Prova Finale (Progetto di Reti Logiche)
Prof. Gianluca Palermo - Anno 2021/2022

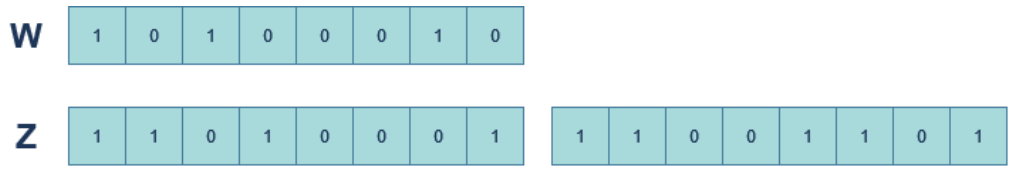
Francesco Palumbo (Codice Persona 10711027 - Matricola 937110)
Samuele Scherini (Codice Persona 10674683 - Matricola 933526)

Indice

1 Introduzione

1.1 Scopo del progetto

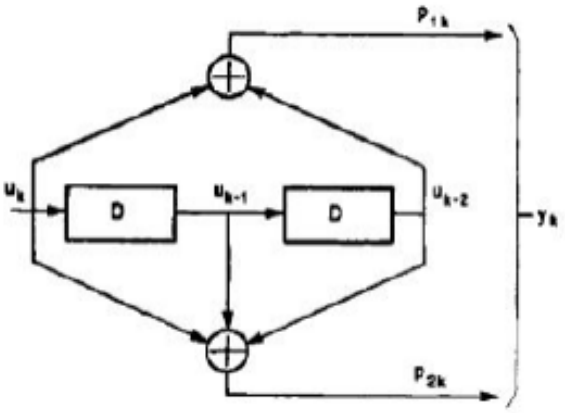
La specifica del progetto prevede l'implementazione di un modulo hardware - descritto in VHDL - che funga da convolutore e si interfacci con una memoria predefinita dai testbench.
Per ogni parola in input letta dalla RAM, vengono generate due parole da 8 bit in output, le quali vengono successivamente scritte in memoria.



La figura mostra il processo di convoluzione applicato al vettore di ingresso W che codifica in binario il numero 162.

1.2 Specifiche del progetto

Tale modulo riceve in ingresso una sequenza continua di W parole, lunghe 8 bit: ogni parola viene serializzata, generando così un flusso continuo di 1 bit, su cui viene successivamente applicato un codice convoluzionale $\frac{1}{2}$.
Questo processo restituisce in output una sequenza di parallelizzazione "Z" a valle di un processo di convoluzione, secondo lo schema rappresentato in figura.



Codificatore convoluzionale con tasso di trasmissione $\frac{1}{2}$.

Questa operazione genera in uscita un flusso continuo Y. Il flusso Y è ottenuto come concatenamento alternato dei due bit di uscita. Utilizzando la notazione riportata in figura, il bit uk genera i bit p1k e p2k che sono poi concatenati per generare un flusso continuo yk.
La sequenza d'uscita Z è la parallelizzazione, su 8 bit, del flusso continuo yk.

1.3 Interfaccia del componente

Il componente da descrivere deve avere la seguente interfaccia:

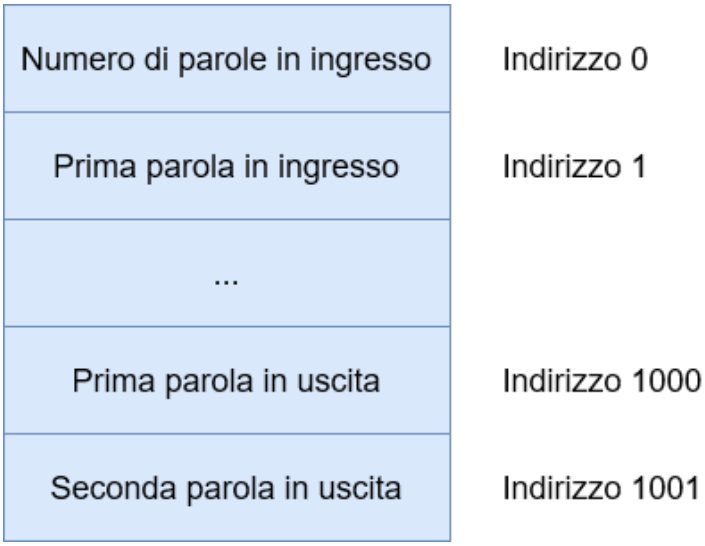
```
entity project_reti_logiche is
port (
    i_clk : in std_logic;
    i_rst : in std_logic;
    i_start : in std_logic;
    i_data : in std_logic_vector(7 downto 0);
    o_address : out std_logic_vector(15 downto 0);
    o_done : out std_logic;
    o_en : out std_logic;
    o_we : out std_logic;
    o_data : out std_logic_vector (7 downto 0)
);
end project_reti_logiche;
```

