# Prova Finale di Reti Logiche

Leonardo Airoldi

A. A. 2021-22

# Indice

# Capitolo 1

# Introduzione

La specifica della Prova finale (Progetto di Reti Logiche) 2021 richiede di implementare un **codificatore convoluzionale** usando il linguaggio VHDL.
Un codificatore convoluzionale è un dispositivo che tramite l'applicazione di un *codice convoluzionale* codifica uno stream di bit in ingresso in uno stream di bit in uscita ridondante che permette a un dispositivo che legge lo stream codificato di risalire allo stream originale anche in presenza di errori applicando tecniche di *error-correction*. Codificatori convoluzionali elettronici sono molto comuni nell'ambito delle telecomunicazioni, in quanto conferiscono a una rete un certo grado di robustezza: un dispositivo ricevente può ricostruire il segnale trasmesso, che potrebbe risultare corrotto a causa del rumore sulla linea, senza richiedere un ulteriore invio dell'informazione.

## 1.1 Specifiche

Nel caso specifico di questo progetto, è richiesto lo sviluppo di un circuito integrato con le seguenti caratteristiche:

1. il *codice convoluzionale* ha **tasso di trasmissione** $\frac{1}{2}$.
2. si interfaccia con una memoria sincrona a word di 8bit e indirizzi a 16bit.
3. lo stream in ingresso è di dimensione $n$ words memorizzate a partire dall'indirizzo `0x01`.
4. la dimensione dello stream è memorizzata all'indirizzo `0x00` ed è al massimo di 255 word.
5. il dispositivo viene sempre inizializzato con un segnale di `reset`.
6. la computazione inizia al segnale di `start` e termina quando il dispositivo alza il segnale `done`. Il dispositivo deve essere in grado di gestire computazioni successive senza aver bisogno di un segnale di `reset`. Si presuppone che il segnale di `start` rimanga alto durante tutta la computazione e che non possa essere rialzato prima che il segnale di `done` sia stato riportato a `0`.
7. il dispositivo deve funzionare con un periodo di clock di almeno `100ns`

## 1.2 Strumenti

Il progetto sarà sviluppato in **VHDL** (VHSIC Hardware Description Language) che permette di descrivere circuiti integrati. Per la simulazione e la sintesi della scheda il software utilizzato è **Xilinx Vivado** (*v2021.1*). Il dispositivo sarà poi implementato su una **FPGA Artix 7** `xc7a200tfbg484-1`.

## 1.3 Esempio

# Capitolo 2

# Architettura

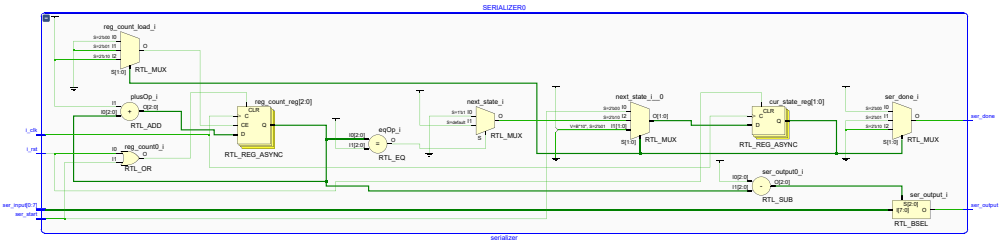
Il dispositivo è stato implementato come una macchina a stati che controlla un datapath, che si interfaccia con diversi componenti più piccoli. La scelta di dividere il progetto moduli più piccoli è stata presa per non rendere troppo complessa e articolata la gestione del dispositivo nella sua interezza. In questo modo la macchina a stati si limita a gestire la comunicazione e coordinazione dei vari componenti, che eseguono ciascuno una funzione definita anch'essa da una macchina a stati interna.

## 2.1 Serializzatore

Il **serializzatore** è il componente addetto alla serializzazione in un flusso continuo di bit dei byte caricati dalla ram. Il serializzatore si interfaccia, oltre al segnale di *clock* e al segnale di *reset*, con il segnale di ingresso da serializzare `[input:8bit]` e dal segnale di controllo *start*. Produce in uscita un segnale di `[output:1bit]` che corrisponde al flusso serializzato del byte in input e un segnale di *done* che indica la fine della serializzazione.

