Prova Finale – Reti Logiche (prof. Gianluca Palermo)

Giacomo Stefanizzi 10610854

A.A 2021/22

1 Introduzione

1.1 Scopo del progetto

Si vuole implementare un modulo hardware, descritto in VHDL, che elabori con il codice convoluzionale ½ un flusso continuo da 1 bit.

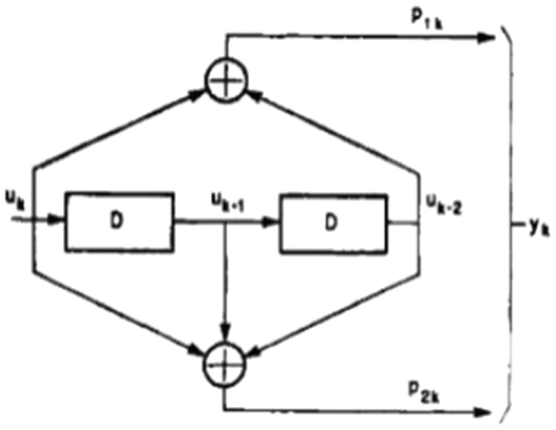
1.2 Specifiche Generali

Il modulo riceve in ingresso una sequenza continua di W parole, ognuna da 8 bit, e restituisce in uscita una sequenza continua di Z parole, sempre da 8 bit ciascuna.

Il modulo serializza ognuna delle parole generando un flusso U, su cui viene applicato il codice convoluzionale ½, dove ogni bit (U_k) viene codificato con 2 bit (P_{1k} e P_{2k}). Questa operazione genera in uscita un flusso Y, che è ottenuto attraverso la concatenazione dei bit P_{1k} e P_{2k} in uscita. Si nota che la lunghezza del flusso U è 8*W, mentre la lunghezza del flusso Y è 8*W*2, infatti Z = 2*W.

La sequenza d'uscita Z, infine, sarà la parallelizzazione su 8 bit del flusso continuo Y.

Il convolutore è una macchina sequenziale sincrona con un clock globale e possiede un segnale di reset.



1 - Codificatore convoluzionale con tasso di trasmissione 1/2

1.3 Dati e descrizione memoria

Le parole, ognuna di dimensione 8 bit, sono memorizzate in una memoria con indirizzamento al byte.

- L'indirizzo 0 è riservato al valore che indica il numero di parole nella sequenza in ingresso (massimo 255 byte);
- Dall'indirizzo 1 in poi, sono memorizzate le parole della sequenza d'ingresso;
- Dall'indirizzo 1000 in poi, vengono memorizzate le parole della sequenza d'uscita.

1.4 Esempio di funzionamento

Un esempio di funzionamento che mostra il contenuto della memoria al termine dell'elaborazione è il seguente:

- *Sequenza in entrata (W):* 10100010 01001011
- *Sequenza in uscita (Z):* 1010001 11001101 11110111 11010010

INDIRIZZO	VALORE	COMMENTO
0	2	Byte di lunghezza sequenza d'ingresso
1	162	Primo byte della sequenza da codificare
2	75	
[…]		
1000	209	Primo byte della sequenza di uscita
1001	205	
1002	247	
1003	210	

La prima parola della sequenza d'ingresso viene elaborata nel modo seguente:

- *Byte (o parola) in ingresso:* 10100010

Partendo dal bit più significativo, viene serializzata come 1 al tempo *t*, 0 al tempo *t+1*, 1 al tempo *t+2*, 0 al tempo *t+3*, 0 al tempo *t+4*, 0 al tempo *t+5*, 1 al tempo *t+6* e 0 al tempo *t+7*).
Applicando l'algoritmo convoluzionale si ottiene la seguente serie di coppie di bit:

T	0	1	2	3	4	5	6	7
Uk	1	0	1	0	0	0	1	0
P1k	1	0	0	0	1	0	1	0
P2k	1	1	0	1	1	0	1	1

Il concatenamento dei valori P1k e P2k per produrre l'uscita Z segue il seguente schema:
P1k al tempo *t*, P2k al tempo *t*, P1k al tempo *t+1*, P2k al tempo *t+1* e così via.
Il risultato prodotto, ovvero i primi 2 byte dell'uscita Z, sarà 11010001 11001101