In particolare:

1. i_clk è il segnale di CLOCK in ingresso generato dal TestBench.
2. i_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START.
3. i_start è il segnale di START generato dal Test Bench.
4. i_data è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura.
5. o_address è il segnale (vettore) di uscita che manda l'indirizzo alla memoria.
6. o_done è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria.
7. o_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura.
8. o_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0.
9. o_data è il segnale (vettore) di uscita dal componente verso la memoria.

1.4 Dati e Memoria

Secondo le specifiche, i dati vengono letti e scritti in una memoria con indirizzamento al byte, ed ogni dato è lungo 8 bit.
Nell'indirizzo 0 è memorizzato il numero di parole da leggere: tramite la sua lettura ci è possibile definire il numero di convoluzioni che il componente dovrà eseguire.
Proseguendo, dall'indirizzo 1 al 255 sono presenti le eventuali parole - fino all'indirizzo 255 solo nel caso in cui il numero di parole in input è massimo.
La scrittura, invece, inizia dall'indirizzo 1000 (DECIMALE) e per ogni parola letta, la macchina ne scrive due in memoria in indirizzi consecutivi, sino ad arrivare all'indirizzo massimo 1509.



1.5 Ulteriori note della specifica

Il modulo partirà nella elaborazione quando un segnale START in ingresso verrà portato a 1. Il segnale di START rimarrà alto fino a che il segnale di DONE non verrà portato alto.
Al termine della computazione (e una volta scritto il risultato in memoria), il modulo da progettare deve alzare (portare a 1) il segnale DONE che notifica la fine dell'elaborazione.
Il segnale DONE deve rimanere alto fino a che il segnale di START non è riportato a 0.
Un nuovo segnale start non può essere dato fin tanto che DONE non è stato riportato a zero. Se a questo punto viene rialzato il segnale di START, il modulo dovrà ripartire con la fase di codifica.
Il modulo deve essere dunque progettato per poter codificare più flussi uno dopo l'altro.
Ad ogni nuova elaborazione (quando START viene riportato alto a seguito del DONE basso), il convolutore viene portato nel suo stato iniziale 00 (che è anche quello di reset).
La quantità di parole da codificare sarà sempre memorizzata all'indirizzo 0 e l'uscita deve essere sempre memorizzata a partire dall'indirizzo 1000.
Il modulo deve essere progettato considerando che prima della prima codifica verrà sempre dato il RESET al modulo. Invece, come descritto nel protocollo precedente, una seconda elaborazione non dovrà attendere il reset del modulo ma solo la terminazione della elaborazione.

2 Scelte progettuali

2.1 Descrizione della macchina

Lo scopo del progetto è quello di descrivere un componente hardware che soddisfi tale specifica. Abbiamo deciso di progettare tale componente attraverso la definizione di una FSM(D), ovvero una macchina a stati finiti con Datapath: questa scelta di implementazione è dovuta alla presenza di variabili ed espressioni algoritmiche che guidano la normale macchina a stati finiti nella transizione al prossimo stato. In particolare, l'elaborazione parte in seguito ad un livello alto del segnale "i_start", effettuando la prima transizione dallo stato iniziale "RESET" allo stato successivo, da cui proseguirà la computazione fino alla lettura del bit '1' del segnale "o_done". Tale elaborazione può, inoltre, ricevere in input un segnale di "i_rst" che forzerà la macchina a tornare nello stato iniziale, detto "di reset". Si può notare come tutti i segnali abbiano il rispettivo segnale di next: questo è dovuto al fatto che ogni assegnamento di valori all'interno dei vari stati non avviene in modo istantaneo, ma vengono aggiornati al termine della transizione. Abbiamo progettato la nostra macchina scomponendola in due processi: il primo ha lo scopo di gestire i registri e i loro aggiornamenti - a seconda che i segnali ricevuti siano di reset o di un evento del clock -, il secondo invece descrive ogni stato con le sue funzionalità e definisce lo stato prossimo in cui continuerà l'elaborazione. La macchina si compone di 14 stati, 4 dei quali si occupano della convoluzione delle parole in ingresso; la FPGA da noi scelta è la "xc7a200tfbg484-1". Segue la descrizione nel dettaglio di tutti gli stati.