Al segnale di `start` della serializzazione il componente passa dallo stato `STOPPED` allo stato attivo. Quando il componente è attivo produce in uscita per **due clicli di clock** l'*n-esimo* bit del segnale in ingresso, incrementando poi il contatore per produrre in uscita l'*n+1 esimo* bit del byte in ingresso nei prossimi due cicli di clock. La serializzazione termina quando il contatore interno raggiunge il valore `111`: il registro viene incrementato tornando quindi a `0` ma il serializzatore torna nello stato `STOPPED`, che alza il segnale di `done` a `1`.

Il flusso in uscita ha periodo di due cicli di clock in quanto al convolutore server produrre due bit sequenzialmente in uscita per ogni bit in entrata. In questo modo la velocità di serializzazione è ridotta della metà ma il convolutore riesce a produrro il flusso risultante sullo stesso ciclo di clock della macchina.




## 2.2 Convolutore

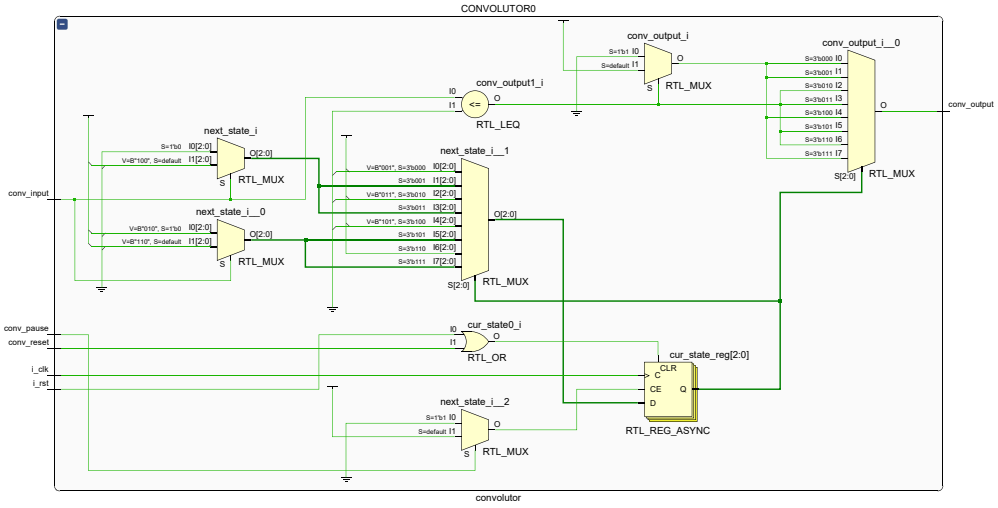
Il **convolutore** è il componente che emula il comportamento della macchina a stati del convolutore convoluzionale. E' stato realizzato seguendo il modello ad *automa a stati finiti (**FSM**)*. Il componente si interfaccia (oltre al segnale di *clock* e al segnale di *reset*) a un segnale di reset interno e a un segnale di *pause*. Questo permette al convolutore di fermare la propria macchina a stati ignorando l'*input* ma mantenendo comunque lo stato corrente della macchina. Questo stato di pausa viene utilizzato per attendere i caricamenti da e verso la memoria RAM che richiedono qualche ciclo di clock per caricare/scaricare i registri nella memoria e viceversa.

L'automa produce due bit in serie per ogni bit ricevuto in ingresso: per questo è fonadmentale che il serializzatore mantenga per due cicli di clock il bit in ingresso. L'automa quindi possiede due stati per ogni macro-stato del convolutore convoluzionale, che corrispondono allo stato in cui l'uscita è settata a pk1 e l'uscita è settata a pk2.

La scelta di implementare il componente in questo modo è stata presa per rispettare le specifiche di progetto:

> *"Il flusso Y è ottenuto come concatenamento alternato dei due bit di uscita"*

Avendo un convolutore implementato come **entity separata a sè stante** è possibile utilizzare il componente come da specifica anche in altri progetti. Se per esempio si fosse scelto di gestire l'output del convolutore come una coppia di segnali pk1 e pk2 si sarebbe ridotto il tempo di computazione ma la specifica riguardante il flusso di uscita di dimensione di 1bit sarebbe stata violata.

