    - C1: letto output del bit 7 e dato in input al codificatore il bit 6
    - C2: letto output del bit 6 e dato in input al codificatore il bit 5
    - C3: letto output del bit 5 e dato in input al codificatore il bit 4
    - C4: letto output del bit 4 e dato in input al codificatore il bit 3
    - C5: letto output del bit 3 e dato in input al codificatore il bit 2
    - C6: letto output del bit 2 e dato in input al codificatore il bit 1
    - C7: letto output del bit 1 e dato in input al codificatore il bit 0
    - C8: letto output del bit 0, disattivando il codificatore
- W: stati di scrittura in memoria
    - W1: scrittura in memoria della parola 1 composta dalla concatenazione degli output dei bit 7,6,5,4
    - W2 scrittura in memoria della parola 2 composta dalla concatenazione degli output dei bit 3,2,1,0
- D: stato di fine con segnale o_done alto

SEGNALI:

```
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;
```

- i_clk: segnale di clock fornito dal testbench
- i_rst: segnale di reset fornito dal testbench
- i_start: segnale di start fornito dal testbench
- i_data: dato (parola da 8 bit) letto da memoria all'indirizzo o_address nel momento in cui o_en=1 e o_we=0
- o_address: indirizzo da 16 bit cui leggere il dato da memoria / su cui scrivere il lato in memoria
- o_done: segnale che viene portato alto quando la codifica delle parole termina
- o_en: segnale che dev'essere settato alto quando si vuole che ci sia una comunicazione con la memoria in lettura o in scrittura

- o_we: segnale che dev'essere alto quando bisogna eseguire un'operazione di scrittura in memoria e dev'essere invece settato basso quando si vuole eseguire un'operazione di lettura da memoria
- o_data: dato scritto in memoria all'indirizzo o_address nel momento in cui o_en=1 e o_we=1

Utilizziamo poi i seguenti SEGNALI INTERNI:

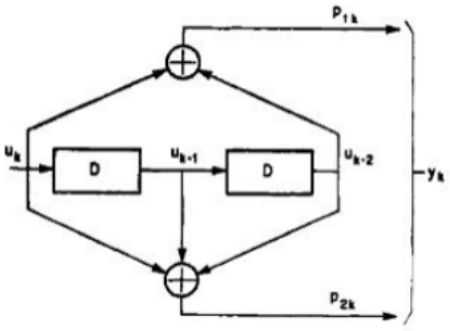
```
signal NS, CS: state_type;

signal w: integer range 0 to 1000;
signal offset: integer range 0 to 1000;
signal i: integer range 0 to 7;
signal output: std_logic_vector (1 downto 0);

signal parola1: std_logic_vector (7 downto 0);
signal parola2: std_logic_vector (7 downto 0);


signal o1: std_logic_vector (1 downto 0);
signal o1_bis: std_logic_vector (1 downto 0);
signal o2: std_logic_vector (1 downto 0);
signal o2_bis: std_logic_vector (1 downto 0);
signal o3: std_logic_vector (1 downto 0);
signal o3_bis: std_logic_vector (1 downto 0);
signal o4: std_logic_vector (1 downto 0);
signal o4_bis: std_logic_vector (1 downto 0);

signal ONcodificatore: std_logic;
signal nuovo_start: std_logic;
signal init_codificatore: std_logic;
```

- NS: stato prossimo del mio FSM
- CS: stato corrente del mio FSM
- w: numero di parole che vanno codificare
- offset: segnale utilizzato per mantenere stato del numero di parole lette e per ricavare l'indirizzo di memoria dove scrivere
- i: segnale per accedere al bit i-esimo del segnale i_data
- output: output ritornato dall'FSM codificatore
- parola1: è la prima delle due parole di memoria risultate dalla codifica della parola appena letta
- parola2: è la seconda delle due parole di memoria risultate dalla codifica della parola appena letta
- o1: bit 7 e bit 6 della parola 1
- o1_bis: bit 7 e bit 6 della parola 2
- o2: bit 5 e bit 4 della parola 1
- o2_bis: bit 5 e bit 4 della parola 2
- o3: bit 3 e bit 2 della parola 1
- o3_bis: bit 3 e bit 2 della parola 2
- o4: bit 1 e bit 0 della parola 1
- o4_bis: bit 1 e bit 0 della parola 2
- ONcodificatore: segnale utilizzato per attivare e disattivare l'avanzamento del codificatore che altrimenti continuerebbe ad avanzare di stato, anche in momenti in cui l'FSM principale sta svolgendo operazioni di lettura o scrittura in memoria, riutilizzando input già utilizzati e producendo quindi nuovi output errati e portandosi in stati non corretti.