1.5 Interfaccia del componente

Il componente ha una interfaccia così definita:

```
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:

- il nome del modulo deve essere 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 Test Bench;
- `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 di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- `o_en` è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- `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;
- `o_data` è il segnale (vettore) di uscita dal componente verso la memoria.

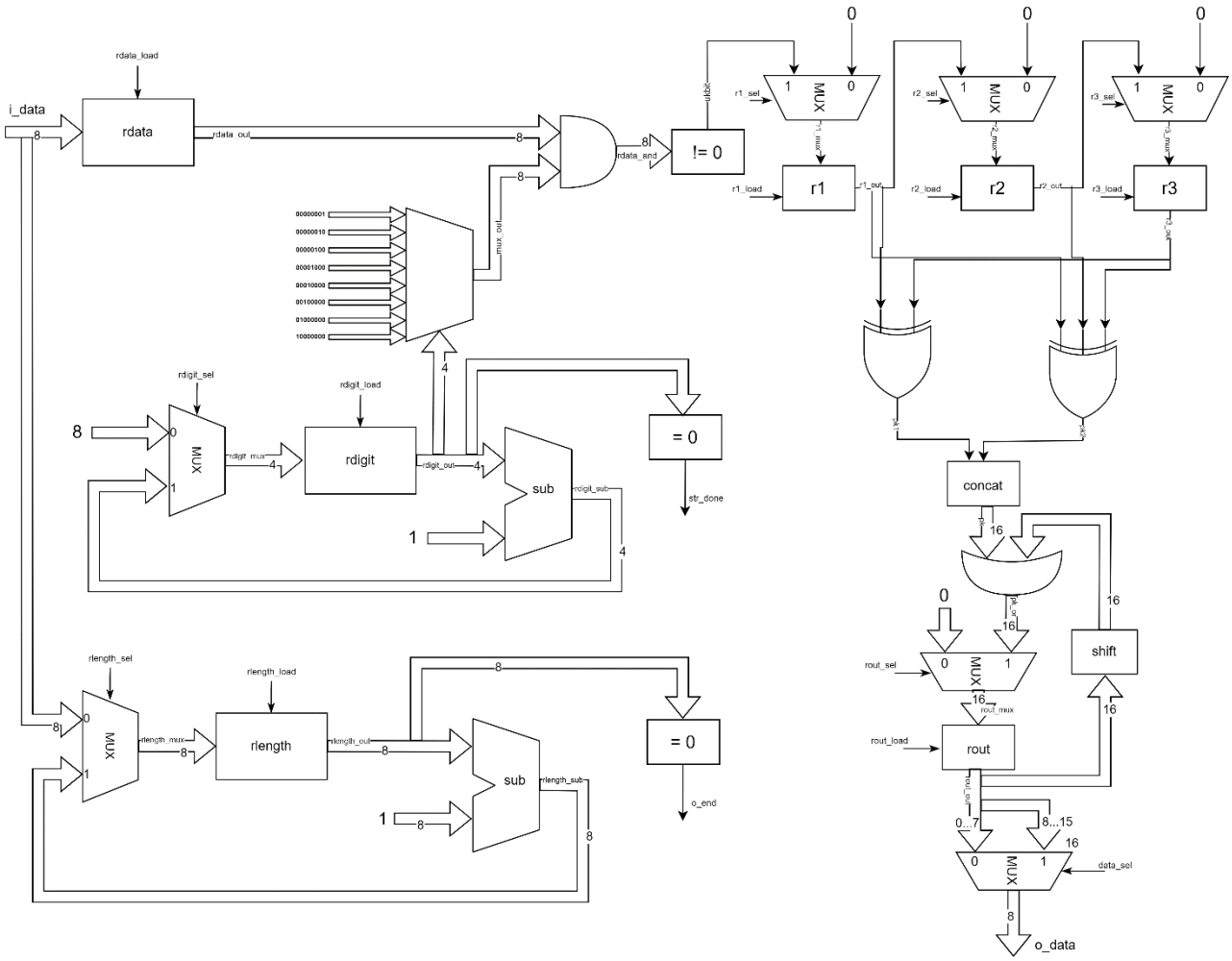
2 Architettura

2.1 Datapath

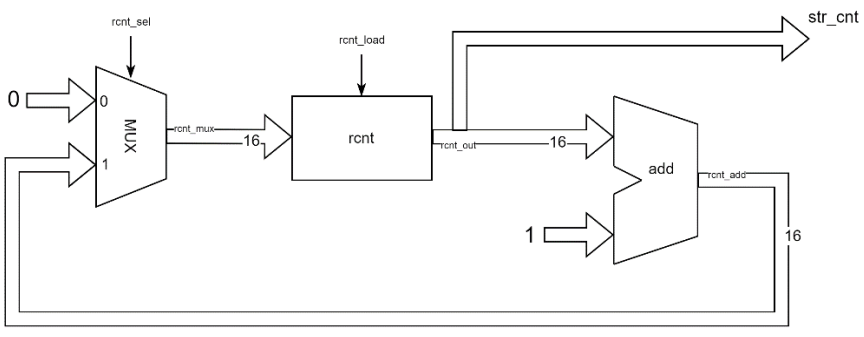
Il modulo che si occupa dell'elaborazione dei dati è suddiviso in due sotto-moduli:

- il primo (quello principale), si occupa dell'elaborazione delle parole in ingresso e della conseguente generazione delle parole da scrivere in memoria
- il secondo, si occupa del conteggio delle parole elaborate, al fine di poter calcolare gli indirizzi di lettura e scrittura in memoria.

MODULO 1:



MODULO 2:



L'algoritmo implementato, dopo aver ricevuto in ingresso una parola, ne seleziona una cifra alla volta attraverso un AND con il segnale **`mux_out`**. Il risultato viene passato attraverso un confronto a zero che determina il bit Uk. Quest'ultimo viene poi caricato di volta in volta nei registri *r1*, *r2*, *r3*, dai quali vengono prelevati gli Uk (Uk, Uk+1, Uk+2) per effettuarne l'operazione XOR.
Dopo la concatenazione di Pk_1 con Pk_2, il risultato viene memorizzato nel registro *rout*, per essere poi concatenato con i successivi Pk.
Quando viene elaborata l'ultima cifra della parola, il segnale **`str_done`** viene portato ad 1 e *rout* conterrà le due parole di output.
