



POLITECNICO
MILANO 1863

PROGETTO FINALE DI RETI LOGICHE

Codificatore Convoluzionale

Giovanni Manfredi

Matricola: 937160
Codice Persona: 10708042

Sebastiano Meneghin

Matricola: 937058
Codice Persona: 10627650

supervisore
Prof. Fabio SALICE

15 giugno 2022

Indice

| | | |
|----------|--|-----------|
| 1 | Introduzione | 2 |
| 1.1 | Analisi delle specifiche | 2 |
| 1.2 | Esempio | 3 |
| 1.3 | Approccio progettuale | 5 |
| 2 | Architettura | 6 |
| 2.1 | Data path | 6 |
| 2.1.1 | Lettura da memoria e serializzazione | 7 |
| 2.1.2 | Convolutore | 7 |
| 2.1.3 | Deserializzazione e Scrittura in memoria | 8 |
| 2.1.4 | Contatore | 8 |
| 2.1.5 | Indirizzo di lettura | 9 |
| 2.1.6 | Indirizzo di scrittura | 9 |
| 2.2 | Finite State Machine - FSM | 10 |
| 2.2.1 | Stato di reset e inizializzazione | 10 |
| 2.2.2 | Lettura e scrittura prima parola (byte) | 11 |
| 2.2.3 | Lettura e scrittura parole successive alla prima | 11 |
| 2.2.4 | Terminazione | 12 |
| 3 | Risultati Sperimentali | 13 |
| 3.1 | Sintesi | 13 |
| 3.2 | Schematics | 14 |
| 3.3 | Simulazioni | 20 |
| 3.3.1 | Test bench : "tb_seq_min" | 20 |
| 3.3.2 | Test bench : "tb_reset" | 20 |
| 3.3.3 | Test bench : "tb_re_encode" | 20 |
| 3.3.4 | Test bench : "tb_doppio_uguale" | 20 |
| 3.3.5 | Test bench : "tb_seq_max " | 20 |
| 3.3.6 | Test bench : "tb_tre_reset " | 20 |
| 4 | Conclusioni | 21 |

1 Introduzione

1.1 Analisi delle specifiche

Lo scopo del progetto è l'implementazione di un componente hardware, descritto in VHDL, che interfacciandosi con la memoria esegua l'operazione seguente.

L'elemento implementato va a **leggere** parole da otto bit (o un byte) da una memoria RAM, per poi andarle a **serializzare** in un flusso continuo di un bit, che arriva in input ad un **convolutore** (secondo lo schema riportato in Figura 1). Questo codifica il suo ingresso in un flusso in uscita di due bit, che vengono conseguentemente **deserializzati** e **scritti** in memoria.

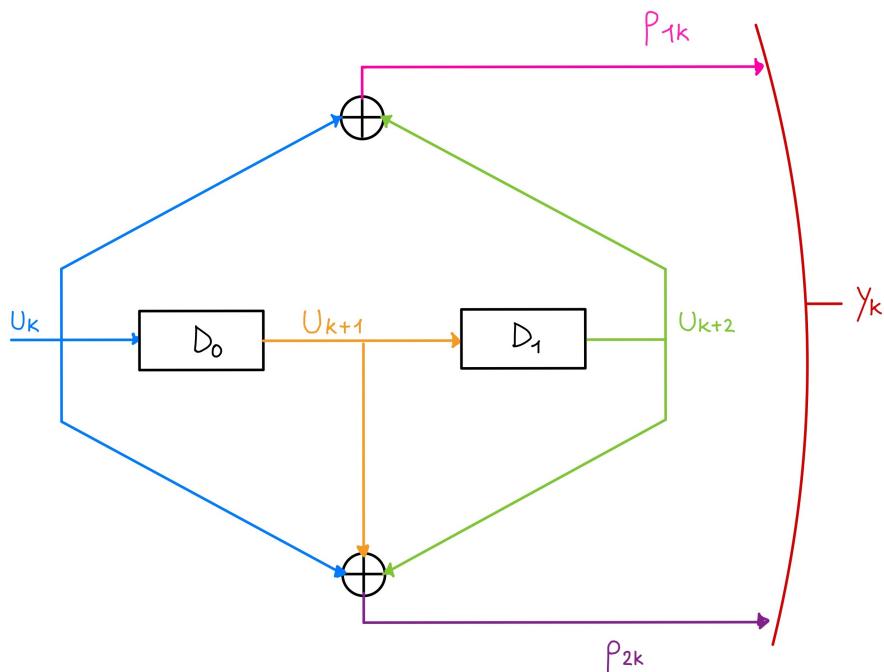


Figura 1: Data path del convolutore

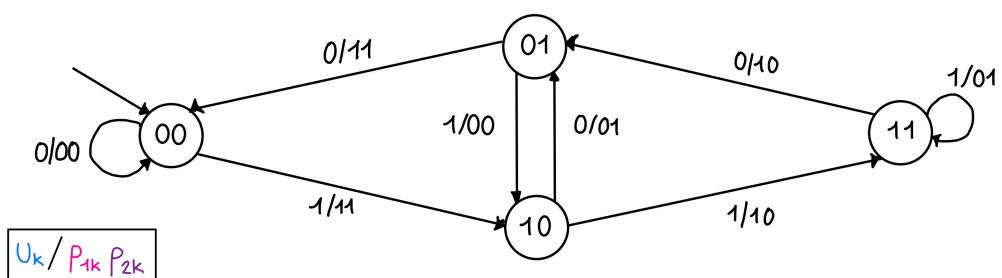


Figura 2: Finite State Machine (FSM) del convolutore

1.2 Esempio

Vediamo ora alcuni esempi riguardanti il funzionamento del componente hardware, mostrando i numeri letti e scritti in memoria

$W : 1010\ 1100\ 0101\ 1111$

$Z : 1101\ 0001\ 0010\ 1011\ 0011\ 0100\ 1001\ 0101$

| INDIRIZZO MEMORIA | VALORE | COMMENTO |
|-------------------|--------|--------------------------------------|
| 0 | 2 | Byte lunghezza sequenza di ingresso |
| 1 | 172 | Primo byte in sequenza da codificare |
| 1 | 95 | |
| ... | 0 | Memoria vuota |
| 1000 | 209 | Primo byte sequenza di uscita |
| 1001 | 43 | |
| 1002 | 52 | |
| 1003 | 149 | |