- nuovo_start: Segnale utilizzato per indicare che la computazione è ripartita da capo, è appena stato letto w e si procederà con la codifica del primo bit della prima parola di memoria letta, perciò l'FSM codificatore dovrà essere resettato.
- init_codificatore: segnale utilizzato per resettare l'FSM codificatore, viene posto alto nel momento in cui è stata letta la prima parola di memoria e deve avere inizio la sua codifica.

Modello FSM del codificatore:

Il codificatore si occupa di produrre in output il codice di convoluzione ½, la funzione che svolge è quindi la seguente: Dato un ingresso u_k (di 1 bit) produce un'uscita y_k (di 2 bit)



Questo codificatore può essere rappresentato come una macchina a stati di Mealy dove:

Il bit d'ingresso è l'i-esimo bit del segnale i_data.

Gli input del componente saranno quindi data, i, clk, rst, start

Gli STATI sono codificati nel seguente modo:

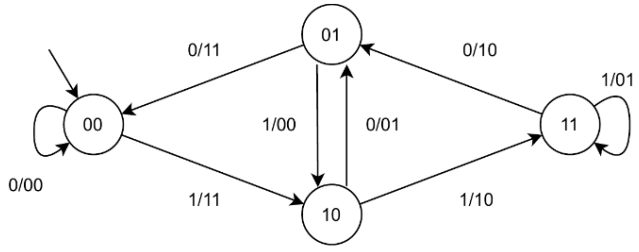
S0 → 00

S1 → 01

S2 → 10

S3 → 11

E l'uscita dipende dallo stato corrente e dall'input che riceviamo



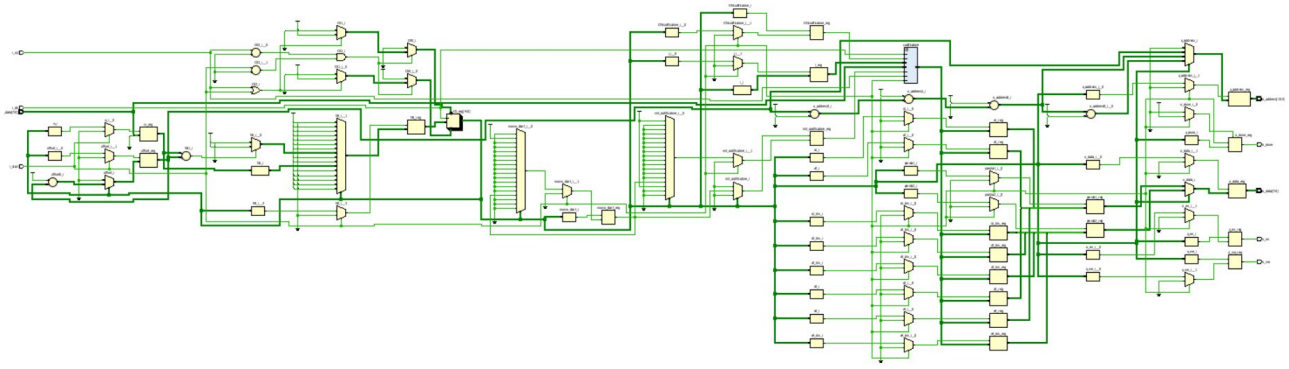
SEGNALI:

```
entity FSM_codificatore is
port(
        init : in std_logic;
        i: in integer range 0 to 7;
        clk: in std_logic;
        rst: in std_logic;
        i_data: in std_logic_vector(7 downto 0);
        start: in std_logic;
        enable: in std_logic;
        output: out std_logic_vector (1 downto 0));
end FSM_codificatore;
```

- init: segnale utilizzato per inizializzare il codificatore allo stato S0 nel momento in cui il modulo principale è in stato di STOP in attesa di un segnale di i_start.
- i: serve per accedere al bit presente all' i-esimo indice del segnale i_data che rappresenterà l'input dell'FSM codificatore.
- clk: clock dato dal testbench.
- rst: reset dato dal testbench.
- i_data: dato letto da memoria da parte dell'FSM principale.
- start: start dato dal testbench.
- enable: segnale che serve per attivare o disattivare l'avanzamento del codificatore che altrimenti ad ogni ciclo di clock cambierebbe stato e fornirebbe output.
- output: segnale di 2 bit che rappresenta la codifica del bit ricevuto in ingresso i_data(i).

