# Prova Finale
# Progetto di Reti Logiche

A.A 2021-2022

Alen Kaja (Codice persona 10696919 - Matricola 936862)
Elis Kina (Codice persona 10636830 - Matricola 896263)

Aprile, 2022

Politecnico di Milano

# Indice

# 1 Introduzione

Il seguente documento descrive in dettaglio l'implementazione della prova finale di Reti Logiche dell'anno accademico 2021-2022, che abbiamo svolto in un gruppo di due studenti.

## 1.1 Specifica del progetto

L'obiettivo del progetto è implementare un modulo HW che realizzi un **Codificatore Convoluzionale con tasso di trasmissione $\frac{1}{2}$**, che, per ogni bit di informazione in ingresso, ne produca due di codice in uscita. Il modulo riceve i bit in ingresso in una sequenza continua di W parole, ciascuna da 8 bit, e restituisce in uscita una sequenza di Z parole, ognuna da 8 bit.
Nello specifico, l'ingresso viene serializzato nel flusso U da 1 bit, sul quale viene applicata la codifica convoluzionale. Si genera in uscita un flusso Y ottenuto concatenando i due bit prodotti dall'algoritmo di codifica e le Z parole si ottengono dalla parallelizzazione su 8 bit del flusso Y.
Secondo la notazione della figura seguente, all'istante k i bit in ingresso sono indicati con uk, uk-1 e uk-2, mentre le uscite, ottenute come XOR di bit, sono rispettivamente pk1 e pk2 e andranno a formare il flusso yk concatenandosi. Per ogni byte in input, si avranno perciò due byte in output.
Si riporta la struttura esemplificativa del modulo tramite uno schema e una FSM:

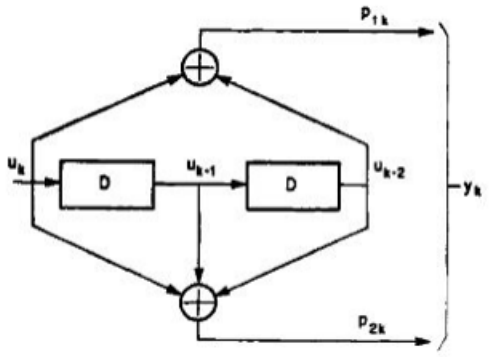


Figura 1: Schema del convolutore: al centro due FF tipo D, che fanno da buffer per l'input

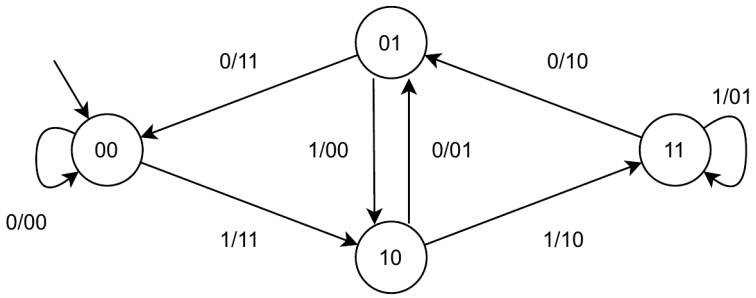


Figura 2: Diagramma degli stati: transizioni annotate come uk/p1k,p2k, stato iniziale in 00; è una Macchina di Mealy (valori di uscita determinati dallo stato attuale e dall'ingresso corrente)

Questa particolare codifica si può considerare con memoria visto che la codifica di un blocco di bit in ingresso influenza la codifica dei blocchi successivi.

Il modulo si interfaccia con una memoria per leggere e scrivere informazione.
Ad alto livello al componente viene richiesto di:

1. Accedere al primo indirizzo (indirizzo 0) della memoria per leggere i byte dell'immagine. La dimensione massima della codifica è 255 byte.
2. Leggere ogni parola in ingresso salvata in memoria.
3. Codificare ogni bit serializzato secondo la codifica convoluzionale $\frac{1}{2}$.
4. Scrivere i byte prodotti come risultato in memoria, e ripetere la procedura dal punto 2 per tutte le parole da codificare.