Quando infine viene elaborata l'ultima parola, il segnale **`o_end`** viene portato ad 1.

Di seguito sono elencati i diversi registri atti a memorizzare informazioni utili durante l'elaborazione:

- ***rdata***: memorizza la parola in input da elaborare;
- ***rdigit***: memorizza l'indice della cifra corrente che si sta valutando come Uk;
- ***r1, r2, r3*** : memorizzano gli Uk bit per poter effettuare le operazioni di XOR;
- ***rlength***: memorizza il numero di parole da elaborare, e viene decrementato ad ogni parola elaborata;
- ***rcnt***: memorizza il numero di parole elaborate, ad ogni parola elaborata.
- ***rout***: memorizza le due parole di output prodotte dall'elaborazione.

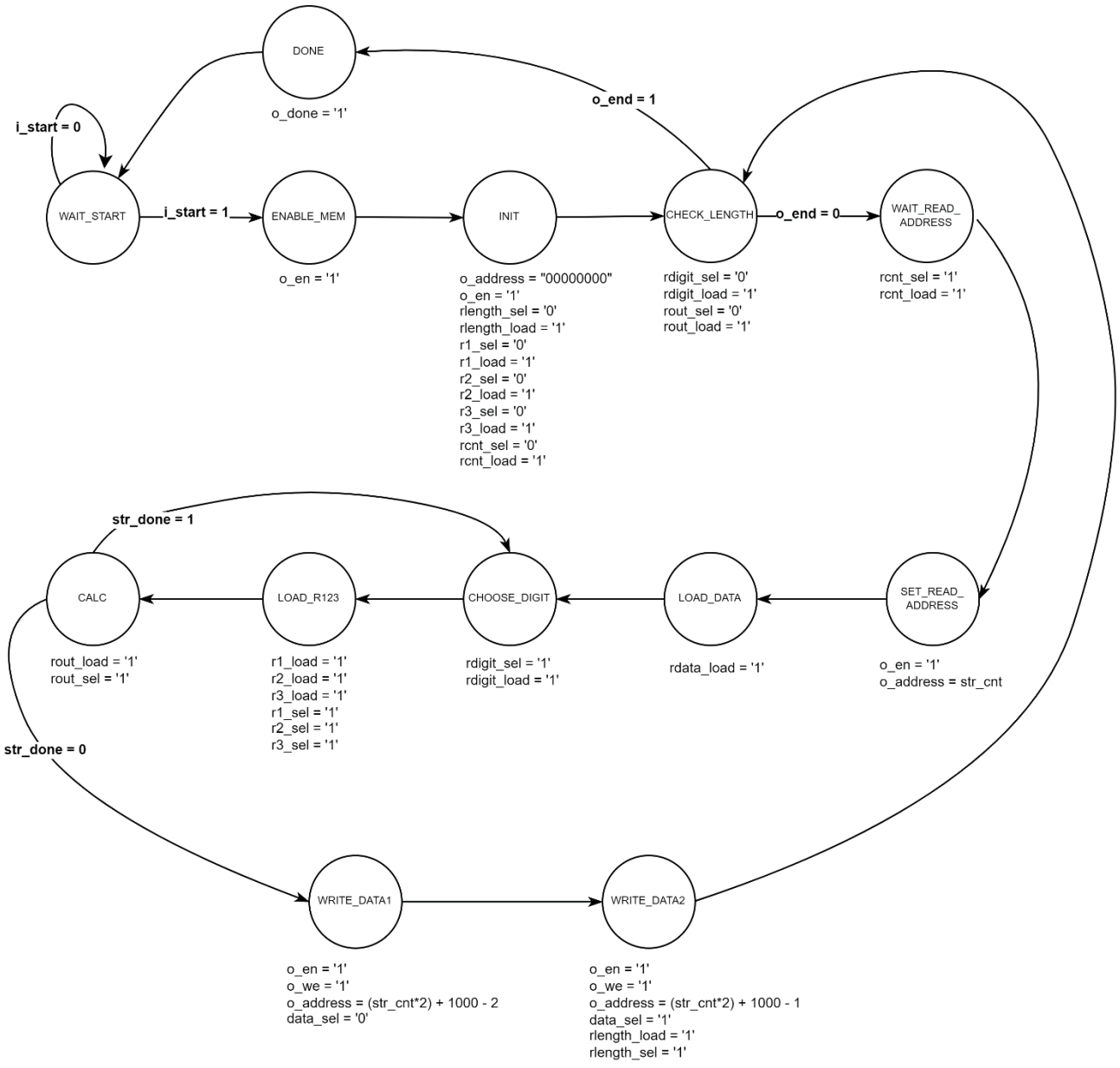
2.2 Macchina a stati

Per gestire il modulo *datapath*, è stato opportuno definire una FSM composta da 13 stati, riportati qui di seguito:

- **WAIT_START:**
  stato iniziale in cui il componente attende che il segnale **`i_start`** sia portato ad 1 dalla testbench;
- **ENABLE_MEM**:
  stato in cui viene abilitata la lettura in memoria;
- **INIT:**
  stato in cui vengono inizializzati i registri e impostato il primo indirizzo di lettura;
- **CHECK_LENGTH:**
  stato in cui viene controllato il numero di parole mancanti da elaborare. Se è 0, lo stato successivo sarà DONE, altrimenti si prosegue;
- **WAIT_READ_ADDRESS:**
  stato in cui viene incrementato di 1 il numero di parole lette e calcolato il successivo indirizzo di memoria;
- **SET_READ_ADDRESS:**
  stato in cui viene assegnato il nuovo indirizzo di lettura;
- **LOAD_DATA:**
  stato in cui viene memorizzata la nuova parola da elaborare;
- **CHOOSE_DIGIT:**
  stato in cui viene selezionata una nuova cifra della parola in elaborazione;
- **LOAD_R123:**