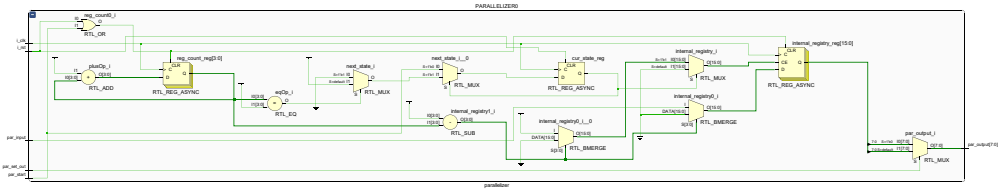



## 2.3 Parallelizzatore

Il **parallelizzatore** è il componente addetto alla raccolta del flusso in uscita dal convolutore e alla sua memorizzazione in un registro interno. Questo permette di *impacchettare* il flusso d'uscita in byte in modo da essere scritti sulla memoria ram. Dato che per questa configurazione del convolutore a **ogni byte in ingresso** corrispondono **2 byte in uscita**, questo componente è stato progettato con un registro interno a **16 bit** in modo da poter memorizzare i 2 byte in uscita dall'elaborazione di un singolo byte e gestire poi il caricamento delle 2 parole sulla RAM.

Il parallelizzatore si interfaccia con un ingresso `[input:1bit]` e produce un'uscita `[output:8bit]`, selezionabile tramite un **multiplexer**, che seleziona sull'uscita i bit `[0-7]` e `[8-15]` del registro interno.

Al segnale di `start` il componente passa dallo stato `STOPPED` allo stato `ACTIVE`. Quando il componente è attivo memorizza il bit di **ingresso** nell'*n-esimo* bit del registro interno, incrementando poi il contatore per il prossimo ciclo di clock. La parallelizzazione termina quando il contatore interno raggiunge il valore `1111`: il registro viene incrementato tornando quindi a `0` e il parallelizzatore torna nello stato `STOPPED`, tenendo però comunque memorizzati i valori nel registro. A questo punto la macchina a stati entra negli stati di *caricamento in RAM*, dove carica nel registro di uscita in ordine il primo byte e il secondo nel registro di uscita, da cui poi il dato vene caricato sulla memoria.



## 2.4 Datapath

Il **datapath** è l'entità che **fornisce ai vari moduli un'interfaccia** per comunicare tra loro o con la **memoria RAM**. Per permettere le comunicazioni tra i vari moduli del dispositivo, istanziati nella macchina a stati (*project_ reti_ logiche*), al datapath sono passati i segnali rappresentanti *dati* (mentre non sono passati quelli di controllo, di responsabilità della macchina a stati) dei vari componenti, che poi vengono connessi all'interno del datapath.

Il datapath contiene inoltre i componenti per interfacciarsi con la RAM e dei registri di supporto per la FSM.

**Registri in out.** Nella fase di lettura dalla memoria ram il dato viene letto e caricato in un registro *reg_ in*. Nella fase di scrittura invece il dato sarà prelevato dal registro *reg_ out*. Entrambi i registri sono da 8bit e sono connessi rispettivamente al serializzatore e al parallelizzatore. Il loro caricamento viene gestito dalla FSM.

**Registro words.** Nella fase di avvio del dispositivo viene letto all'indirizzo 0 della memoria il numero di byte da cui è formato lo stream di ingresso. Questo valore è memorizzato all'interno del registro *reg_ words*, di 8bit in quanto la dimensione massima dello stream in ingresso è di 255 parole. Il suo caricamento è gestito dalla FSM.

**Registro count.** Durante l'esecuzione è necessario memorizzare lo stato di avanzamento in modo da sapere a quale indirizzo della RAM accedere per leggere o scrivere dati. Questo resgistro memorizza al suo interno *l'indirizzo* della cella di memoria di *read* a cui si sta puntando. Per esempio se si sta leggendo la terza parola dello stream nel registro sarà presente il valore `0x0003` in quanto lo stream è memorizzato a partire dall'indirizzo `0x0001`. E' un registro a 16bit e il suo caricamento e incremento è gestito dalla FSM.