Il convolutore è una macchina sincrona con un clock globale che deve avere un periodo di almeno 100ns. Inoltre, l'implementazione deve essere in grado di gestire un segnale di Reset, che si è scelto di supporre asincrono rispetto al segnale di clock. In questo modo una nuova codifica può avvenire dal punto 1, dopo averne interrotta una in corso.

## 1.2 Input, Output e Memoria

I dati di input e di output saranno letti e scritti su una memoria di tipo RAM con indirizzamento al byte e bus con indirizzi di 16 bit. All' indirizzo 0 sarà fornito in binario il numero di parole da codificare w, seguiranno poi le parole. Dall'indirizzo 1000 in poi saranno scritte le parole ottenute come concatenamento dei bit prodotti dalla codifica.
Si riporta uno schema rappresentativo della memoria:

| **INDIRIZZO** | **VALORE** |
|---|---|
| 0 | numero di parole da codificare (w) |
| 1 | prima parola da codificare |
| 2 | seconda parola da codificare |
| ... | ... |
| w | ultima parola da codificare |
| ... | ... |
| 1000 | primo byte codificato |
| 1001 | secondo byte codificato |
| ... | ... |
| 1000 + w*2 - 1 | ultimo byte codificato |

Tabella 1: Struttura RAM

### 1.3 Interfaccia del componente

Il componente avrà 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;
```

Dove:

- i_clk sarà il segnale di clock in ingresso al componente;
- i_rst sarà il segnale di reset utilizzato per inizializzare la macchina;
- i_start sarà il senglae utilizzato per avviare la computazione;
- i_data sarà il vettore dei dati letti dalla memoria;
- o_address sarà il vettore con cui il componente; comunicherà l'indirizzo per leggere o scrivere la memoria;
- o_done sarà il segnale con cui il componente notificherà il termine della computazione;
- o_en sarà il segnale di Memory Enable che il componente porrà ad 1 per comunicare con la memoria;
- o_we sarà il segnale di Write Enable che verrà posto ad 1 per utilizzare la memoria in scrittura e a 0 per utilizzarla in lettura;
- o_data sarà il vettore contenente i dati da scrivere in memoria.

## 1.4 Specifiche di funzionamento

All'avvio, prima dell'inizio della prima computazione, il componente dovrà essere resettato e inizierà la computazione quando il segnale di start sarà posto a 1. Si assume come prerequisito che il contenuto della memoria non verrà modificato nel mentre e che il segnale di start resterà a 1 durante tutto il processo.
Una volta terminato il processo di codifica il segnale di done sarà posto ad 1 ed il componente aspetterà che il segnale di start torni a 0 prima di riportarsi nello stato iniziale da cui si potrà avviare un nuovo processo di codifica, senza dover necessariamente resettare il componente.

