PROGETTO DI RETI LOGICHE AA 2021-2022

Baldelli Andrea, CP: 10673020

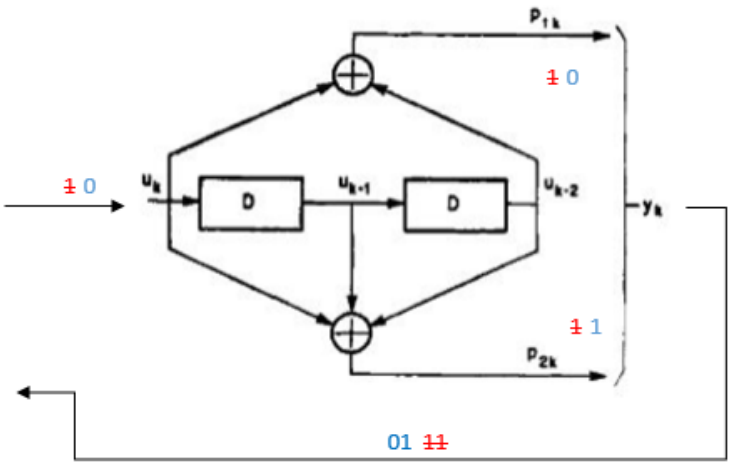
Introduzione

Il progetto di Reti Logiche prevede la descrizione in linguaggio VHDL di un componente hardware in grado di convertire una sequenza di parole di 8 bit in input in una sequenza in output di lunghezza doppia attraverso un codificatore convoluzionale con tasso ½.

Il componente legge all'indirizzo di memoria 0 il numero di parole che dovrà elaborare e successivamente inizia a leggere le parole a partire dall'indirizzo 1.

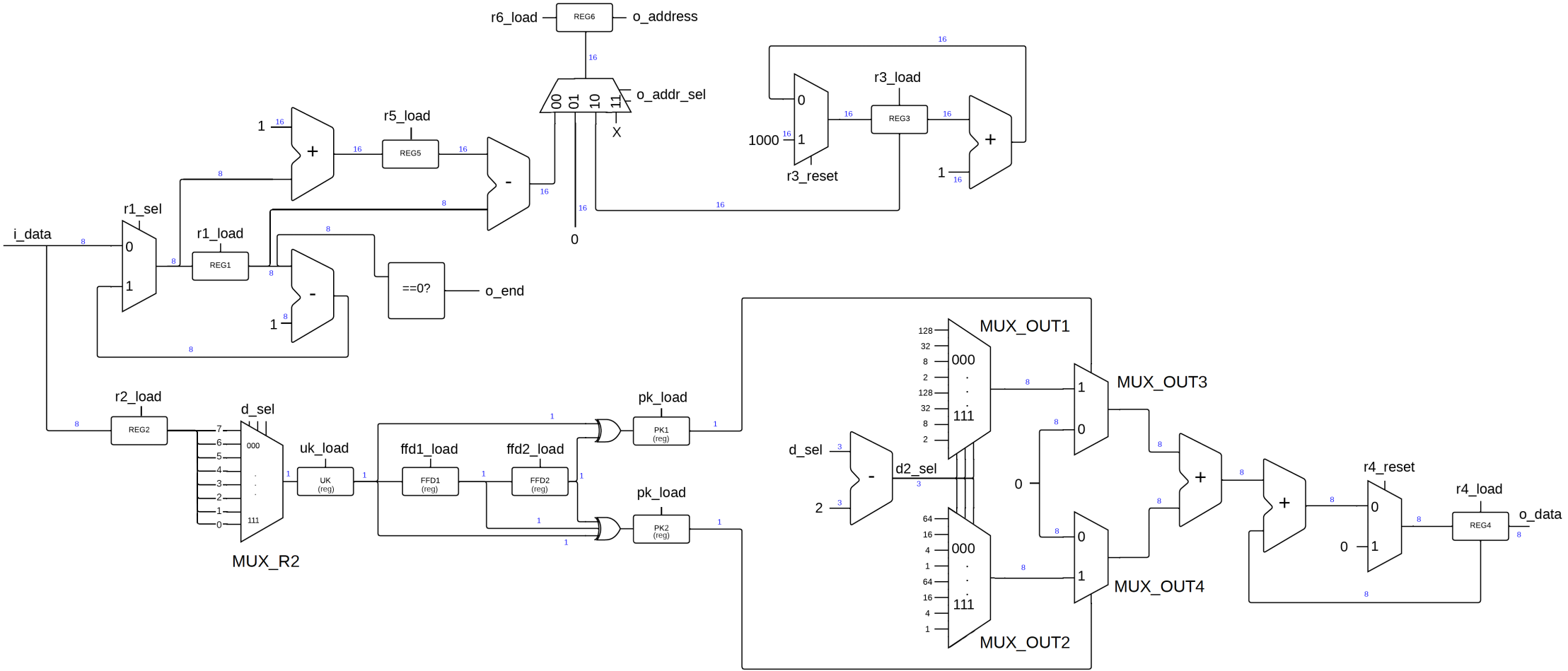
Ogni parola viene analizzata un bit alla volta ed esso arriva in ingresso al codificatore. Questo produce due bit che vengono scritti in sequenza a partire dall'indirizzo di memoria 1000.