**Done comparator.** Per rilevare la fine dell'elaborazione nel datapath è presente un comparatore. Esso effettua un controllo che porta un segnale di *done* interno a 1 se il valore contenuto

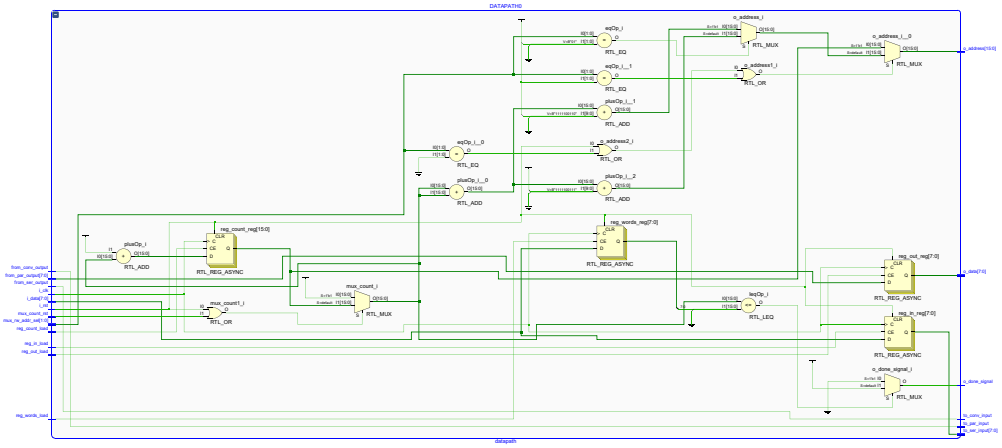
all'interno del registro di *count* supera il numero di parole. Questo sfrutta il fatto che il numero della parola corrente corrisponde al contenuto del registro *count*, in quando l'indirizzo della parola n-esima è n. Inoltre, se dovessimo essere nel caso in cui la lunghezza dello stream fosse 255, il confronto funzionerebbe ancora perchè fino a 255 il processo continuerebbe regolarmente mentre al raggiungimento di 256, valore possibile nel registro degli indirizzi a 16bit *count*, il confronto finirebbe correttamente per segnalare la fine dell'esecuzione.

```
if mux_count <= ("00000000" & reg_words) then
          o_done_signal <= '0';
      else
          o_done_signal <= '1';
```

**Multiplexer read write.** Il registro *count* non è direttamente collegato con il segnale *address* della memoria RAM. Questo perchè in scrittura devono essere utilizzati gli indirizzi contigui partendo dall'indirizzo 1000. Questo multiplexer permette di selezionare quale indirizzo viene passato al segnale di indirizzo della memoria:

```
if i_rst = '1' or mux_rw_addr_sel = "00" or mux_rw_addr_sel = "11" then
          o_address <= reg_count;
      elsif mux_rw_addr_sel = "01" then
          o_address <= (mux_count + mux_count) + "0000001111100110";
      else --if mux_rw_addr_sel = "10" then
          o_address <= (mux_count + mux_count) + "0000001111100111";
      end if;
```

Il mux seleziona di default il contenuto del registro *count* usato in lettura dello stream. Per la scrittura dei due byte corrispondenti all'elaborazione dell'n-esima parola, viene selezionato l'indirizzo $2n + 998$ per il primo byte e $2n + 999$ per il secondo. Questo permette di scrivere nell'indirizzo giusto i due byte derivati dall'elaborazione dell'n-esimo byte. Per esempio, la convoluzione terza parola verrà scritta negli indirizzi `1004 1005`, in quanto la prima parola è scritta all'interno di `1000 1001` e la seconda in `1002 1003`, come da specifica.



## 2.5 Macchina a stati

Il progetto è tutto contenuto all'interno della entity Vivado **project_reti_logiche**. Questa contiene al suo interno tutti i **components** del dispositivo (serializer, convolutor, parallelizer e datapath) e implementa tramite tre **process** (*FSM_STATE_CHANGE, FSM_FLOW, FSM_OUT*). Questa FSM permette di controllare il flusso d'esecuzione di tutto il dispositivo portando i segnali di controllo ai valori corretti in base allo stato in cui ci si trova.

### 2.5.1 Elenco delle transizioni

Segue un elenco delle transizioni. Vedi figura 2.1

```
RESET                   => i_start            ? READ_WORD_RAM_REQUEST : RESET
READ_WORDS_RAM_REQUEST  => READ_WORDS_RAM_READ
READ_WORDS_RAM_READ     => READ_RAM_REQUEST
READ_RAM_REQUEST        => o_done_signal      ? DONE : READ_RAM
READ_RAM                => SERIALIZE
SERIALIZE               => ser_done           ? LOAD_FROM_PAR : SERIALIZE
LOAD_FROM_PAR           => WRITE_RAM_1
WRITE_RAM_1             => WRITE_RAM_2
WRITE_RAM_2             => WRITE_RAM_WAIT
WRITE_RAM_WAIT          => READ_RAM_REQUEST
DONE                    => i_start            ? DONE : RESET
```

### 2.5.2 Descrizione degli stati

La macchina a stati è modellata come una macchina di Moore. I segnali di controllo sono dipendenti solamente dallo stato corrente della macchina. In seguito sono elencate le uscite dei segnali di controllo per ogni stato. Di default tutti gli stati sono portati al livello logico basso.