## 1.5 Esempio

La seguente sequenza di numeri mostra un esempio del contenuto della memoria al termine di una elaborazione. I valori che qui sono rappresentati in decimale, sono memorizzati in memoria con l'equivalente codifica binaria su 8 bit senza segno.

`Esempio1`: (Sequenza lunghezza 2)

W: 10100010 01001011
Z: 11010001 11001101 11110111 11010010

| INDIRIZZO | VALORE | COMMENTO |
|---|---|---|
| 0 | 2 | byte lunghezza sequenza di ingresso |
| 1 | 162 | primo Byte sequenza da codificare |
| 2 | 75 | |
| ... | ... | |
| 1000 | 209 | primo Byte sequenza di uscita |
| 1001 | 205 | |
| 1002 | 247 | |
| 1003 | 210 | |

Tabella 2: Contenuto RAM dell'esempio

# 2 Architettura

L'architettura da noi progettata è basata su una Macchina a Stati Finiti (FSM) che abbiamo deciso di modellare utilizzando un unico processo, il quale ha come parametro della lista di sensitività il segnale di clock, in modo che ogni variazione di quest'ultimo risvegli il processo stesso.

## 2.1 Descrizione ad Alto Livello

La macchina a stati finiti rappresenta il top-level component che gestisce tutto il processo di codifica e la sincronizzazione dei vari segnali in ingresso e in uscita al nostro modulo.
Da un'ottica di alto livello, l'implementazione esegue i seguenti passi:

1. Legge il primo indirizzo (0) e salva il numero delle parole da codificare;
2. Legge dal secondo indirizzo (1) in poi le parole da codificare;
3. Per ogni parola letta, si serializzano i bit e si entra nella sottomacchina[1] che esegue l'algoritmo convoluzionale;
4. Ogni stato della sottomacchina porta ad uno stato di controllo, che verifica se si sono prodotti 8 bit per scriverli in uscita:
   (a) Se sono stati prodotti 8 bit a partire dai primi 4 della parola letta, salva questa codifica in RAM e torna a codificare gli altri 4 bit dell'ingresso;
   (b) Se si sono scanditi gli ultimi 4 bit della parola in ingresso, scrive il byte prodotto in uscita e controlla il numero di parole codificate.
5. Se non ci sono più parole da leggere, la codifica è terminata e si sposta nello stato di terminazione, altrimenti si torna al punto (2);
6. Si attende il segnale di start per iniziare una nuova codifica, se si avverte si riparte dal punto (1);
7. Se si rileva un segnale di reset in qualunque momento, si ritorna al punto (1) per iniziare una nuova codifica.

Lo spostamento tra i registri della memoria avviene attraverso due segnali, uno per la lettura ed uno per la scrittura, che si descriveranno in seguito (sezione 2.3).

[1] Con il termine sottomacchina si intende la finite-state machine dedotta dal diagramma degli stati(vedi `Figura 2`), che rappresenta il vero e proprio algoritmo di codifica convoluzionale di 1 bit in 2 di codice.