Tabella 1: Esempio 1 - Sequenza di lunghezza 2

$W : 1000\ 0011\ 0000\ 1100$
 0101 1101 0010 1001
 0011 1111 0000 0110
 $Z : 1101\ 1100\ 0000\ 1110\ 1011\ 0000\ 1110\ 1011$
 0011 0100 1001 1000 0111 1101 0001 1111
 0111 1110 0101 0101 1011 0000 0011 1010

| INDIRIZZO MEMORIA | VALORE | COMMENTO |
|-------------------|--------|--------------------------------------|
| 0 | 6 | Byte lunghezza sequenza di ingresso |
| 1 | 131 | Primo byte in sequenza da codificare |
| 1 | 12 | |
| 1 | 93 | |
| 1 | 41 | |
| 1 | 63 | |
| 1 | 6 | |
| ... | 0 | Memoria vuota |
| 1000 | 209 | Primo byte sequenza di uscita |
| 1001 | 14 | |
| 1002 | 176 | |
| 1003 | 235 | |
| 1004 | 52 | |
| 1005 | 152 | |
| 1006 | 125 | |
| 1007 | 31 | |
| 1008 | 126 | |
| 1009 | 85 | |
| 1010 | 176 | |
| 1011 | 58 | |

Tabella 2: Esempio 2 - Sequenza di lunghezza 6

1.3 Approccio progettuale

Ci siamo approcciati al progetto andando ad individuare quali fossero i singoli passaggi che il componente esegue per realizzare l'operazione richiesta. Abbiamo individuato cinque *steps* principali:

- Lettura da memoria
- Trasformazione parallela - seriale
- Convolutore
- Trasformazione seriale - parallela

Abbiamo inoltre notato la necessità di tenere d'occhio altri due elementi che se non gestiti bene potrebbero essere problematici: **il ciclo di clock** e i **buffer**.

Data la natura **sincrona** del processo, nella lettura da memoria dovremo "aspettare il valore" un ciclo di clock in più (rispetto ad un processo asincrono). Per quanto riguarda **il tempo di clock**, visto il funzionamento del convolatore (ricava due bit da uno) "*il tempo*" sarà scandito dalla parte di scrittura in memoria, in quanto non potremo leggere una nuova parola finché non avremo salvato entrambe le parole in output al convolatore.

Siamo quindi andati a delinare un **Data path** che descrivesse il circuito del componente che abbiamo poi controllato tramite dei segnali di una **Macchina a Stati Finiti o Finite State Machine - FSM**. Le immagini e le descrizioni dettagliate del circuito e macchina a stati sono nella Sezione 2.

2 Architettura

Come introdotto nell'introduzione (Sezione 1) andremo a vedere lo schema del **Data path** e a spiegarne nel dettaglio i suoi punti e la **FSM** che lo controlla.

2.1 Data path

Qui sotto riportato in Figura 3 l'intero Data path che andremo ad analizzare più nel dettaglio in questo capitolo. Specificati in figura ci sono anche i segnali di controllo gestiti dalla FSM e per ogni flusso è specificato il numero di bit.

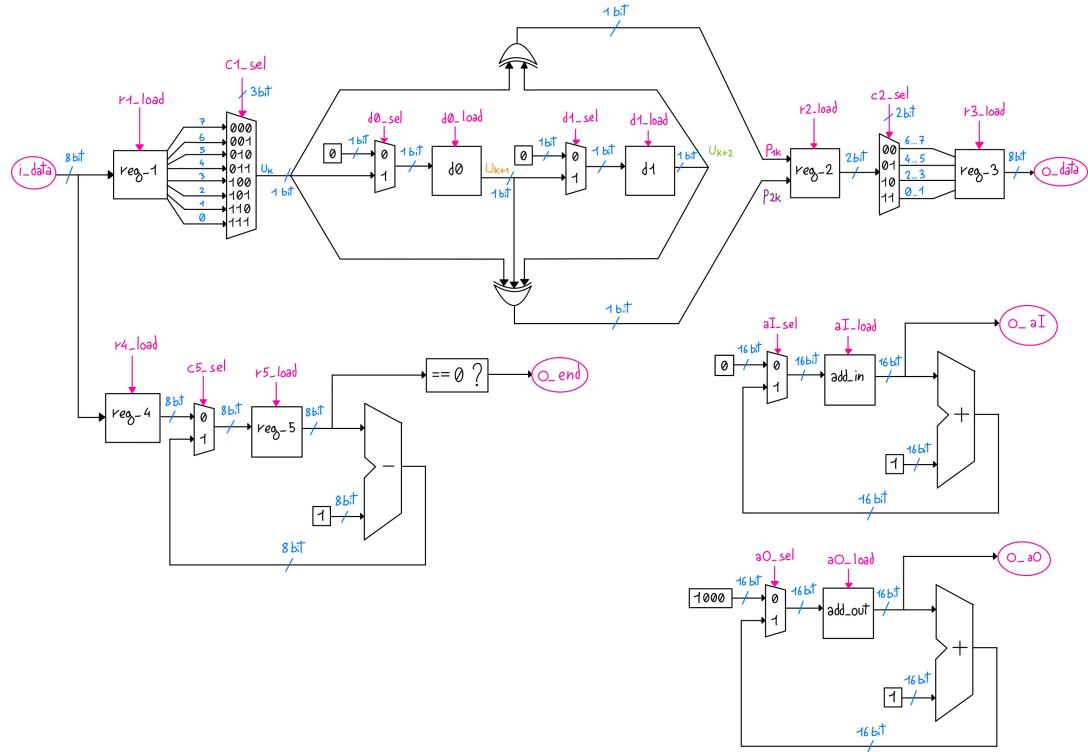


Figura 3: Data path

2.1.1 Lettura da memoria e serializzazione

La prima sezione è dedicata alla **lettura da memoria e serializzazione**. La lettura dalla memoria avviene in parole da 8 bit, che il processo salva sul registro reg_1 . Da questo registro, servendosi di un **multiplexer**, genera un flusso da 1 bit (u_k) (serializzazione).