I segnali dell'FSM codificatore sono mappati con i segnali dell'FSM principale come segue:

```
codificatore: FSM_codificatore
    port map(
        init => init_codificatore,
        i => i,
        rst => i_rst,
        clk => i_clk,
        i_data => i_data,
        start => i_start,
        enable => ONcodificatore,
        output => output);
```



Testing:

TB_EXAMPLE: test sul corretto funzionamento della logica di codifica

All'indirizzo di memoria 0 è presente il numero di parole da leggere w, pari in questo caso a 2. Ci si aspetta quindi che dopo la lettura di w, il modulo legga 2 parole di memoria e le codifichi correttamene salvandole in memoria.

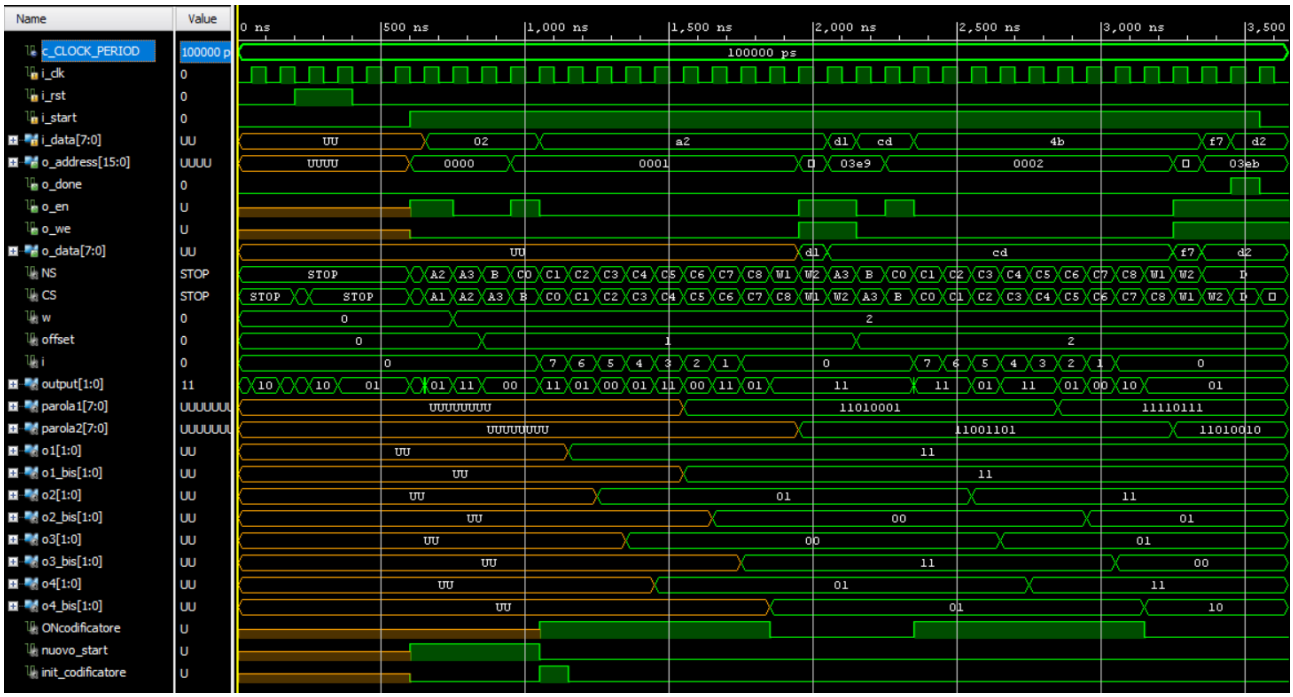
Lette da memoria, a partire dall'indirizzo 1, le due parole: 162, 75 ci si aspetta quindi scritte in memoria, a partire dall'indirizzo 1000, 4 parole: 209,205,247,201.