## 2.2 Macchina a Stati Finiti

In particolare, l'architettura dell'implementazione è stata realizzata con specifica *Behavioural*. La macchina a stati finiti è costituita da undici stati principali, alcuni fra i quali hanno dei "sottostati". Questo significa che anche se gli stati concettuali (da un punto di vista algoritmico) sono undici, la realizzazione mono process, dipendente dal clock, richiede in alcuni casi più stati.

Segue una descrizione degli stati formali della FSM:

- `RESET`: Stato di idle in cui si posiziona la FSM al reset della computazione. Qui i vari segnali vengono inizializzati.
- `READY`: La macchina attende il segnale `i_start` per iniziare una nuova codifica.
- `NUM_W`: Stato in cui viene letta la cella zero della RAM, che contiene il numero di parole da codificare. Se non ci sono parole da codificare rimanda a `DONE`.
- `READ_IN`: Stato in cui si leggono le parole dalla ram e si salvano per processarle.
- `ENTER_FSM`: Stato preparatorio per entrare nella FSM di codifica.
- `FSM_STATES`: Stato che comprende altri quattro sottostati, `S0`, `S1`, `S2`, `S3` rappresentanti il diagramma degli stati dell'algoritmo di codifica di un bit in ingresso in due di uscita.
- `CHECK`: Stato che comprende due sottostati, `CHECK` e `ITERATOR_UPDATE` che servono rispettivamente a controllare se possiamo scrivere in memoria e ad aggiornare la posizione di salvataggio di ogni bit nel vettore ottenuto dalla codifica. Ad ogni byte prodotto rimanda a `SAVE_VECTOR`.
- `SAVE_VECTOR`: Stato che indirizza agli stati di scrittura del primo byte di uscita o del secondo valutando un flag che indica l'avvenuta codifica dei primi 4 bit o dei secondi 4 su 8 del byte in ingresso.
- `FIRST_O_WRITE`: Stato che scrive in RAM il byte che codifica i primi 4 bit della parola in ingresso. Poi riporta a `READ_IN` per continuare la lettura.
- `SECOND_O_WRITE`: Stato che scrive in RAM il byte che codifica gli ultimi 4 bit della parola in ingresso. Se si è codificata tutta la sequenza rimanda a `DONE`, altrimenti torna a `READ_IN`.
- `DONE`: Stato in cui si aspetta che `i_start` venga riazzerato, settando `o_done` a 1 e riportandosi allo stato di `RESET`.

Nella figura a seguito è riportato il disegno dell'automa generale.
Per comodità si è rinominato lo stato `FIRST_O_WRITE` con `WRITE_1` e `SECOND_O_WRITE` con `WRITE_2`. Inoltre si è omessa la rappresentazione di un'arco che riporti a `RESET` da qualsiasi stato, avendo precedentemente supposto il segnale asincrono.

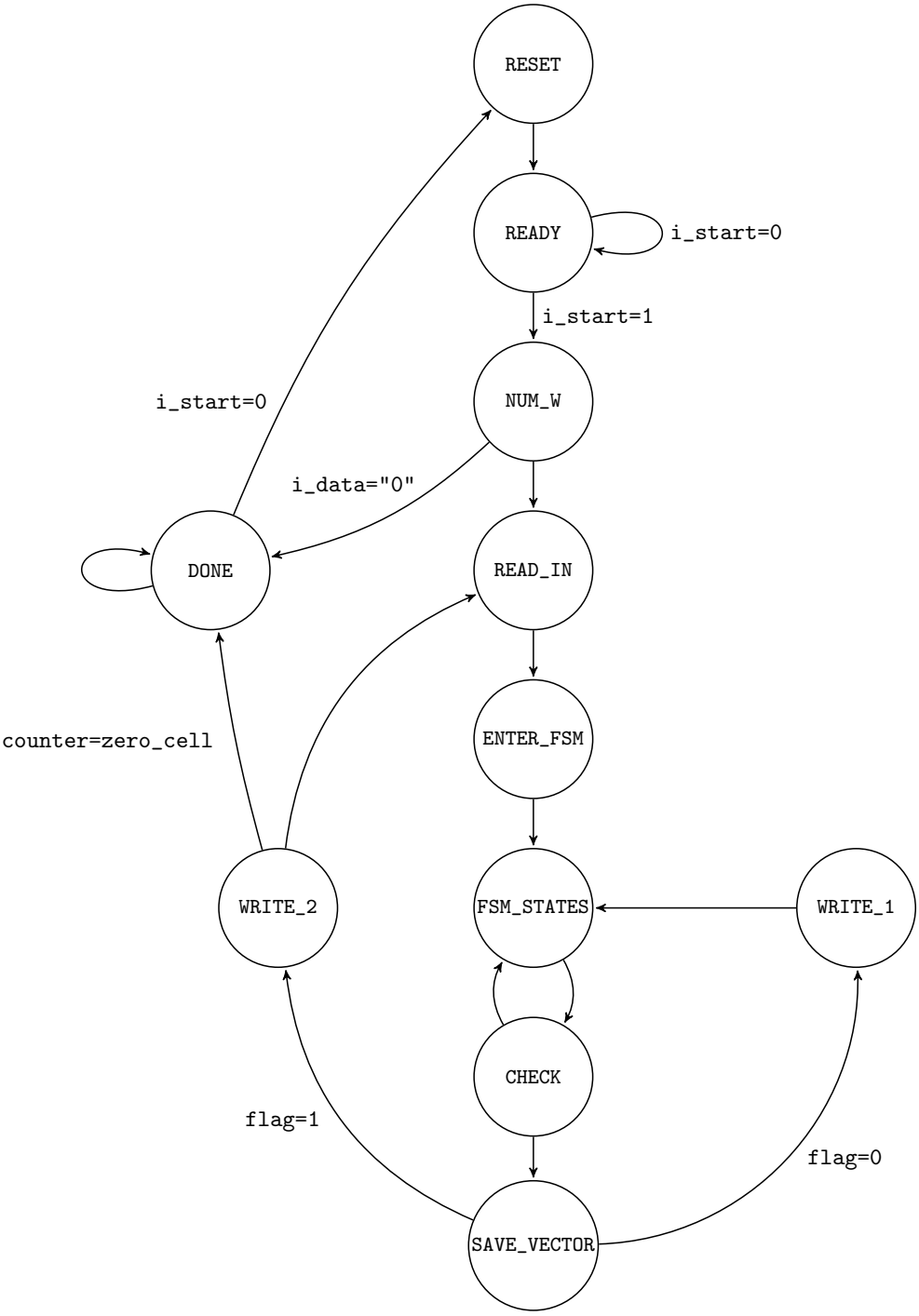


Figura 3: Diagramma degli stati della Macchina a Stati Finiti

## 2.3 Segnali utilizzati

I segnali da noi utilizzati nel codice sono i seguenti:

```
signal curr_s, last_state : state;
signal counter : std_logic_vector(15 downto 0);
signal zero_cell : std_logic_vector(7 downto 0);
signal input : std_logic_vector(7 downto 0);
signal output : std_logic_vector(7 downto 0);
signal i: std_logic_vector( 2 downto 0);
signal tmp: std_logic_vector( 2 downto 0) := (others =>'0');
signal flag: std_logic := '0';
signal mem_write: std_logic_vector(15 downto 0) := (others =>'0');
```

Figura 4: Segnali utilizzati

- *curr_s, last_state*: servono a spostarsi tra gli stati della macchina; *last_state* salva il riferimento all'ultimo stato attraversato nella FSM di codifica;
- *counter*: tiene conto del numero di parole codificate, poiché si aggiorna ad ogni nuova lettura di parole dalla memoria. Se assume lo stesso valore di *zero_cell* la computazione termina;
- *zero_cell*: vettore che memorizza la grandezza della sequenza da codificare;
- *input*: vettore che memorizza le parole in ingresso per processarle;
- *output*: vettore che scriverà in *o_data* i byte codificati;
- *i*: segnale che serve a iterare sui bit del vettore *input*;
- *tmp*: segnale che serve a iterare sui bit del vettore *output*;
- *flag*: segnale che distingue tra la scrittura del primo byte (avvenuta codifica dei primi 4 bit ingresso) o del secondo byte (avvenuta codifica dei secondi 4 bit ingresso) in uscita;
- *mem_write*: segnale ausiliario per assegnare ad *o_address* il riferimento all ram per scrivere.

# 3 Risultati Sperimentali

## 3.1 Sintesi

La sintesi e l'implementazione sono state fatte usando il software a disposizione Xilinx Vivado Webpack e la FPGA target usata è stata la Artix-7, xc7a200tfbg484-1.
Per la realizzazione non sono stati forniti vincoli stringenti sull'area di utilizzo e sul tempo di esecuzione, dunque non si è badato ad ottimizzare molto questi componenti.
Inoltre la strategia scelta nella scrittura del codice è quella user-friendly e sopratutto facile da interpretare e/o modificare in futuro. Di seguito sono riportati i componenti della FPGA utilizzati.

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Slice LUTs* | 130 | 0 | 134600 | 0.10 |
| LUT as Logic | 130 | 0 | 134600 | 0.10 |
| LUT as Memory | 0 | 0 | 46200 | 0.00 |
| Slice Registers | 98 | 0 | 269200 | 0.04 |
| Register as Flip Flop | 98 | 0 | 269200 | 0.04 |