INDIRIZZO	VALORE DECIMALE	VALORE BINARIO
0	2	00000010
1	162	10100010
2	75	01001011
...	...	...
...	...	...
...	...	...
1000	209	11010001
1001	205	11001101
1002	...	...
1003	...	...



Architettura

Nelle pagine seguenti sono riportati gli schemi del Datapath e della macchina a stati che controlla il comportamento del componente. Segue la descrizione dei singoli moduli.


r6_load
REG6
o_address
o_addr_sel
00
01
10
11
X
r5_load
REG5
r3_load
REG3
r3_reset
1000
i_data
r1_sel
r1_load
REG1
==0?
o_end
r2_load
REG2
d_sel
uk_load
UK (reg)
ffd1_load
FFD1 (reg)
ffd2_load
FFD2 (reg)
pk_load
PK1 (reg)
PK2 (reg)
MUX_R2
MUX_OUT1
MUX_OUT2
MUX_OUT3
MUX_OUT4
d2_sel
r4_reset
r4_load
REG4
o_data

All'inizio dell'esecuzione viene dato il segnale di reset che serve per scrivere 0 in tutti i registri.

Controllo del numero di iterazioni e calcolo degli indirizzi (parte alta del Datapath)

Registro REG1
Nel registro Reg1 viene inizialmente scritto il numero di parole da elaborare (letto dalla memoria all'indirizzo 0). Successivamente questo valore verrà decrementato attraverso un sottrattore e un multiplexer. Una volta che il registro conterrà il valore 0 sarà messo a 1 il segnale di *o_end* ad indicare che l'esecuzione è terminata.

Registro REG5
Questo registro serve per calcolare (insieme a REG1) l'indirizzo in cui andare a leggere la parola dalla memoria. Viene inizialmente settato con il numero di parole da leggere incrementato di 1 e rimane invariato per tutta l'esecuzione. Questo consente, attraverso la differenza con REG1, di conoscere il prossimo indirizzo in cui andare a leggere.
Ad esempio: ci sono 2 parole da leggere quindi REG1=2 e REG5=3. La prima parola da leggere si trova a REG5-REG1 = 1. REG1 viene decrementato (REG1=1). La prossima parola è all'indirizzo REG5-REG1 = 2 e così via.

Registro REG3
Questo registro serve per calcolare l'indirizzo in cui andare a scrivere in memoria. Viene dapprima inizializzato a 1000 per poi essere di volta in volta incrementato attraverso sommatore e multiplexer.

Registro REG6
Serve per mantenere l'indirizzo di memoria a cui leggere o in cui andare a scrivere. Attraverso il multiplexer posto prima del registro è possibile scegliere il segnale proveniente dalla differenza tra REG5 e REG1 (indirizzo in cui leggere), il segnale proveniente da REG3 (indirizzo in cui scrivere) oppure un segnale costante a 0 (indirizzo iniziale in cui leggere il numero di parole).

Calcolo del risultato (parte bassa del Datapath)

Registro REG2
In questo registro viene scritta di volta in volta la parola da analizzare.

Multiplexer MUX_R2
Questo multiplexer a 3 bit serve per selezionare di volta in volta un bit della parola contenuta in REG2.
Il segnale di selezione *d_sel* assume i valori da 0 a 7, uno per ogni bit di REG2.

Registro UK
Qui viene salvato il bit corrente letto dal registro REG2.