Viene dato:

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned(  2  , 8)),
                         1 => std_logic_vector(to_unsigned(  162  , 8)),
                         2 => std_logic_vector(to_unsigned(  75  , 8)),
                         others => (others =>'0'));
```

Ci si aspetta:

```
assert RAM(1000) = std_logic_vector(to_unsigned( 209 , 8)) report "TEST FALLITO (ENCODE)."
assert RAM(1001) = std_logic_vector(to_unsigned( 205 , 8)) report "TEST FALLITO (ENCODE)."
assert RAM(1002) = std_logic_vector(to_unsigned( 247 , 8)) report "TEST FALLITO (ENCODE)."
assert RAM(1003) = std_logic_vector(to_unsigned( 210 , 8)) report "TEST FALLITO (ENCODE)."
assert RAM(1004) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO (POST_ENCODE)."

assert false report "Simulation Ended! TEST PASSATO (ENCODE_EXAMPLE)" severity failure;
```



TB_SEQUENZA_MASSIMA:

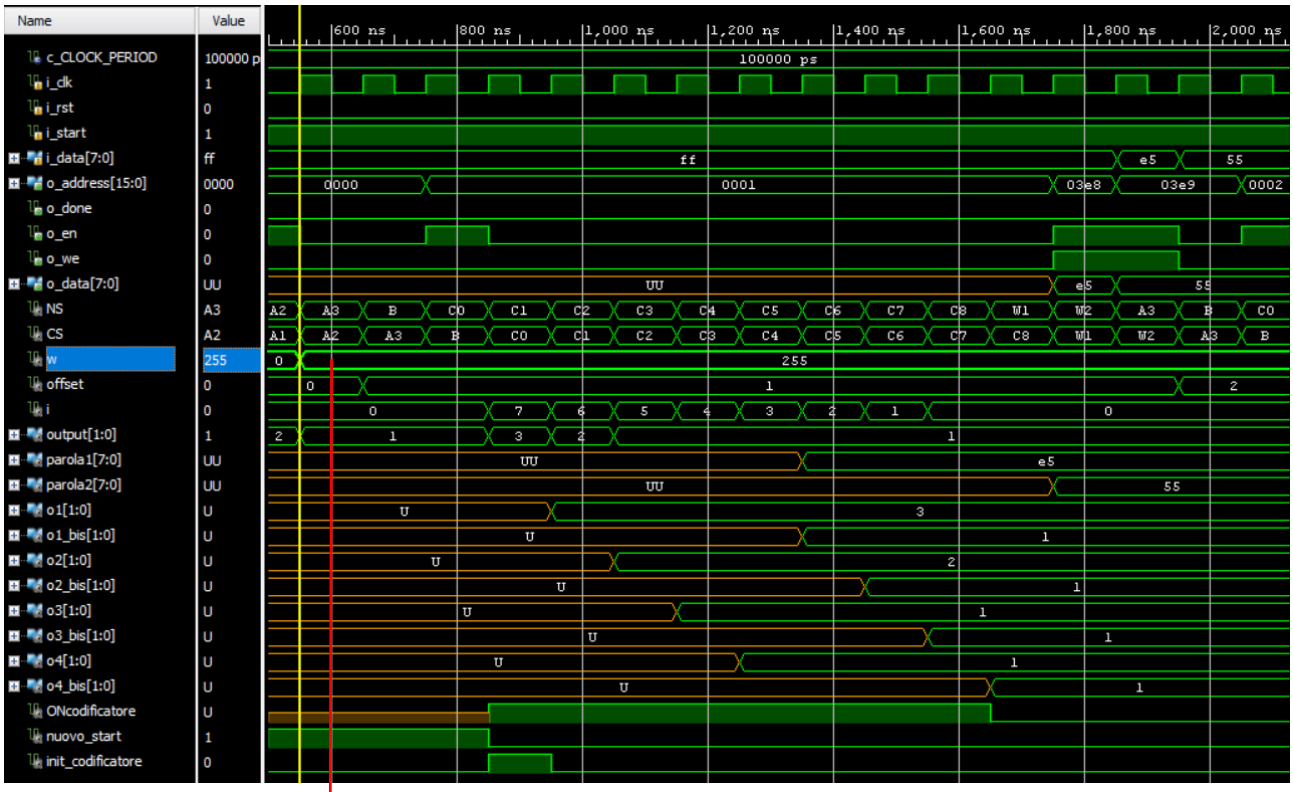
Lo scopo di questo testbench è verificare che la codifica funzioni con w = 255 parole.

Dato che w è di 8 bit il valora massimo sarà 255 cioè 1111 1111.

Viene dato:

W = 255