  stato in cui vengono aggiornati i registri con i nuovi Uk bit;

- **CALC:**
  stato in cui vengono effettuate le operazioni di XOR e concatenamento. Se `str_done` è a 1, si prosegue con gli stati di scrittura in memoria, altrimenti si ritorna a CHOOSE_DIGIT;
- **WRITE_DATA1:**
  stato in cui viene scritta in memoria la prima parola di output;
- **WRITE_DATA2:**
  stato in cui viene scritta in memoria la seconda parola di output;
- **DONE**:
  stato in cui viene portato ad 1 il segnale `o_done`, necessario alla testbench per poter portare il proprio segnale `i_start` a 0 e ricominciare eventualmente con nuovi test.

Il componente è inoltre dotato di un segnale di reset `i_rst`, che se attivato, riporta allo stato iniziale WAIT_START.



3 Risultati sperimentali

3.1 Simulazioni

Per verificare il corretto funzionamento del componente sintetizzato, sono stati effettuati molteplici test con valori casuali e test specifici per coprire i “corner case”, specificati qui di seguito:

- **Sequenza nulla**: il test dà in input una sequenza nulla, ovvero di dimensione 0 parole, e si assicura che non venga scritta alcuna parola in output.
  In questa condizione la FSM, una volta arrivata allo stato CHECK_LENGTH, deve valutare il segnale `o_end` (che sarà 0) e passare immediatamente allo stato di DONE.
- **Sequenza massima**: il test dà in input una sequenza di parole di dimensione massima (ovvero 255 byte) assicurandosi che l’elaborazione di ogni parola venga effettuata nel modo corretto.
- **Codifica multipla**: il test dispone di 3 differenti memorie (quindi di 3 flussi) e si assicura che vengano correttamente codificate una dopo l’altra.
  In questo modo si verifica l’implementazione corretta del protocollo di comunicazione tra componente e testbench.