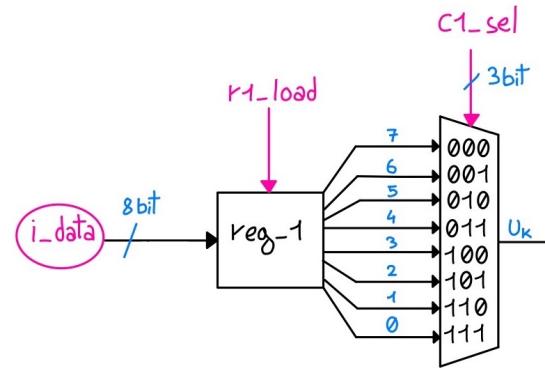


Figura 4: Lettura da memoria e serializzazione

2.1.2 Convolutore

Dal flusso u_k , seguendo lo schema indicato dalla specifica, si arriva al **convolutore**, che svolge l'operazione salvando il risultato (di 2 bit) nel registro reg_2 . Si notino i vari selettori utilizzati per resettare lo stato del convolutore.

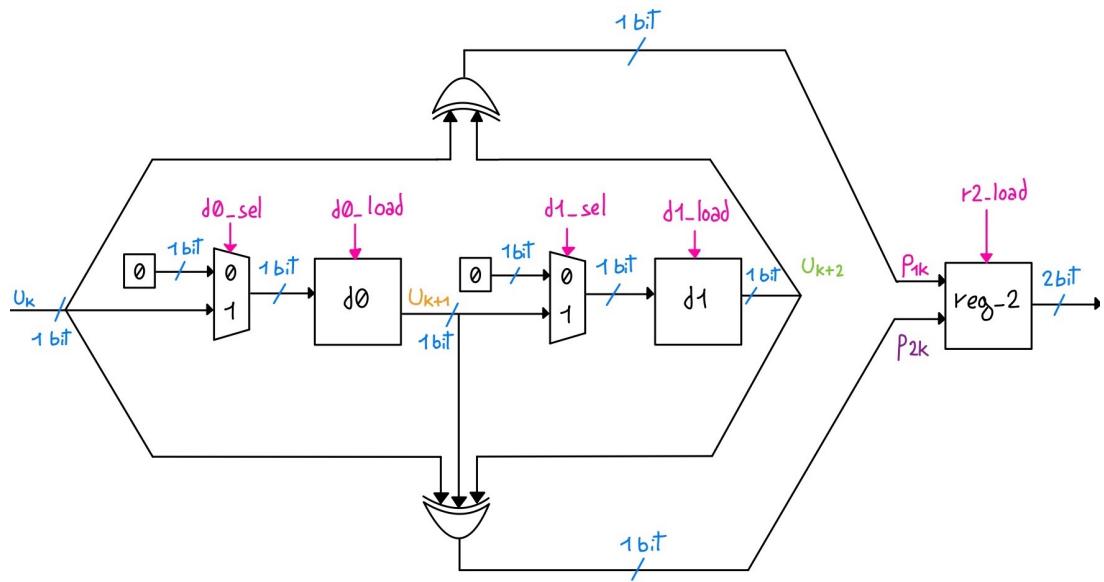


Figura 5: Convolutore operazionale

2.1.3 Deserializzazione e Scrittura in memoria

Dal registro reg_2 si genera un flusso di 2 bit che utilizzando un **decoder** (che deserializza il flusso) viene salvato nel registro reg_3 da 8 bit, che verrà poi **scritto in memoria**.

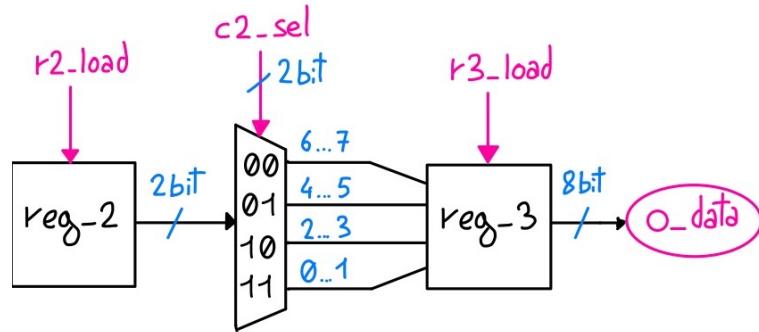


Figura 6: Deserializzazione e scrittura in memoria

2.1.4 Contatore

Parallelamente all'operazione principale (ovvero: lettura → serializzazione → convolutore → deserializzazione → scrittura) il processo **conta** il numero di volte che traduce una parola. Visto che nel primo indirizzo (l'indirizzo 0) è presente il numero di parole da leggere, una volta che il contatore sarà arrivato a zero il componente imposterà il **segnale di $o_{end} = 1$** . Si noti che bisogna prestare particolare attenzione nella FSM a quando decrementare il contatore.

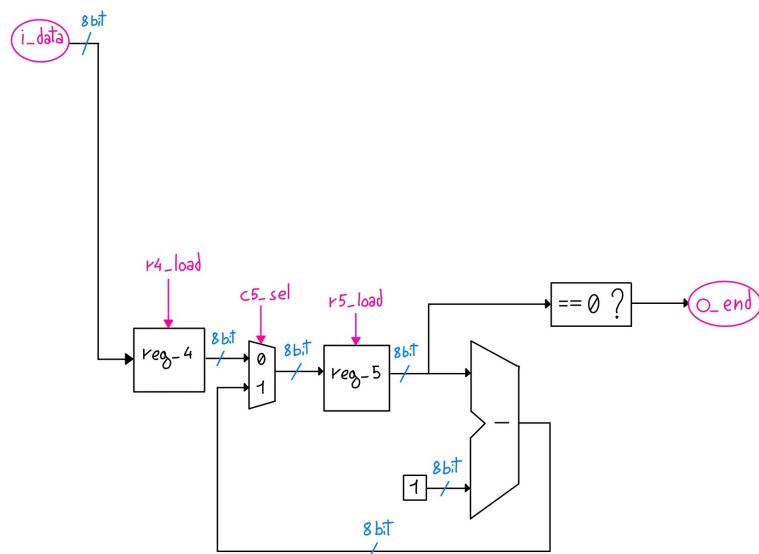


Figura 7: Contatore

2.1.5 Indirizzo di lettura

Per aggiornare correttamente l'indirizzo di lettura da memoria si è pensato all'uso di un modulo a sé stante. L'indirizzo di lettura da memoria è **inizializzato a 0** nel registro add_{in} e aumentato di 1 ogni volta che il processo deve leggere una nuova parola.