```
signal RAM: ram_type := (
                          0 => std_logic_vector(to_unsigned( 255,8)),
                          1 => std_logic_vector(to_unsigned( 255,8)),
                          2 => std_logic_vector(to_unsigned( 222,8)),
                          3 => std_logic_vector(to_unsigned( 145,8)),
                          4 => std_logic_vector(to_unsigned( 250,8)),
                          5 => std_logic_vector(to_unsigned( 140,8)),
                          6 => std_logic_vector(to_unsigned( 74,8)),
```



W = 255

TB_SEQUENZA_MINIMA:

Con questo testbench si vuole verificare che il progetto sia in grado di gestire il caso in cui il numero di parole da leggere da memoria e codificare sia 0.

In questo caso la computazione deve terminare subito dopo aver letto w e senza leggere nessuna parola da memoria, grazie al controllo del valore di w.

Viene dato:

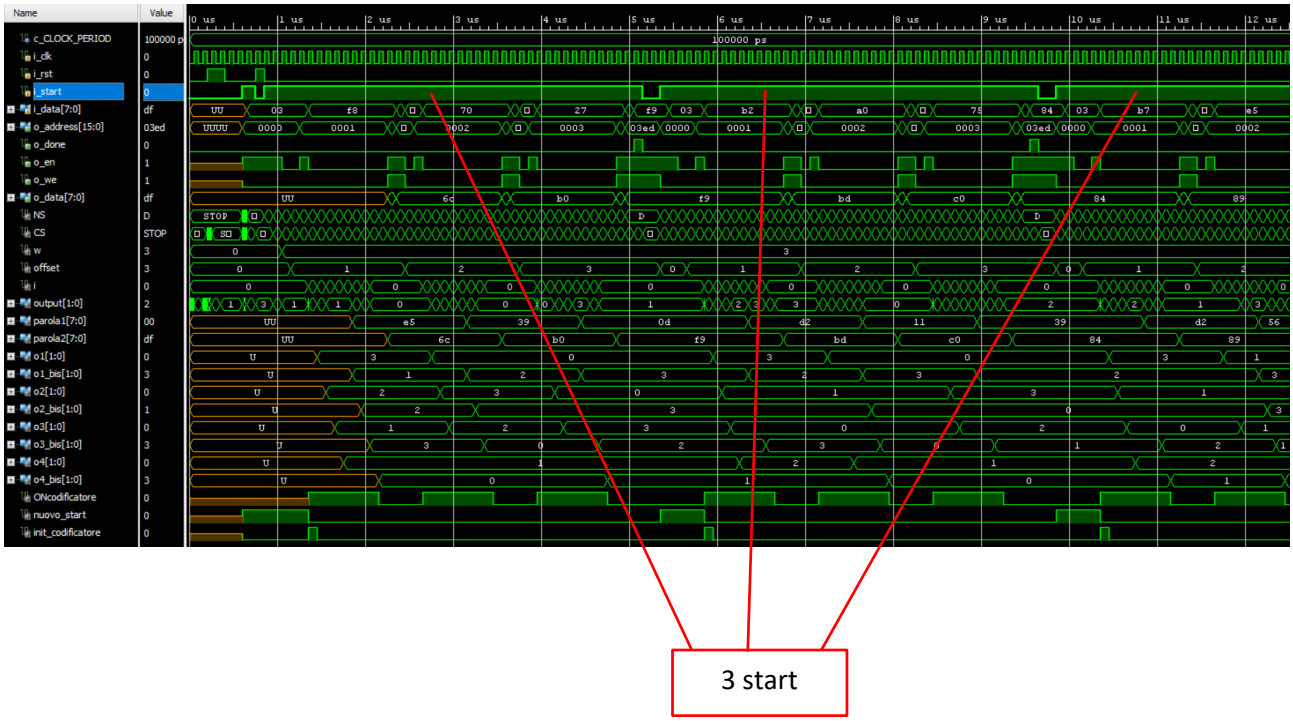
```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned(  0  , 8)),
                         1 => std_logic_vector(to_unsigned(  107  , 8)),
                         2 => std_logic_vector(to_unsigned(  84  , 8)),
                         3 => std_logic_vector(to_unsigned(  22  , 8)),
                         4 => std_logic_vector(to_unsigned(  59  , 8)),
                         5 => std_logic_vector(to_unsigned(  213  , 8)),
                         others => (others =>'0'));
```

Ci si aspetta:

```
assert RAM(1000) = STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)) report "TEST FALLITO" severity failure;
assert RAM(1001) = STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)) report "TEST FALLITO" severity failure;
assert RAM(1002) = STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)) report "TEST FALLITO" severity failure;
assert RAM(1003) = STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)) report "TEST FALLITO" severity failure;
assert RAM(1004) = STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)) report "TEST FALLITO" severity failure;
assert RAM(1005) = STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)) report "TEST FALLITO" severity failure;
assert RAM(1006) = STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)) report "TEST FALLITO" severity failure;
assert RAM(1007) = STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)) report "TEST FALLITO" severity failure;
assert RAM(1008) = STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)) report "TEST FALLITO" severity failure;
assert RAM(1009) = STD_LOGIC_VECTOR(TO_UNSIGNED(0, 8)) report "TEST FALLITO" severity failure;

assert false report "Simulation Ended! TEST PASSATO" severity failure;
```

TB_MULTI_START:

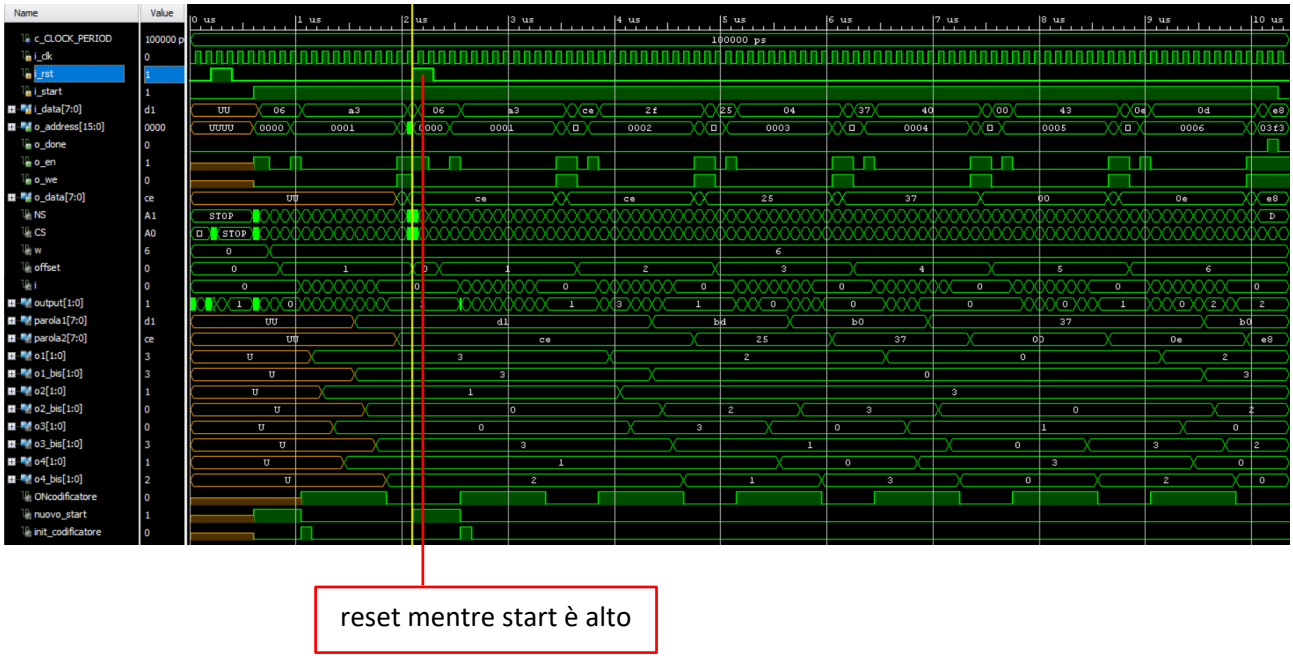
In questo caso start viene alzato e abbassato più volte, ciò significa che bisogna resettare e preparare il modulo del progetto ad una nuova codifica in modo che sia pronto per ricodificare tutto a partire dalla prima parola.



TB_RESET:

Questo testbench testa la capacità del progetto di gestire un segnale di reset nel mezzo della codifica, mentre il segnale di start continua a restare alto.