2.2 Stati della macchina

2.2.1 RESET

Rappresenta lo stato iniziale della macchina, la cui unica funzione è quella di attendere che venga portato ad '1' il segnale i_start, dopo il quale procede con i successivi stati. Si torna in tale stato se viene alzato il segnale i_rst.

2.2.2 REQUEST_LENGTH

In tale stato viene richiesta alla RAM la lettura dell'indirizzo 0, nel quale è contenuto il numero di parole da leggere, che indica alla macchina quante convoluzioni eseguire. Ciò viene fatto portando ad '1' il segnale o_en e a '0' o_we.

2.2.3 WAIT_RAM

Stato intermedio che verifica la ricezione del numero di parole: se la lettura è già stata effettuata, la computazione continua con la lettura della parola, altrimenti prosegue con il salvataggio del numero in una variabile.

2.2.4 GET_LENGTH

Il suo compito consiste nella conversione del dato letto in un intero ed il successivo salvataggio nella variabile 'length'.

2.2.5 CHECK_LENGTH

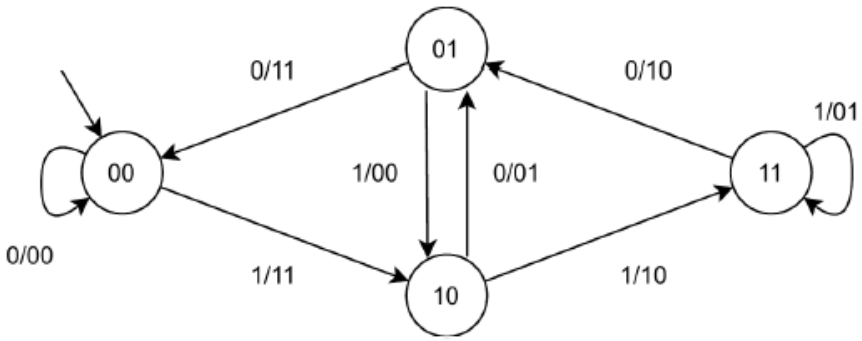
Stato di controllo della variabile 'length', in cui viene deciso se continuare a leggere parole in input o se la computazione può terminare (per length = 0). In tal caso, viene alzato il segnale di o_done che comunica il termine della computazione e il raggiungimento dello stato finale (DONE).

2.2.6 GET_WORD

Stato simile a GET_LENGTH, nel quale viene salvato i_data nella variabile word. Una volta superato questo stato, si passa alla vera e propria convoluzione dei dati in input.

2.2.7 CONV(CONV0, CONV1, CONV2, CONV3)

Stato della convoluzione: sulla base della FSM delle specifiche, tale stato (scomposto in 4 piccoli stati) effettua la convoluzione della parola. Ricevendo in ingresso un bit per volta, seleziona lo stato da raggiungere e produce in risposta due bit a seconda del bit letto, formando così le due parole in output. Ognuno di questi 4 stati opera secondo dei controlli su alcune variabili: in particolare, se il contatore i è minore di 0, la computazione può passare alla scrittura del dato in memoria e il salvataggio dello stato in cui la convoluzione termina (in un segnale chiamato state_conv, il quale servirà ad indicare lo stato d'inizio della prossima convoluzione); altrimenti continua la sua elaborazione.



In particolare CONV0 sarà 00 ecc.

2.2.8 MEM_WRITE

Stato intermedio che prepara la scrittura del dato: una volta salvata la parola nel segnale word, esso porta ad '1' i segnali relativi alla memoria o_en e o_we, cambiando gli indirizzi di scrittura.

2.2.9 WRITE_W1

Viene scritta in memoria la prima parola in output, lunga 1 byte e generata dalla convoluzione dei primi 4 bit della parola in input (infatti, ogni bit in input produce due bit in output) e successivamente prepara nuovamente la scrittura della seconda parola, aggiornando i segnali di o_en, o_we e gli indirizzi della RAM.