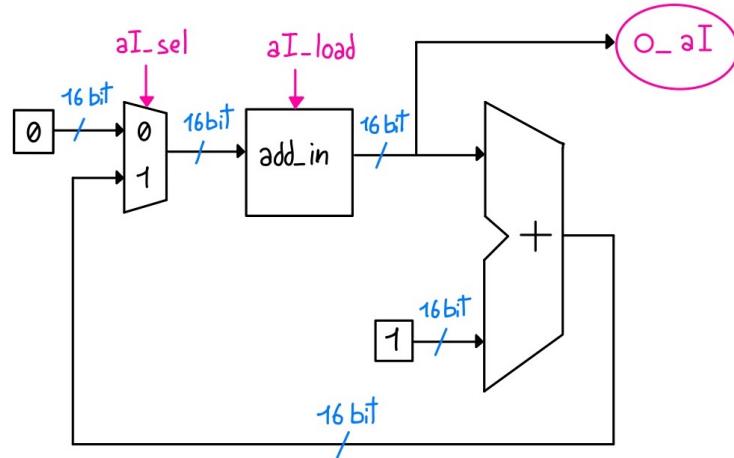


Figura 8: Indirizzo di lettura

2.1.6 Indirizzo di scrittura

Per aggiornare correttamente l'indirizzo di scrittura in memoria si è pensato all'uso di un modulo a sé stante. L'indirizzo di scrittura in memoria è **inizializzato a 1000** nel registro add_{out} e aumentato di 1 ogni volta che il processo deve scrivere una nuova parola.

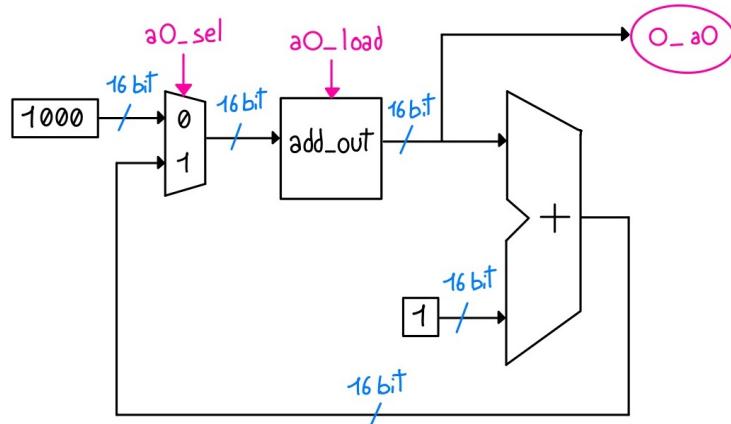
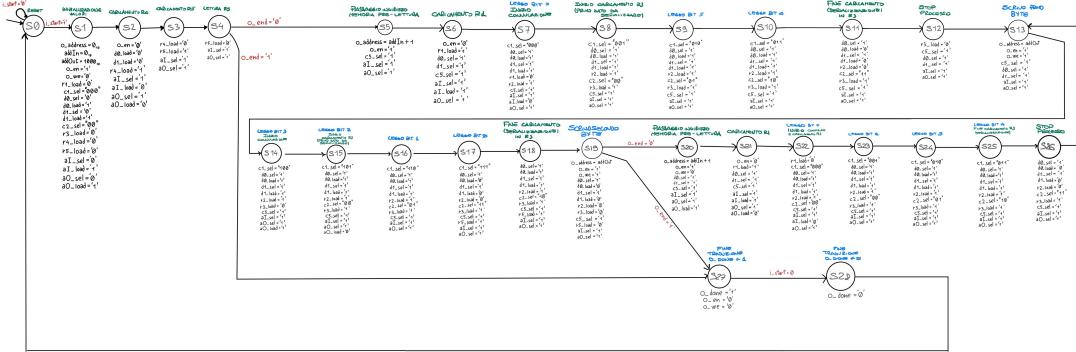


Figura 9: Indirizzo di scrittura

2.2 Finite State Machine - FSM

La macchina a stati (FSM) controlla il Data path visto nella sezione 2.1. Vedremo ora come funziona la sequenza di esecuzione del programma e quali sono le sezioni che lo compongono. Si noti che in tutti i cambi di stato (indicati con le frecce) avvengono sul fronte di salita del clock (i_{ck}).



2.2.2 Lettura e scrittura prima parola (byte)

Avendo verificato che il primo numero salvato in memoria (il numero di parole) sia diverso da zero, il processo esegue la fase di lettura, convoluzione e scrittura per almeno una parola.

La lettura del primo byte è diversa dalla iterazioni successive alla prima, in quanto si deve re-inizializzare il convolutore prima di usarlo. Come detto precedentemente, vista la natura **sincrona** della memoria, è necessario aspettare due stati per poter avere a disposizione il dato dalla memoria. Alla fine della scrittura di entrambe le parole in memoria (da ciascuna parola in memoria ne vengono generate due), viene decrementato il contatore e verificato se è stato azzerato.

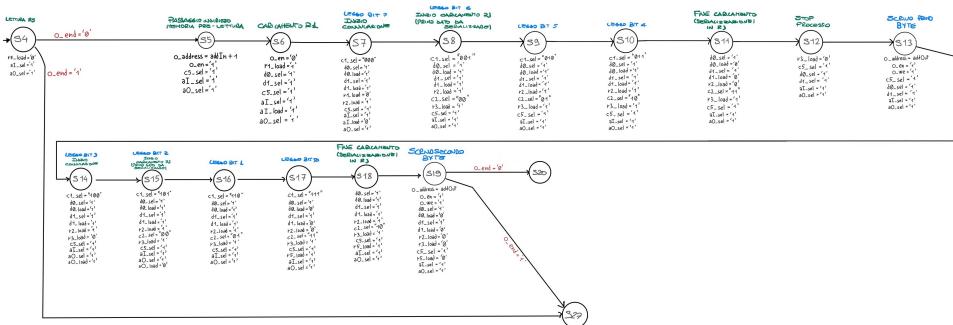


Figura 12: Lettura e scrittura della prima parola da memoria

2.2.3 Lettura e scrittura parole successive alla prima

Se il contatore di parole non si è azzerato, l'operazione continua, andando a leggere una nuova parola da memoria, seguendo lo stesso procedimento di prima, ma senza la preoccupazione di resettare il convolatore. Anche alla fine di questa sezione verifichiamo, decrementando il contatore, se esso è stato portato a 0. Si noti che viene riutilizzata la seconda parte della lettura e scrittura della Sezione 2.2.2.