Registri FFD1 e FFD2
Il codificatore fa uso di Flip Flop D quindi utilizzo due registri che vengono aggiornati a cicli di clock successivi. All'arrivo del segnale di clock salvo in FFD2 il contenuto del registro FFD1 e in FFD1 il contenuto di UK. In questo modo il bit della parola viene propagato in tutti e tre i registri uno alla volta.

Registri PK1 e PK2
Contengono i due bit che saranno scritti in memoria. Sono calcolati attraverso delle porte XOR con i segnali provenienti da UK, FFD1 e FFD2.

Multiplexer MUX_OUT1 e MUX_OUT2
Come detto sopra prima di ottenere in PK1 e PK2 i valori corretti occorre che il bit venga propagato in FFD1 e FFD2. Ciò avviene in 2 cicli di clock. Per poter riutilizzare il segnale *d_sel* (che nel frattempo viene incrementato) è necessario sottrarre 2 ottenendo *d2_sel*.

Con *d2_sel* comando due multilexer a ingressi costanti che producono in output un segnale a 8 bit con un solo bit a 1 (potenze di 2).

Multiplexer MUX_OUT3 e MUX_OUT4
L'uscita di questi multiplexer verrà sommata per produrre una porzione del byte finale. Utilizzo i segnali PK1 e PK2 per decidere se sommare 0 o il numero corrispondente all'uscita dei precedenti multiplexer.

<u>Esempio</u>:
In ingresso ho una parola il cui primo bit è 1 (*d_sel*=0).
Dopo due cicli di clock (*d_sel*=2) ottengo i valori:
PK1=1
PK2=1

A questo punto:
d2_sel = (*d_sel* – 2) = 0
quindi:
MUX_OUT1=10000000
MUX_OUT2=01000000.

Essendo PK1 e PK2 entrambi 1:
MUX_OUT3=MUX_OUT1= 10000000
MUX_OUT4=MUX_OUT2= 01000000
Che sommati insieme danno il risultato parziale: 11000000

Se PK1 fosse stato 0 avrei avuto:
MUX_OUT3= 00000000
MUX_OUT4=MUX_OUT2= 01000000
Che sommati insieme danno il risultato parziale: 01000000

Registro REG4
REG4 svolge la funzione di accumulatore delle somme appena descritte.
A metà della parola in ingresso REG4 contiene il primo byte da scrivere in memoria.
In seguito alla scrittura viene rimesso a 0 attraverso il multiplexer (*r4_reset*=1) per poter produrre la seconda parola di output.

Esempio: sequenza di PK1-PK2: 11 01 00 01

Passo 0: REG4=0

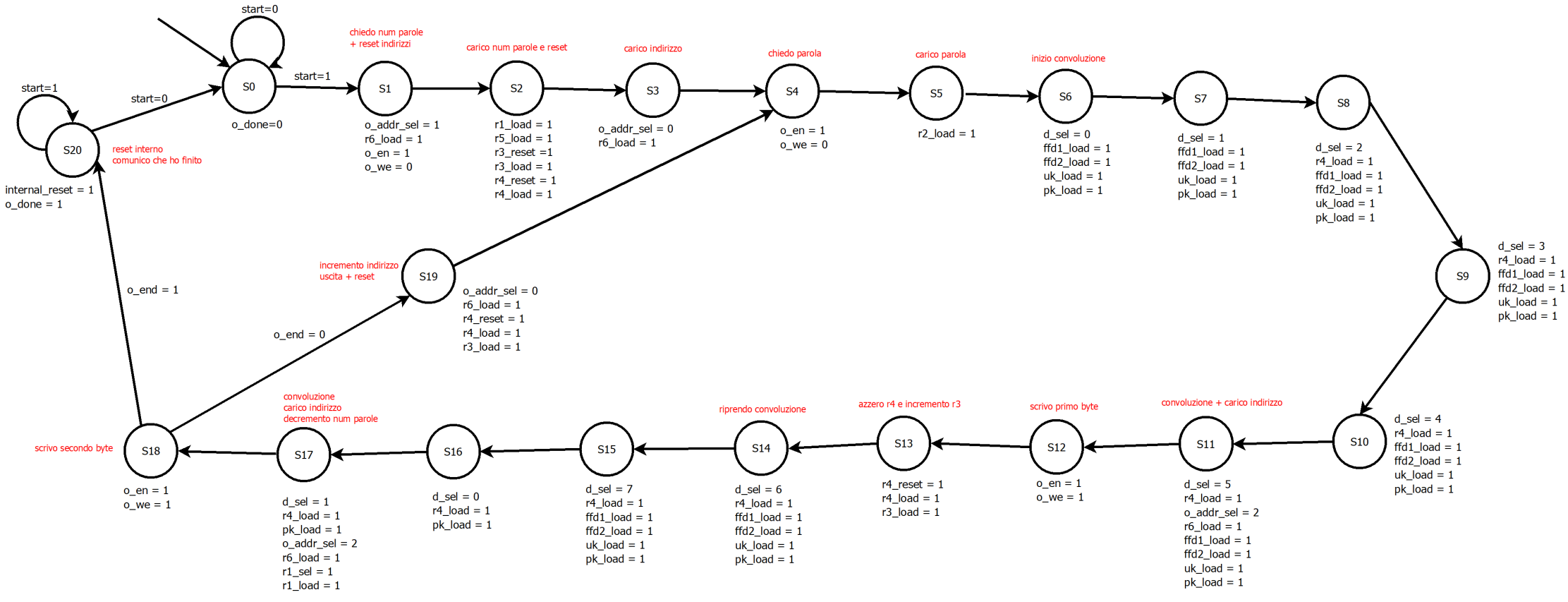
Passo 1:
PK1=1, PK2=1
MUX_OUT1=10000000
MUX_OUT2=01000000
MUX_OUT3=MUX_OUT1= 10000000
MUX_OUT4=MUX_OUT2= 01000000
REG4 = REG4 + MUX_OUT3+MUX_OUT4 = 11000000