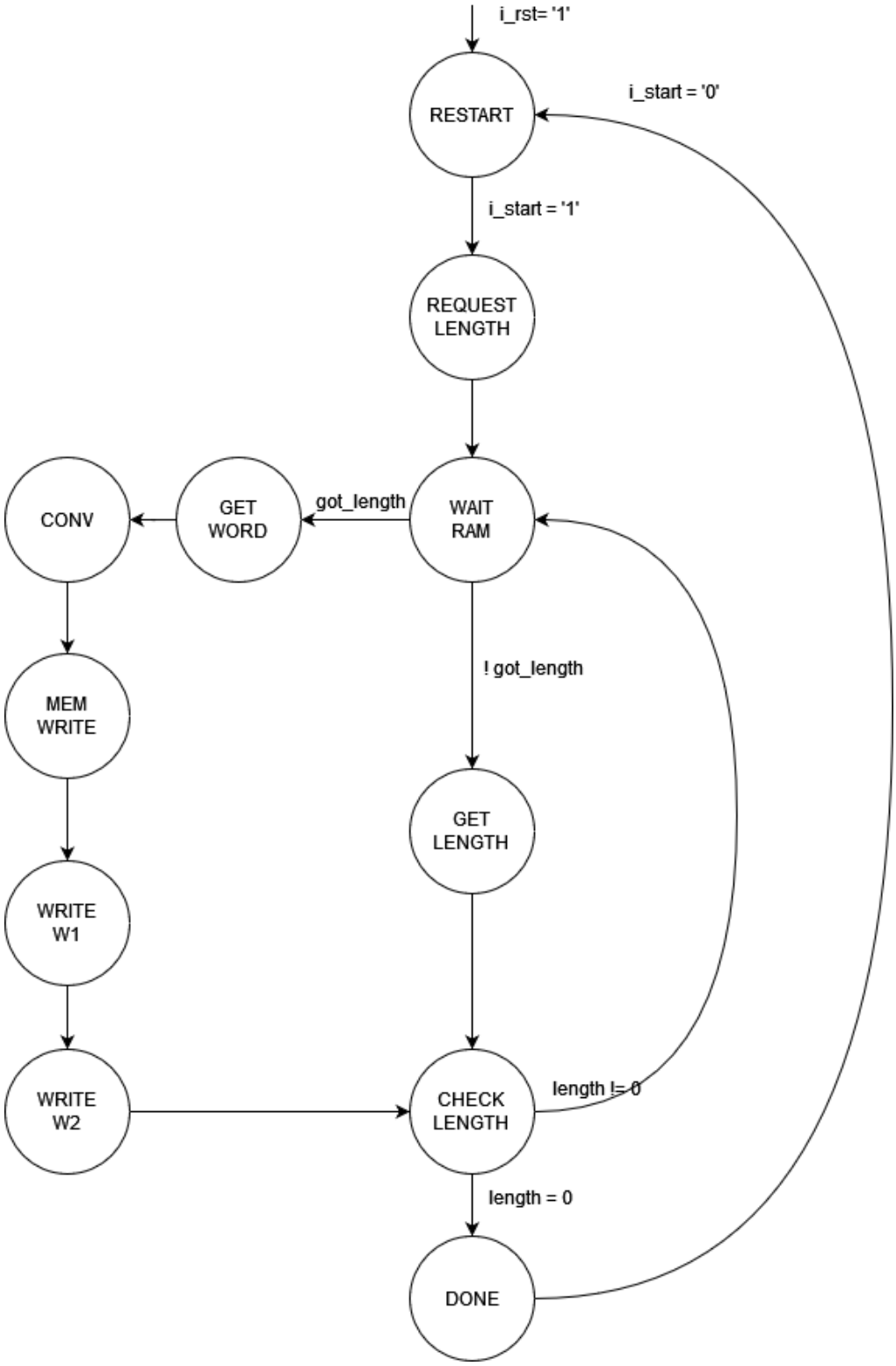
2.2.10 WRITE_W2

Stato di completamento della scrittura, in cui (dopo aver scritto in memoria) viene incrementato l'indirizzo di scrittura, pronto per il prossimo eventuale ciclo di convoluzione; inoltre, viene decrementato il numero di parole rimanenti. Da qui riparto dallo stato CHECK LENGTH, il quale indica alla macchina se la computazione è terminata o deve andare avanti.

2.2.11 DONE

Stato finale dell'intera elaborazione: ci si arriva dopo aver verificato che il numero di parole lette sia uguale a 0. In tal caso, vengono reinizializzati tutti gli indirizzi (sia di lettura che di scrittura) e i vari segnali utilizzati. In particolare, prima di arrivare in questo stato viene portato ad '1' il segnale

o_done, che indica alla macchina il termine della computazione, il quale verrà tenuto alto finché non verrà abbassato il segnale i_start.