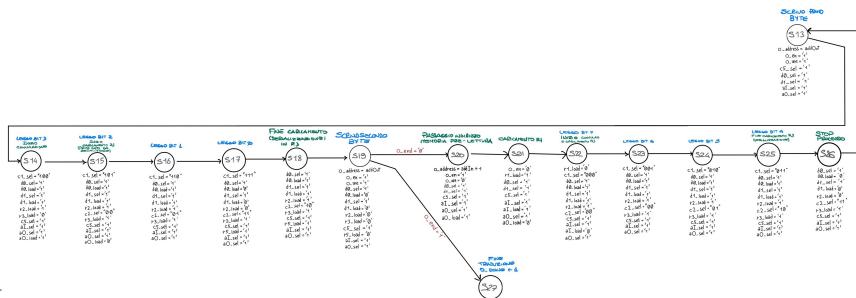


Figura 13: Lettura e scrittura delle parole successive alla prima da memoria

2.2.4 Terminazione

Una volta che dallo stato $S4$ oppure $S19$ viene sollevato il segnale o_{end} il processo va nello stato $S27$. Da questo stato il componente imposta il segnale $o_{done} = 1$ e attende che i_{start} sia abbassato per resettare $o_{done} = 0$ e quindi tornare allo stato di RESET ($S0$).

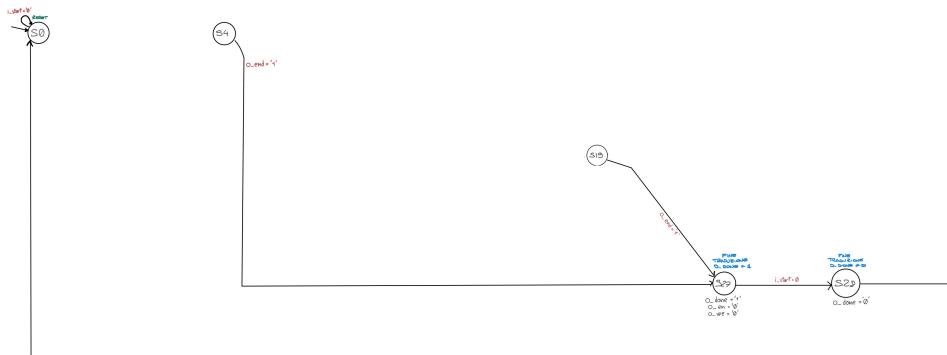


Figura 14: Terminazione

3 Risultati Sperimentali

3.1 Sintesi

Il progetto è stato realizzato e testato sulla versione di **Vivado 2018.3**, utilizzando come FPGA target **xc7a200tfgbg484-1**, e ha generato il seguente report di sintesi:

| Site Type | Used | Fixed | Available | Util% |
|-----------------------|------|-------|-----------|-------|
| Slice LUTs* | 94 | 0 | 134600 | 0.07 |
| LUT as Logic | 94 | 0 | 134600 | 0.07 |
| LUT as Memory | 0 | 0 | 46200 | 0.00 |
| Slice Registers | 73 | 0 | 269200 | 0.03 |
| Register as Flip Flop | 73 | 0 | 269200 | 0.03 |
| Register as Latch | 0 | 0 | 269200 | 0.00 |
| F7 Muxes | 1 | 0 | 67300 | <0.01 |
| F8 Muxes | 0 | 0 | 33650 | 0.00 |

Per il consumo invece:

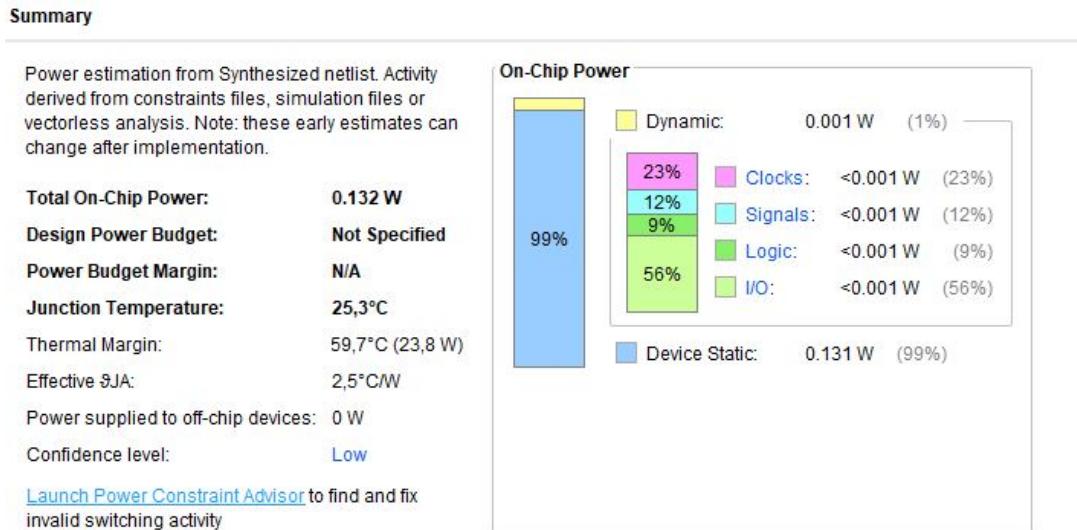


Figura 15: Consumi e potenza

3.2 Schematics

In seguito una serie di immagini che mostrano come i circuiti sono stati descritti internamente da Vivado.