Passo 2:
PK1=0, PK2=1
MUX_OUT1=00100000
MUX_OUT2=00010000
MUX_OUT3= 00000000
MUX_OUT4=MUX_OUT2= 00010000
REG4 = REG4 + MUX_OUT3+MUX_OUT4 = 11010000

Passo 3:
PK1=0, PK2=0
MUX_OUT1=00001000
MUX_OUT2=00000100
MUX_OUT3= 00000000
MUX_OUT4= 00000000
REG4 = REG4 + MUX_OUT3+MUX_OUT4 = 11010000

Passo 4:
PK1=0, PK2=1
MUX_OUT1=00000010
MUX_OUT2=00000001
MUX_OUT3= 00000000
MUX_OUT4=MUX_OUT2= 00000001
REG4 = REG4 + MUX_OUT3+MUX_OUT4 = 11010001

A questo punto REG4 contiene il primo byte da scrivere in memoria


start=0
start=1
start=0
start=1
S0
o_done=0
S1
chiedo num parole
+ reset indirizzi
o_addr_sel = 1
r6_load = 1
o_en = 1
o_we = 0
S2
carico num parole e reset
r1_load = 1
r5_load = 1
r3_reset =1
r3_load = 1
r4_reset = 1
r4_load = 1
S3
carico indirizzo
o_addr_sel = 0
r6_load = 1
S4
chiedo parola
o_en = 1
o_we = 0
S5
carico parola
r2_load = 1
S6
inizio convoluzione
d_sel = 0
ffd1_load = 1
ffd2_load = 1
uk_load = 1
pk_load = 1
S7
d_sel = 1
ffd1_load = 1
ffd2_load = 1
uk_load = 1
pk_load = 1
S8
d_sel = 2
r4_load = 1
ffd1_load = 1
ffd2_load = 1
uk_load = 1
pk_load = 1
S9
d_sel = 3
r4_load = 1
ffd1_load = 1
ffd2_load = 1
uk_load = 1
pk_load = 1
S10
d_sel = 4
r4_load = 1
ffd1_load = 1
ffd2_load = 1
uk_load = 1
pk_load = 1
S11
convoluzione + carico indirizzo
d_sel = 5
r4_load = 1
o_addr_sel = 2
r6_load = 1
ffd1_load = 1
ffd2_load = 1
uk_load = 1
pk_load = 1
S12
scrivo primo byte
o_en = 1
o_we = 1
S13
azzero r4 e incremento r3
r4_reset = 1
r4_load = 1
r3_load = 1
S14
riprendo convoluzione
d_sel = 6
r4_load = 1
ffd1_load = 1
ffd2_load = 1
uk_load = 1
pk_load = 1
S15
d_sel = 7
r4_load = 1
ffd1_load = 1
ffd2_load = 1
uk_load = 1
pk_load = 1
S16
d_sel = 0
r4_load = 1
pk_load = 1
S17
convoluzione
carico indirizzo
decremento num parole
d_sel = 1
r4_load = 1
pk_load = 1
o_addr_sel = 2
r6_load = 1
r1_sel = 1
r1_load = 1
S18
scrivo secondo byte
o_en = 1
o_we = 1
S19
incremento indirizzo
uscita + reset
o_addr_sel = 0
r6_load = 1
r4_reset = 1
r4_load = 1
r3_load = 1
S20
start=1
reset interno
comunico che ho finito
internal_reset = 1
o_done = 1
o_end = 1
o_end = 0