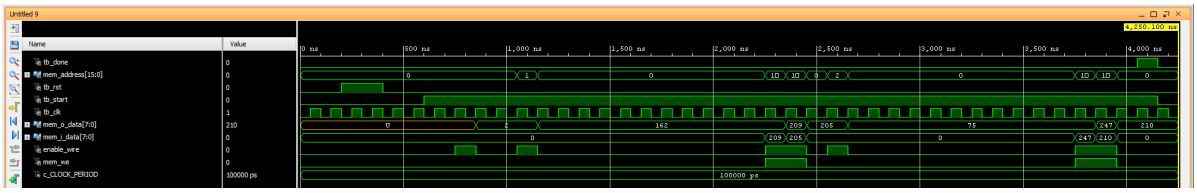


3 Risultati dei test

Al fine di verificare il corretto funzionamento della nostra macchina a stati, sono stati generati vari testbench, oltre al test d'esempio già fornitoci.
Tali test si propongono di coprire i cammini e le computazioni potenzialmente critiche per la macchina, in modo tale da evitare la presenza di situazioni non ben definite.
Il componente è stato sottoposto, per tutti i test, sia in simulazione pre-sintesi (Behavioral Simulation) che in post-sintesi (Functional Simulation).
Di seguito sono mostrati i testbench più significativi utilizzati, seguiti dal relativo screenshot che mostra l'andamento dei segnali a seguito della computazione. In particolare:

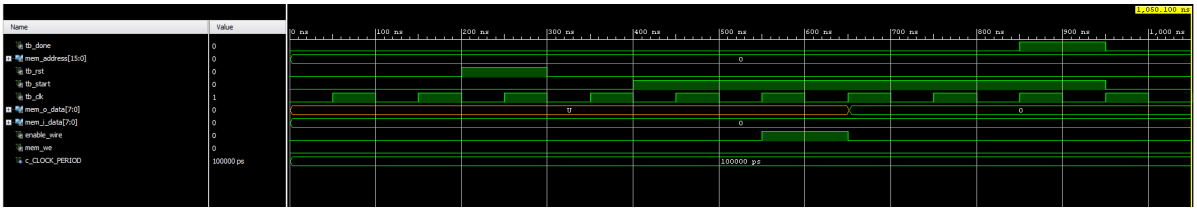
3.1 Testbench fornito

Il test fornisce 2 parole in input: 162 e 75; non sono presenti segnali di reset.
Esso ci ha permesso di verificare il corretto funzionamento base della macchina, in particolare gli stati della convoluzione e l'interfaccia con la memoria (RAM).



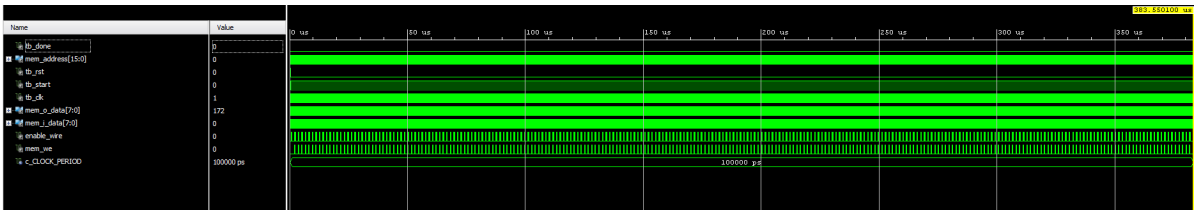
3.2 Numero minimo di parole (length = 0)

E' il caso più semplice, in cui il numero di parole da leggere è uguale a 0: durante l'elaborazione, nello stato di controllo del segnale length la macchina prosegue direttamente nello stato finale, senza entrare in alcun ciclo di lettura delle parole.