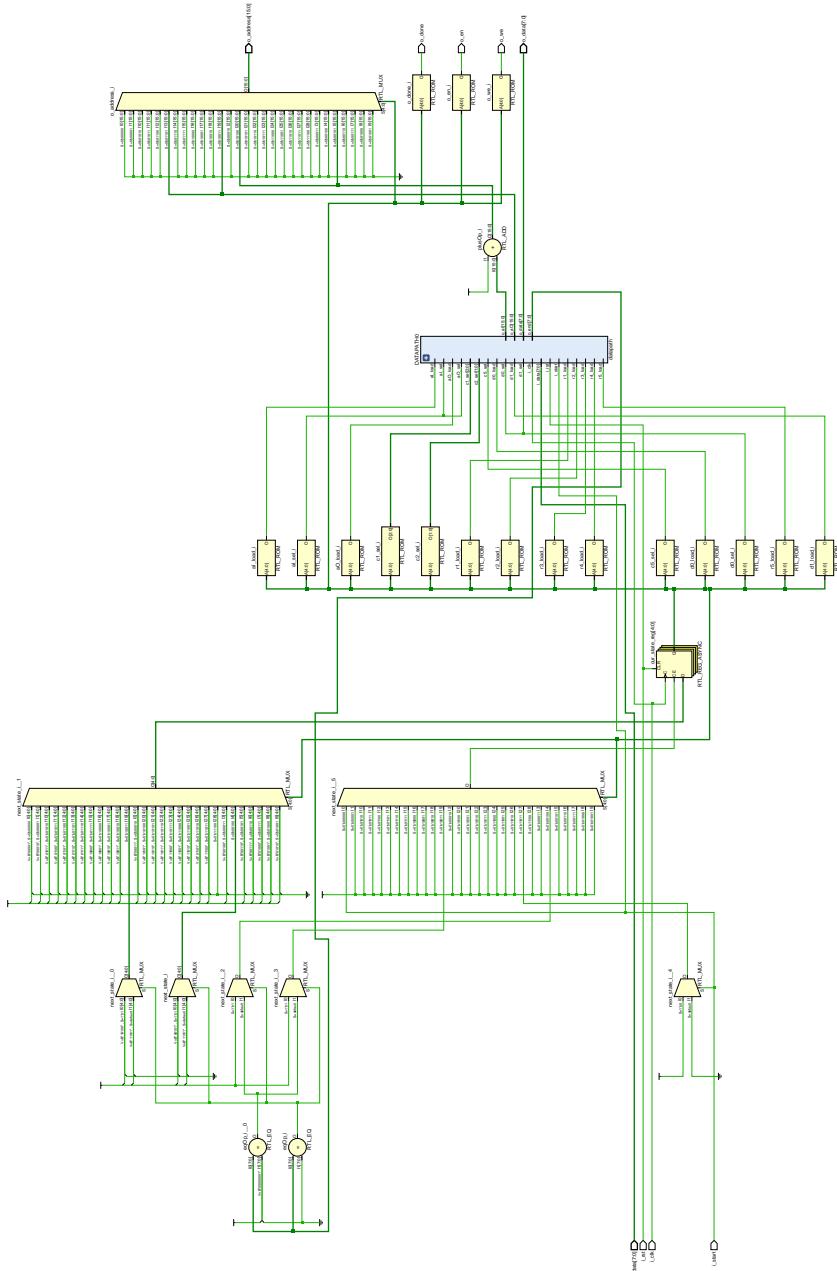


Figura 16: Circuito in pre-sintesi con datapath non esplicito

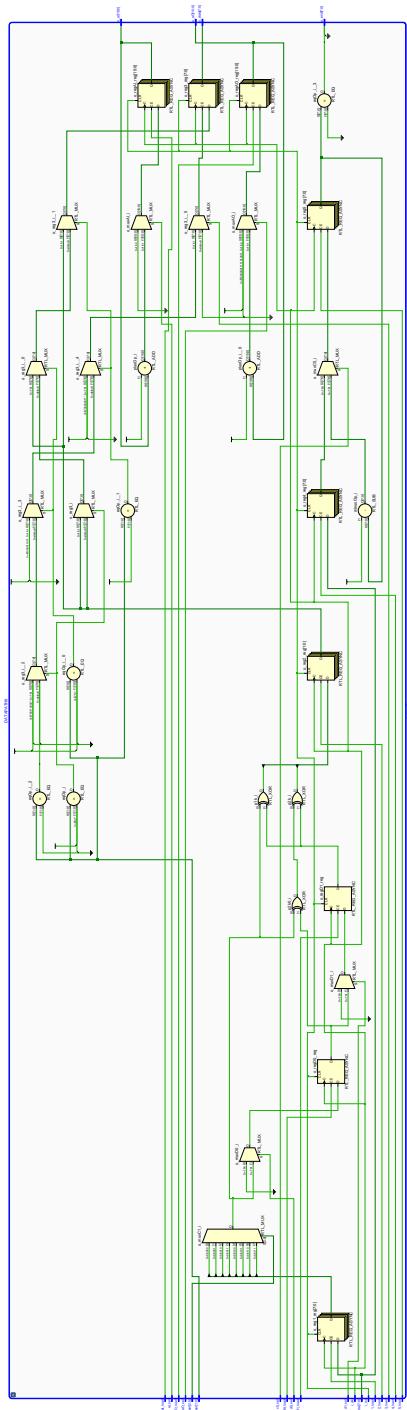


Figura 17: Circuito in pre-sintesi del datapath

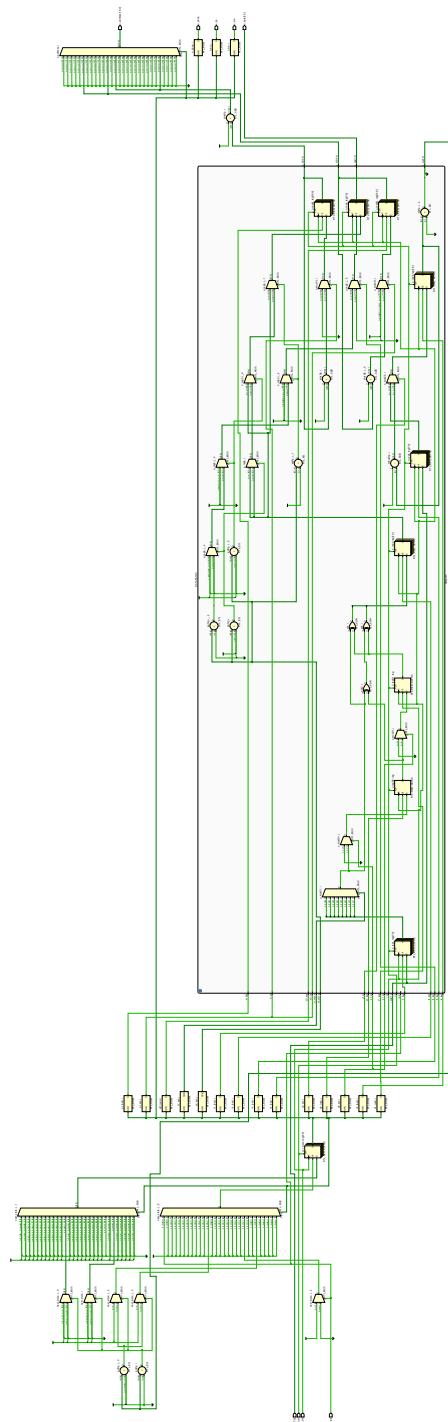


Figura 18: Circuito in pre-sintesi completo

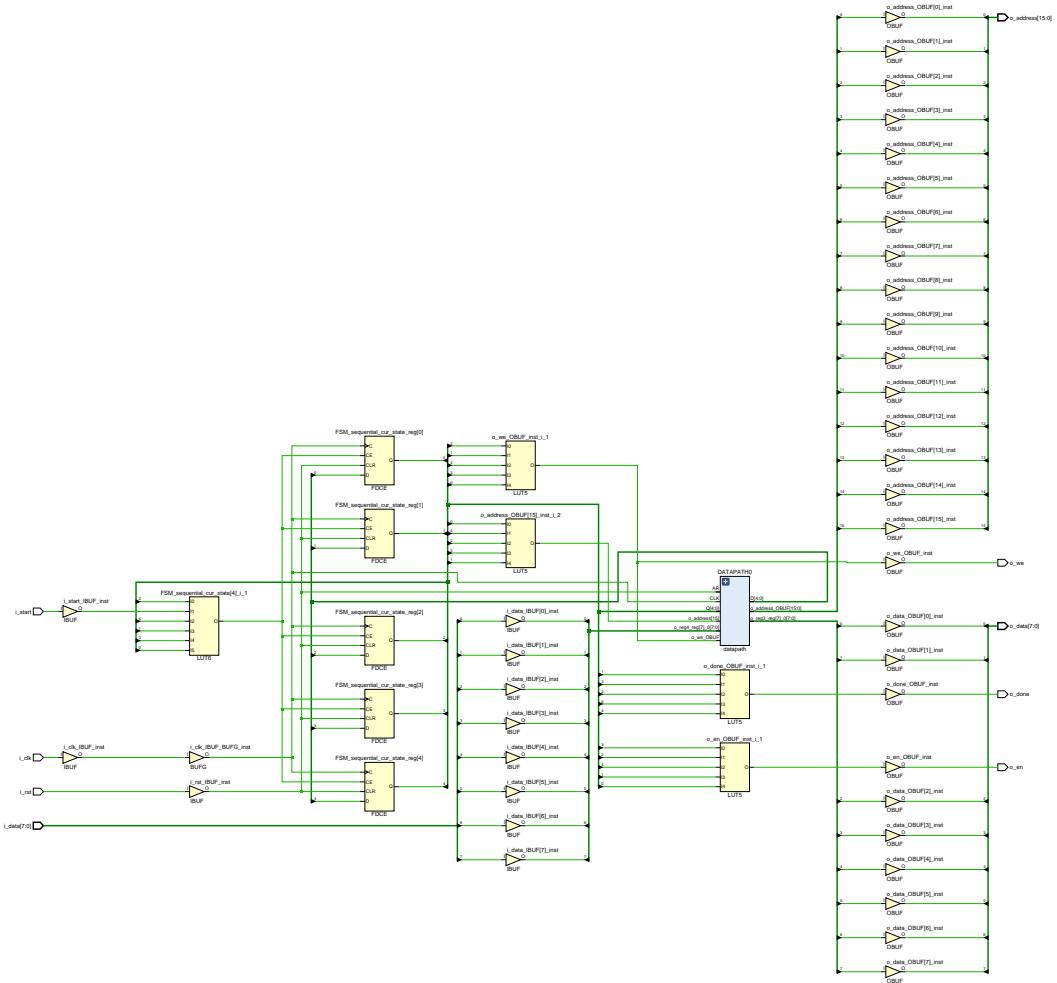


Figura 19: Circuito in post-sintesi con datapath non esplicito

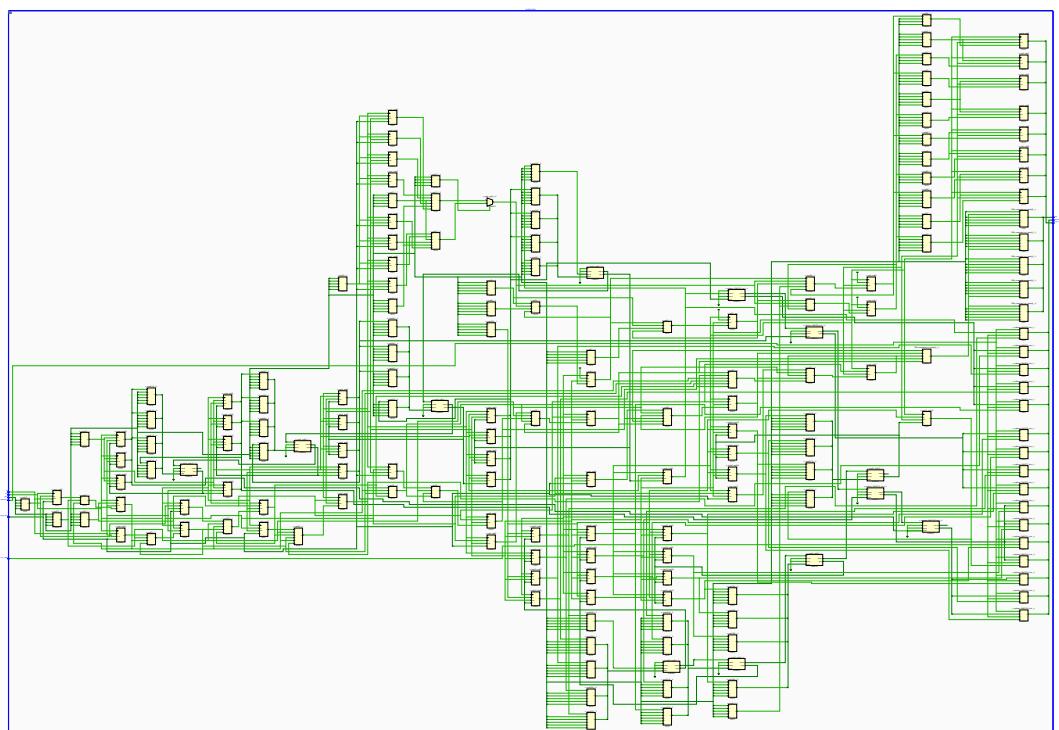


Figura 20: Circuito in post-sintesi del datapath

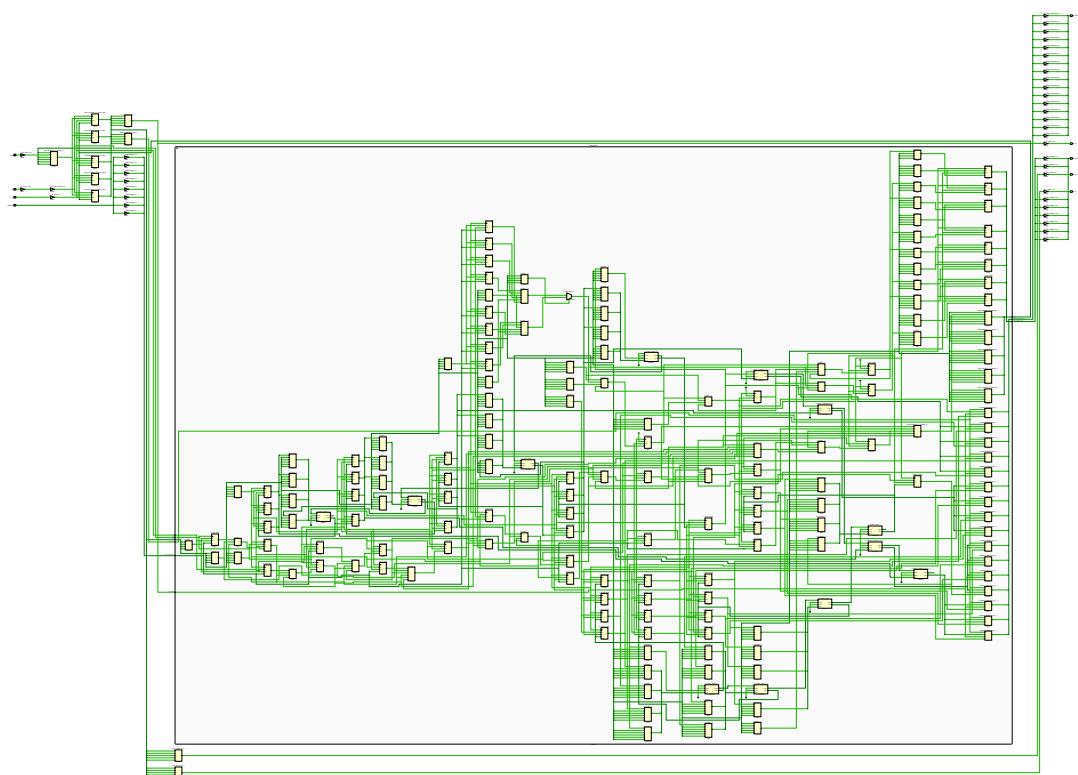


Figura 21: Circuito in post-sintesi completo

3.3 Simulazioni

In questa sezione andremo a descrivere i casi di test forniti e come questi ci hanno permesso di comprendere meglio cosa non funzionasse nel nostro componente, fino all'esecuzione corretta di tutti i test.

3.3.1 Test bench : "tb_seq_min"

In questo test bench si vuole verificare che non avvenga alcuna computazione dei valori, nel caso in cui all'indirizzo di memoria 0 sia contenuto il valore 0000, indicante l'assenza di parole negli indirizzi successivi. Questo è verificato inserendo dei valori nelle celle di memoria seguenti e controllando che agli indirizzi di scrittura le celle rimangano identificamente nulle.