| Register as Latch | 0 | 0 | 269200 | 0.00 |
| F7 Muxes | 1 | 0 | 67300 | <0.01 |
| F8 Muxes | 0 | 0 | 33650 | 0.00 |

Figura 5: Report di sintesi

Per quanto riguarda il design timing summary, si sono rilevati un Worst Negative Slack di 94.906 ns, un Worst Hold Slack di 0.161 ns e un Worst Pulse Width Slack di 49.5 ns.
Complessivamente il risultato rientra nei limiti del funzionamento.

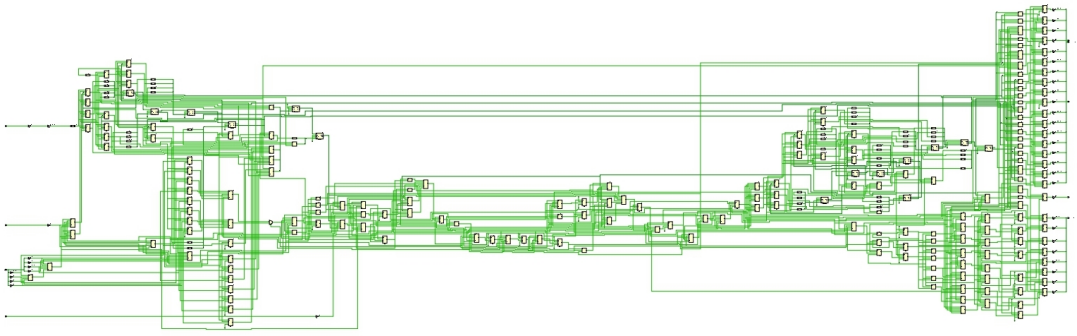

Figura 6: Schema dell'implementazione: 302 cells, 38 I/O ports, 398 nets

Altre caratteristiche, come l'on-chip power (potenza), sono comunque piccole in confronto con le risorse a disposizione, vista la ridotta complessità del progetto. Il totale on-chip power è 0.132 W, il 99% di questo è dinamico e il 75% è usato dall' I/O.

## 3.2 Simulazioni

Abbiamo verificato il corretto funzionamento del componente utilizzando sia i test benches forniti che altri da noi pensati per stressare particolari casi della computazione. Si è cercato di coprire tutti i possibili cammini che il componente potesse intraprendere durante il funzionamento: osservando la waveform dei vari segnali, ci siamo accertati che i segnali ricevuti e trasmessi avessero un andamento conforme alla specifica.

**Test Casuali**
Attraverso uno script in python, abbiamo generato numerosi test casuali e il componente ha risposto correttamente nella totalità dei casi, arrivando a superare anche migliaia di codifiche. Questi test comprendevano sequenze lunghe e randomiche di byte in RAM.

**Test Forniti**
La suite di test forniti era piuttosto completa, poiché prevedeva vari corner case. Di seguito è fornita una breve descrizione dei test utilizzati e per quelli più significativi viene anche mostrato l'effettivo corretto funzionamento grazie allo screenshot dell'andamento dei segnali durante la simulazione.
Si riportano alcuni test significativi:

- `Test Reset`: si tratta di un test con RESET asincrono (sequenza di lunghezza 6). Il segnale comporta il reinizializzazione della computazione;

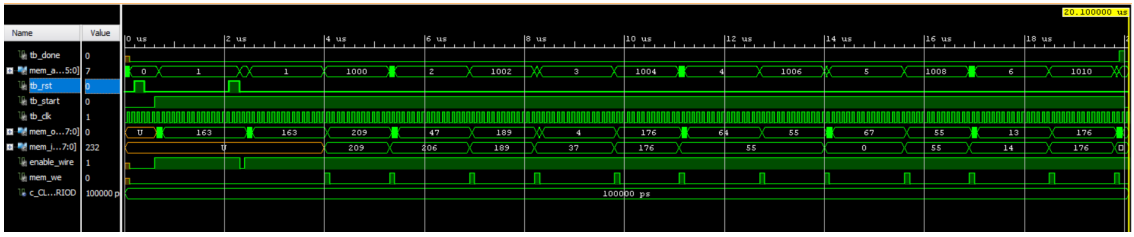


Figura 7: Simulazione con `i_rst` asincrono

- `Test Sequenza Minima`: test con sequenza di lunghezza nulla, cioè RAM(0) = "00000000". Non scrive in memoria e interrompe subito la codifica;

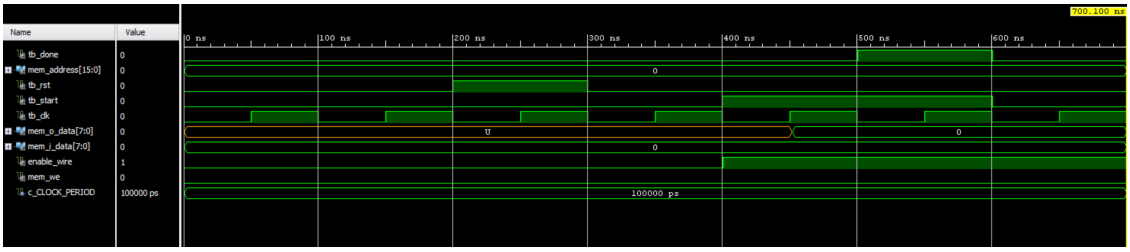


Figura 8: Simulazione con `zero_cell`="00000000"