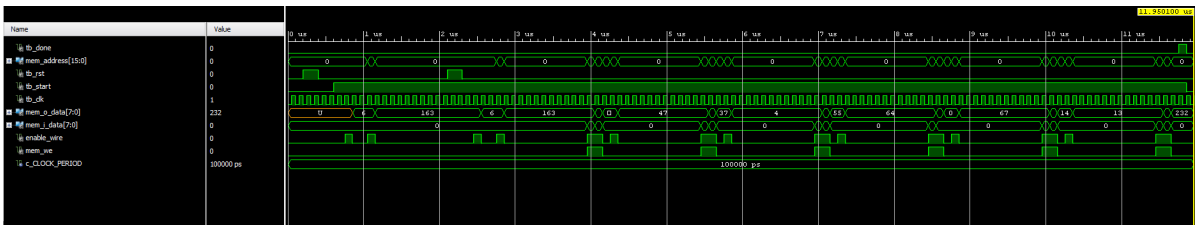
3.3 Numero massimo di parole (length = 255)

Secondo caso limite, in cui il numero di parole è massimo, ovvero 255. Durante tale computazione la macchina esegue 255 cicli di lettura parole, come si può osservare nel seguente screen di simulazione.



3.4 Reset asincrono

Tale test ha lo scopo di verificare che la macchina, prendendo in input un segnale di reset asincrono, si riporti correttamente allo stato iniziale.



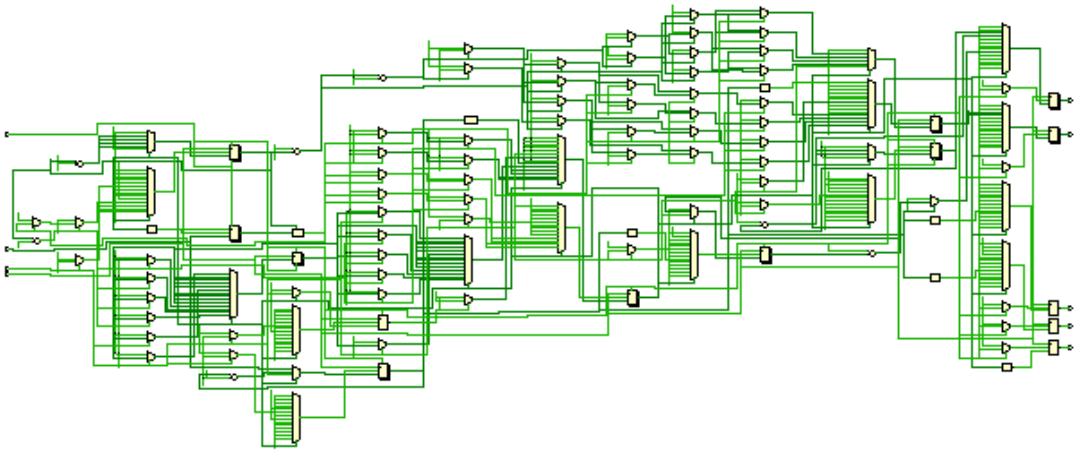
4 Conclusioni

4.1 Risultati della sintesi

Il componente sintetizzato supera correttamente entrambe le simulazioni di pre-sintesi e post-sintesi (Behavioral e Post-Synthesis Functional). Qui di seguito è riportato un confronto tra i tempi di simulazione dei due casi limite che portano la macchina verso la più breve e la più lunga simulazione:

1. 1050ns - tempo di simulazione (Behavioral) con numero di parole = 0.
2. 383550ns - tempo di simulazione (Behavioral) con numero di parole = 255.

I tempi di simulazione sono pressoché identici tra Behavioral e Post-Synthesis Functional.



Questo in figura è lo schematico realizzato dalla RTL Analysis

4.2 Ottimizzazioni

Partendo dalla macchina iniziale, sono state effettuate diverse ottimizzazioni. La prima consiste nella riduzione degli stati: inizialmente era composta da 16 stati, tra cui due stati di attesa della memoria, uno per la lettura e l'altro per la scrittura.
Successivamente, abbiamo unificato i due stati e implementato un ciclo che inizia non appena termina la scrittura in memoria.
Un ulteriore stato di richiesta alla RAM "REQUEST WORD" è stato eliminato e la sua funzione è stata implementata all'interno dello stato precedente, ovvero "CHECK LENGTH".
Infine, dopo la prima simulazione in post-sintesi, abbiamo notato che alcuni componenti erano inizialmente implementati tramite dei latch: abbiamo riscritto parte del codice VHDL per evitarne l'utilizzo.

La progettazione del modulo, la sua descrizione nel linguaggio di descrizione VHDL e l'utilizzo dell'applicativo VIVADO ci hanno permesso di avere una visione più completa e nel dettaglio degli argomenti trattati nel corso; di particolare interesse è stata la gestione dell'interfaccia della RAM con il componente da noi implementato, e la visualizzazione degli effettivi ritardi nella trasmissione dei segnali.