Considerazione: questo tb ci ha permesso di comprendere quale fosse il nostro problema nella gestione del segnale di o_{end} , facendoci rendere conto di un errore in fase di dichiarazione dei segnali.

3.3.2 Test bench : "tb_reset"

In questo test bench si vuole verificare la corretta gestione di un segnale di reset, che viene impostato alto asincronamente rispetto all'usuale utilizzo del componente.

3.3.3 Test bench : "tb_re_encode"

In questo test bench viene effettuato il controllo sulla correttezza della gestione di segnali di tipo i_{start} , senza l'utilizzo del segnale di reset i_{rst} . E' perciò possibile comprendere se il componente descritto è in grado di iniziare nuovamente un ciclo di traduzione senza essere esplicitamente resettato allo stato $S0$.

3.3.4 Test bench : "tb_doppio_uguale"

In questo test bench viene effettuato il controllo sulla corretta lettura e scrittura dei dati dalla memoria. Viene infatti controllato che i dati presenti all'inizio del test siano gli stessi presenti dopo la prima esecuzione della routine che viene doppiamente ripetuta: garantisce perciò che i dati presenti in memoria vengano correttamente letti, ma non sovrascritti.

3.3.5 Test bench : "tb_seq_max "

In questo test bench viene effettuato il controllo sull'utilizzo del componente con una sequenza avente la massima lunghezza possibile.

3.3.6 Test bench : "tb_tre_reset "

In questo test bench viene effettuato il controllo del funzionamento del componente in caso di segnale di reset venga attivato più volte.

4 Conclusioni

Dai risultati sperimentali e dalla sintesi ottenuta riteniamo che l'architettura progettata del codificatore convoluzionale rispetti le specifiche richieste. I casi di test forniti sono stati eseguiti correttamente e hanno contribuito al raffinamento del progetto fino al momento della consegna.

La descrizione hardware tramite linguaggio VHDL è completamente diversa dai linguaggi di programmazione, perciò durante la definizione del progetto abbiamo incontrato diversi problemi relativi ad alcune errate o mancanti considerazioni.

Grazie al lavoro svolto sul progetto abbiamo guadagnato la capacità di modificare il livello di astrazione con cui pensare al progetto, dovendo passare da alto a basso livello di astrazione, e viceversa, per capire come effettivamente occorresse descrivere il componente. Questa è stata dunque una possibilità per slegarci dal puro punto di vista del programmatore e dunque di poterci porre delle domande sull'effettivo funzionamento dei componenti fisici dei quali talvolta ignoriamo la complessità e l'utilizzo.