Macchina a Stati

La macchina a stati rappresenta il comportamento del componente e ha 20 stati.

- Alla partenza il componente viene resettato e inizia dallo stato S0, lì vi resta finché non riceve il segnale di start
- A quel punto legge il numero delle parole ed effettua alcune inizializzazioni (S1, S2)
- Successivamente carica il primo indirizzo e la prima parola (S3, S4, S5)
- A questo punto inizia la codifica (S6->S10)
- Una volta giunto a metà della parola (S11) carico in REG6 l'indirizzo in cui andare a scrivere
- Al ciclo successivo scrivo effettivamente in memoria (S12)
- Resetto REG4 e riprendo la codifica della seconda metà della parola (S13->S16)
- Carico in REG6 l'indirizzo in cui andare a scrivere (S17)
- Al ciclo successivo scrivo effettivamente in memoria (S18)
- A questo punto controllo se *o_end* è 0 o 1 (a seconda che il contenuto di REG1 sia 0 o meno)
    - Se *o_end* è 0 effettuo alcuni reset e imposto il nuovo indirizzo da cui leggere (S19) per poi ripartire dallo stato S4 (lettura della prossima parola)
    - Se *o_end* è 1 significa che ho terminato (S20). Mando il segnale di *o_done* e metto a 1 un segnale di reset interno che svolge la stessa funzione del reset. In questo modo potrò ricominciare l'esecuzione a fronte di un nuovo start senza dover attendere prima un segnale di reset dall'esterno. Resterò in questo stato fino a quando *o_start* non tornerà a 0.

Risultati sperimentali

Report Sintesi

- report_utilization

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	90	0	0	134600	0.07
LUT as Logic	90	0	0	134600	0.07
LUT as Memory	0	0	0	46200	0.00
Slice Registers	91	0	0	269200	0.03
Register as Flip Flop	91	0	0	269200	0.03
Register as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

- report_timing

```
Timing Report

Slack (MET) :             95.838ns  (required time - arrival time)
  Source:                 FSM_onehot_cur_state_reg[11]/C
                            (rising  edge-triggered  cell  FDCE  clocked  by  clock    {rise@0.000ns  fall@50.000ns
period=100.000ns})
  Destination:            DATAPATH0/o_reg4_reg[7]/D
                            (rising  edge-triggered  cell  FDCE  clocked  by  clock    {rise@0.000ns  fall@50.000ns
period=100.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        4.011ns  (logic 2.055ns (51.234%)  route 1.956ns (48.766%))
  Logic Levels:           5  (CARRY4=2 LUT4=2 LUT5=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.100ns = ( 102.100 - 100.000 )
    Source Clock Delay      (SCD):    2.424ns
    Clock Pessimism Removal (CPR):    0.178ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Simulazioni

- Test 1: Esempio 1 della specifica

```
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'));

...

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

INDIRIZZO	VALORE
0	2
1	162
2	75
[...]	
1000	209
1001	205
1002	247
1003	210

- Test 2 e 3: Esempi 2 e 3 della specifica (come il test 1, verifico solo che funzionino)

- Test 4: 0 parole in ingresso, verifico che il programma termini e non scriva niente in memoria

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

- Test 5: 255 parole in ingresso, verifico che il programma termini e che gli output siano corretti

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned(  255  , 8)),
                         1 => std_logic_vector(to_unsigned(  204  , 8)),
                         2 => std_logic_vector(to_unsigned(  204  , 8)),
                         3 => std_logic_vector(to_unsigned(  204  , 8)),
                         4 => std_logic_vector(to_unsigned(  204  , 8)),
                         5 => std_logic_vector(to_unsigned(  204  , 8)),
                         6 => std_logic_vector(to_unsigned(  204  , 8)),
                         7 => std_logic_vector(to_unsigned(  204  , 8)),