Infatti il modulo del progetto deve essere in grado di accettare un segnale di reset durante la computazione delle parole, il quale, una volta portato a livello alto, ovvero 1, porta il modulo a resettarsi e riprendere da capo la lettura e codifica delle parole senza provocare errori o malfunzionamenti



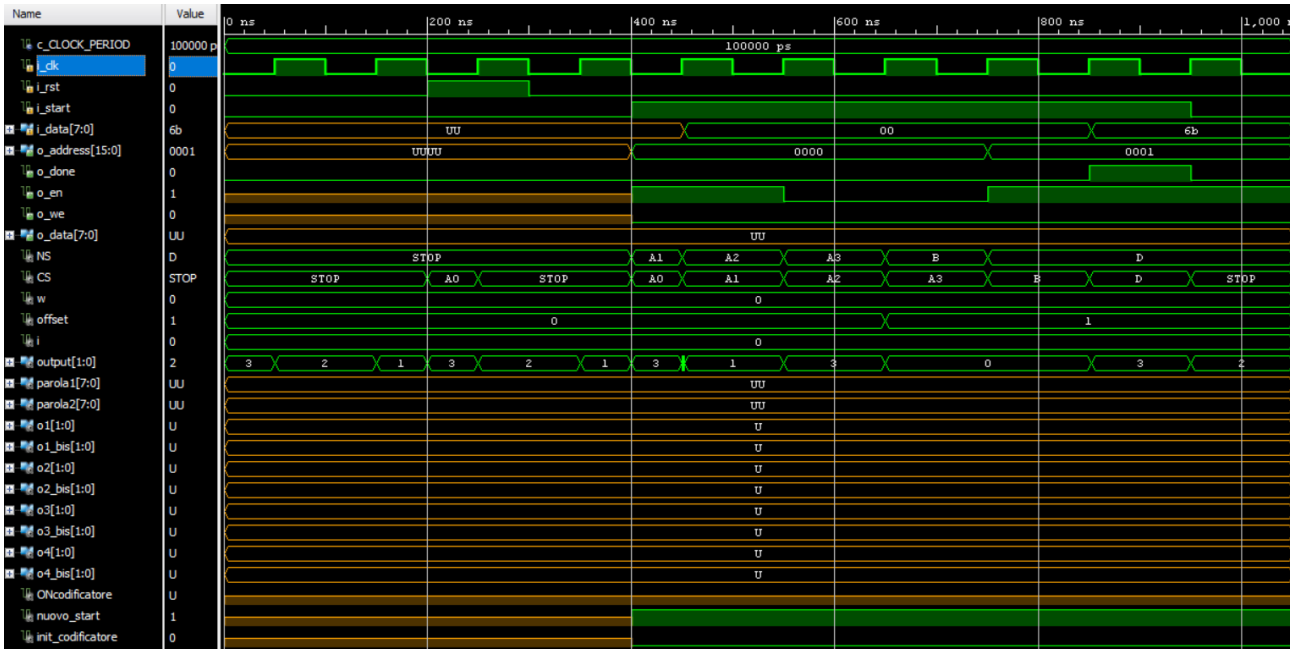
Conclusioni:

Per il progetto è stato usato l'Artix-7 FPGA xc7a200fbg484-1

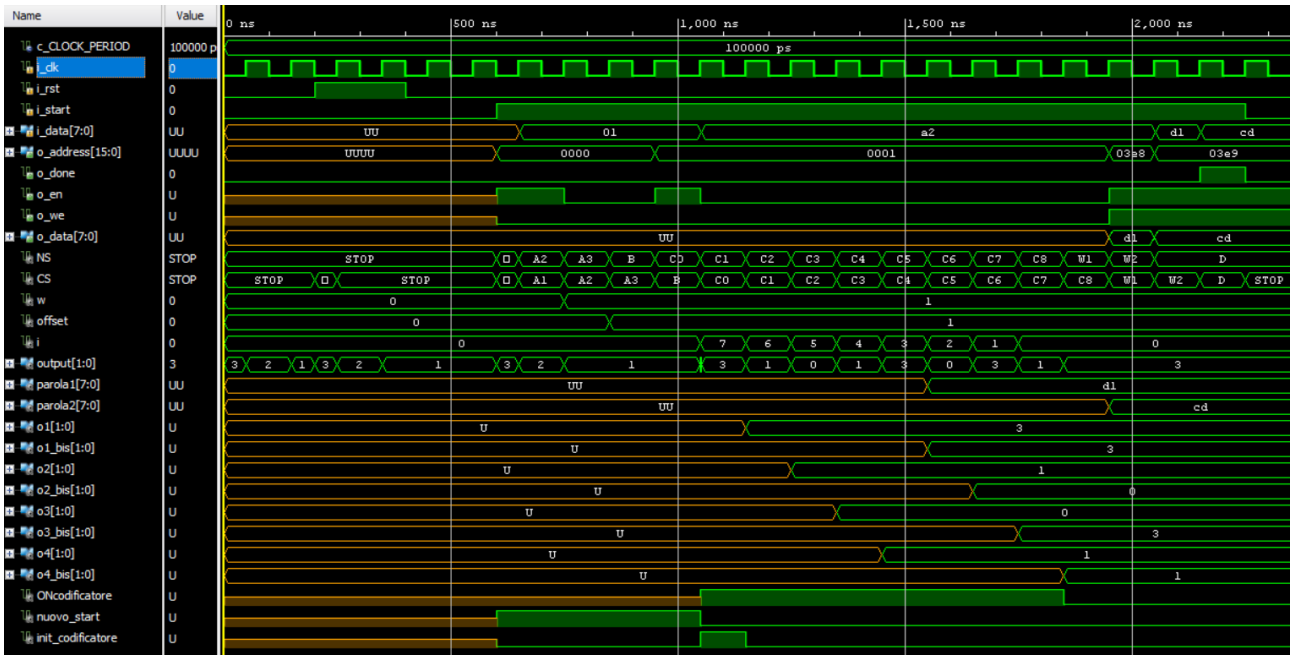
Il modulo progettato è risultato correttamente simulabile in:

- Behavioral Simulation

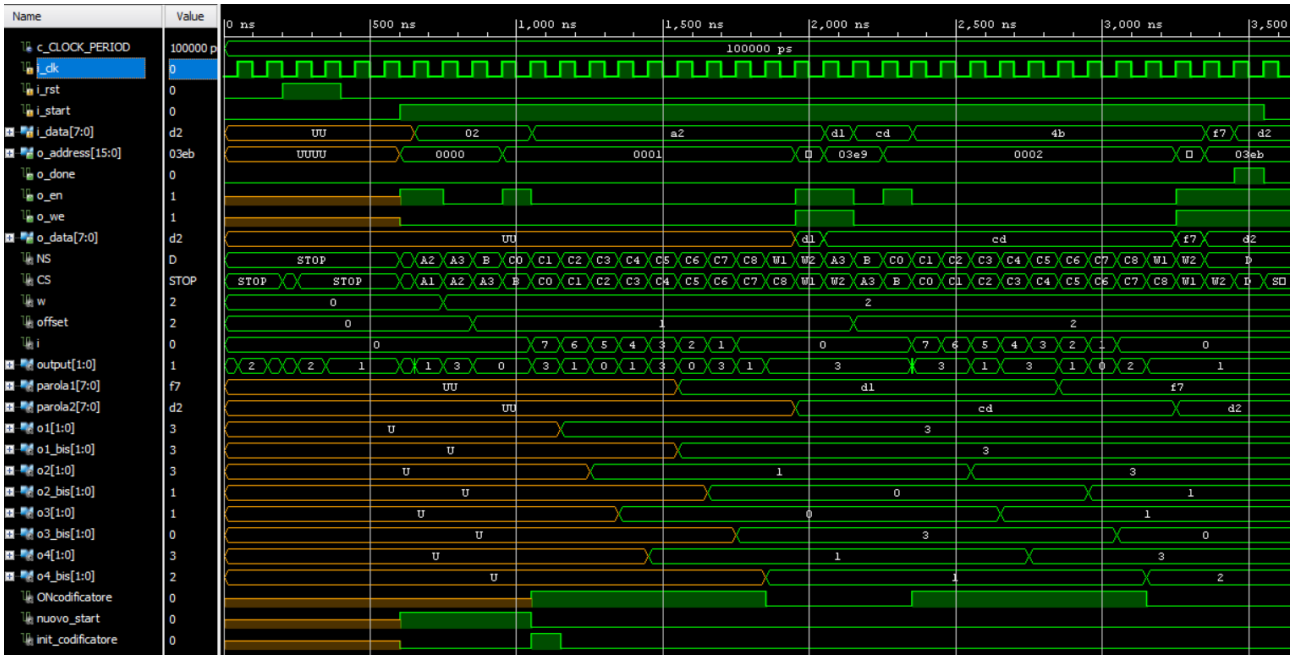
Con 0 parole ci vogliono 10 cicli di Clock per terminare la computazione



Con 1 parola ci vogliono 23 cicli di Clock per effettuare la codifica



Con 2 parole ci vogliono 36 cicli di Clock per effettuare la codifica



Con 3 parole ci vogliono 49 cicli di Clock per effettuare la codifica