- `Test Sequenza Massima`: test con sequenza di lunghezza massima, cioè RAM(0) = "11111111";

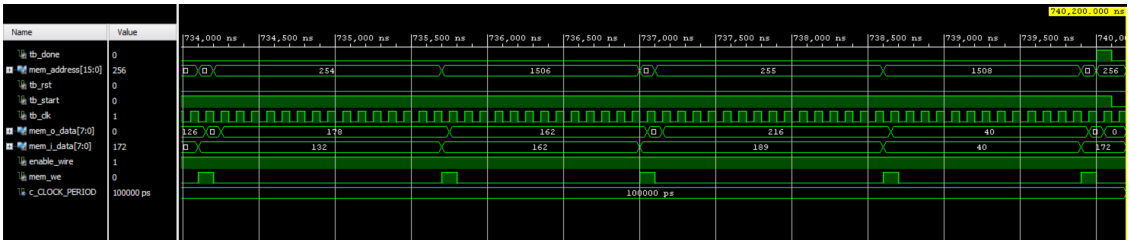


Figura 9: Parte finale della simulazione con `zero_cell`="11111111"

- `Test Re-Encode`: codifica di più flussi uno dopo l'altro (3 flussi). Il segnale `i_start` si porta alto appena dopo la terminazione di una codifica, permettendo la codifica di un altro flusso.

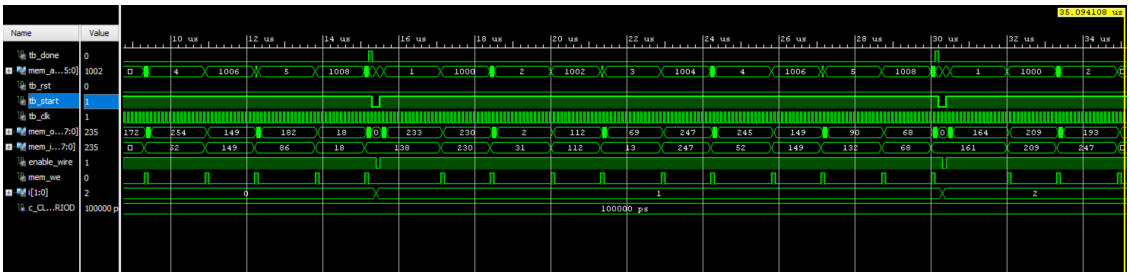


Figura 10: Estratto della simulazione

**Test Custom**
Abbiamo poi verificato alcuni casi particolari, come sequenze nulle all'interno di più flussi di codifica, oppure reset asincroni mandati più volte al componente.

Il testing ha evidenziato alcune criticità nel codice iniziale e ha guidato lo sviluppo della soluzione fino alla sua versione finale.
Per ogni test bench usato abbiamo testato il modulo sia in pre-sintesi (behavioural simulation) che in post-sintesi (functional simulation), ottenendo sempre esiti positivi.

# 4 Conclusioni

Si ritiene che l'architettura progettata rispetti anzitutto le specifiche, fatto che è stato verificato mediante estensivo testing sia casuale, che con test benches manualmente scritti. Oltre a ciò l'architettura è stata pensata sulla base di una FSM, evitando l'utilizzo di Latch che avrebbero potuto creare l'instaurarsi di cicli infiniti.
Dal punto di vista del design, si è scelto di utilizzare una unica FSM senza componenti esterni. Visto l'ordine di complessità del progetto abbiamo optato per una soluzione compatta.
Il componente, come previsto, non utilizza che una piccola percentuale dell'intera FPGA e il ritardo massimo dell'esecuzione rientra largamente all'interno del margine di clock richiesto.

## 4.1 Ottimizzazioni

Le ottimizzazioni principali sono state ridurre gli stati della macchina al numero finale. Nel caso in cui il numero di parole da codificare fosse zero si è deciso di riportare subito il componente nello stato di `DONE`, in attesa di una nuova codifica.
Un miglioramento possibile consiste nel suddividere la computazione in più process che abbiano dei compiti separati, ad esempio gestire dei segnali in particolare. Per questo progetto, infine, sono possibili ottimizzazioni che permettono di recuperare alcuni cicli di clock nell'elaborazione tra gli stati, il cui numero si potrebbe ulteriormente ridurre.