                         …
                         250 => std_logic_vector(to_unsigned(  204  , 8)),
                         251 => std_logic_vector(to_unsigned(  204  , 8)),
                         252 => std_logic_vector(to_unsigned(  204  , 8)),
                         253 => std_logic_vector(to_unsigned(  204  , 8)),
                         254 => std_logic_vector(to_unsigned(  204  , 8)),
                         255 => std_logic_vector(to_unsigned(  204  , 8)),
                         others => (others =>'0'));
...

assert RAM(1000) = std_logic_vector(to_unsigned( 235 , 8)) report "TEST FALLITO";
assert RAM(1001) = std_logic_vector(to_unsigned( 235 , 8)) report "TEST FALLITO";
assert RAM(1002) = std_logic_vector(to_unsigned( 235 , 8)) report "TEST FALLITO";
assert RAM(1003) = std_logic_vector(to_unsigned( 235 , 8)) report "TEST FALLITO";
...
assert RAM(1507) = std_logic_vector(to_unsigned( 235 , 8)) report "TEST FALLITO";
assert RAM(1508) = std_logic_vector(to_unsigned( 235 , 8)) report "TEST FALLITO";
assert RAM(1509) = std_logic_vector(to_unsigned( 235 , 8)) report "TEST FALLITO";
assert RAM(1510) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO";
```

- Test 6: Arriva un nuovo start dopo un segnale di reset senza modificare la RAM, verifico che l'output sia corretto anche dopo la nuova esecuzione

```
    tb_start <= '0'; wait for 100 ns; wait for c_CLOCK_PERIOD;
    tb_rst <= '1'; wait for c_CLOCK_PERIOD; wait for 100 ns;
    tb_rst <= '0'; wait for c_CLOCK_PERIOD; wait for 100 ns;
    tb_start <= '1'; wait for c_CLOCK_PERIOD;
    wait until tb_done = '1';
    wait for c_CLOCK_PERIOD;
    tb_start <= '0';
    wait until tb_done = '0';
    wait for 100 ns;
    tb_rst <= '1'; wait for c_CLOCK_PERIOD; wait for 100 ns;
    tb_rst <= '0'; wait for c_CLOCK_PERIOD; wait for 100 ns;
    tb_start <= '1'; wait for c_CLOCK_PERIOD;
    wait until tb_done = '1';
    wait for c_CLOCK_PERIOD;
    tb_start <= '0';
    wait until tb_done = '0';
    wait for 100 ns;
```

- Test 7: Arriva un nuovo start prima di un segnale di reset senza modificare la RAM, verifico che l'output sia corretto anche dopo la nuova esecuzione

```
tb_start <= '0'; wait for 100 ns; wait for c_CLOCK_PERIOD;
tb_rst <= '1'; wait for c_CLOCK_PERIOD; wait for 100 ns;
tb_rst <= '0'; wait for c_CLOCK_PERIOD; wait for 100 ns;
tb_start <= '1'; wait for c_CLOCK_PERIOD;
wait until tb_done = '1';
wait for c_CLOCK_PERIOD;
tb_start <= '0';
wait until tb_done = '0';
wait for 100 ns;
tb_start <= '1'; wait for c_CLOCK_PERIOD;
wait until tb_done = '1';
wait for c_CLOCK_PERIOD;
tb_start <= '0';
wait until tb_done = '0';
wait for 100 ns;
```

- Test 8: Arriva un nuovo start dopo un segnale di reset modificando la RAM tra la prima e la seconda esecuzione, verifico che l'output sia corretto.

- Test 9: Arriva un nuovo start prima un segnale di reset modificando la RAM tra la prima e la seconda esecuzione, verifico che l'output sia corretto.

- Test 10: Non rimetto start a 0 alla fine dell'esecuzione. Verifico che resti bloccato nell'ultimo stato.

Tutti i test da me effettuati riportati qui sopra hanno dato esito positivo.

Conclusioni

È stato realizzato un codificatore convoluzionale in liguaggio VHDL tramite la definizione dei componenti interni (Datapath) e del comportamento (Macchina a Stati).
Il codificatore è stato scritto in modo che potesse interagire con una memoria RAM da cui leggere e su cui scrivere i dati.
È stato in seguito testato attraverso dei test bench per verificarne il corretto funzionamento in diverse condizioni.
È infine stato sintetizzato con successo su un FPGA virtuale.