- **Reset asincrono**: il test effettua un reset durante l’elaborazione, dopodiché viene fatta ripartire. Infine, il test si assicura che l’elaborazione sia stata eseguita correttamente.
  In questo modo si verifica che il componente torni correttamente allo stato di WAIT_START dopo il segnale di reset.

Il componente dimostra di poter superare tutti i test sia in simulazione *Behavioral* che in simulazione *Post-Synthesis Functional*.

3.2 Report di sintesi

L’area occupata dal componente sintetizzato risulta la seguente:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	114	0	134600	0.08
LUT as Logic	114	0	134600	0.08
LUT as Memory	0	0	46200	0.00
Slice Registers	68	0	269200	0.03
Register as Flip Flop	68	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

Per avere un’idea della velocità del componente, è utile analizzare il *Worst Negative Slack* presente nel *Design Timing Summary:*

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	96,351 ns	Worst Hold Slack (WHS):	0,139 ns	Worst Pulse Width Slack (WPWS):	4,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	128	Total Number of Endpoints:	128	Total Number of Endpoints:	69

All user specified timing constraints are met.

4 Conclusioni

La progettazione di un datapath che utilizza dei registri come strumento di memorizzazione dei dati, è risultata molto ordinata ed efficace.

Il componente dimostra di poter lavorare anche a frequenze di clock maggiori, ed è stato ottimizzato per ridurre al minimo il numero di stati.

Ogni parola da elaborare viene caricata solamente una volta, e le due parole in output prodotte vengono scritte in memoria consecutivamente, così da poter ridurre al minimo le operazioni di lettura e scrittura in memoria.