#### RESET

Lo stato di reset corrisponde allo stato iniziale della macchina. La macchina torna in questo stato alla fine di una computazione quando è stato riabbassato il segnale di start o quando il segnale di reset è portato a 1.

In questo stato si azzera il registro count degli indirizzi tramite un multiplexer e si azzera lo stato del convolutore, riportandolo nello stato iniziale in modo da poter iniziare una elaborazione sempre con il convolutore azzerato anche in caso di multiple elaborazioni in sequenza.

Si esce dallo stato di reset non appena una nuova elaborazione è richiesta tramite il segnale di start.

| Signal | Value |
|---|---|
| reg_in_load | 0 |
| reg_out_load | 0 |
| reg_words_load | 0 |
| **reg_count_load** | **1** |
| **mux_count_rst** | **1** |
| mux_rw_addr_sel | 00 |
| ser_start | 0 |
| **conv_reset** | **1** |
| par_set_out | 0 |
| o_en | 0 |
| o_we | 0 |
| o_done | 0 |

Tabella 2.1: Uscite nello stato RESET.

#### READ_WORDS_RAM_REQUEST

Nello stato di READ_WORDS_RAM_REQUEST si carica all'interno del registro words il numero di byte corrispondente alla lunghezza dello stream in input. Questo valore viene richiesto alla RAM in questo stato.

| Signal | Value |
|---|---|
| reg_in_load | 0 |
| reg_out_load | 0 |
| reg_words_load | 0 |
| reg_count_load | 0 |
| mux_count_rst | 0 |
| **mux_rw_addr_sel** | **00** |
| ser_start | 0 |
| conv_reset | 0 |
| par_set_out | 0 |
| **o_en** | **1** |
| o_we | 0 |
| o_done | 0 |

Tabella 2.2: Uscite nello stato READ_WORDS_RAM_REQUEST.

### READ_WORDS_RAM_READ

Essendo la memoria sincrona, essa rilasia il dato richiesto al ciclo di clock precedente solamente durante quello successivo. In questo stato, raggiunto nel ciclo di clock immediatamente successivo al ciclo dello stato READ_WORDS_RAM_REQUEST, si carica il valore recuperato dalla RAM all'interno del registro *words*.

| Signal | Value |
|---|---|
| reg_in_load | 0 |
| reg_out_load | 0 |
| **reg_words_load** | **1** |
| **reg_count_load** | **1** |
| mux_count_rst | 0 |
| mux_rw_addr_sel | 00 |
| ser_start | 0 |
| conv_reset | 0 |
| par_set_out | 0 |
| o_en | 0 |
| o_we | 0 |
| o_done | 0 |

Tabella 2.3: Uscite nello stato READ_WORDS_RAM_READ.

### READ_RAM_REQUEST

READ_RAM_REQUEST è lo stato che richiede alla memoria RAM il byte dello stream da leggere per proseguire con l'elaborazione. Per questo utilizza il mux di selettore indirizzo a 00 in modo da andare a leggere dall'indirizzo di lettura già contenuto nel registro di count.

Oltre all'interrogaizone della RAM lo stato effettua una funzione di controllo: se dal datapth arriva il segnale *o_done_signal* il prossimo stato non sarà quello di caricamento nel registro in della parola letta dalla memoria ma lo stato di DONE, dove viene notificata la fine dell'elaborazione.

| Signal | Value |
|---|---|
| reg_in_load | 0 |
| reg_out_load | 0 |
| reg_words_load | 0 |
| reg_count_load | 0 |
| mux_count_rst | 0 |
| **mux_rw_addr_sel** | **00** |
| ser_start | 0 |
| conv_reset | 0 |
| par_set_out | 0 |
| **o_en** | **1** |
| o_we | 0 |
| o_done | 0 |

Tabella 2.4: Uscite nello stato READ_RAM_REQUEST.

### READ_RAM

In questo stato il byte dello stream richiesto al ciclo precedente viene letto e caricato all'interno del registro *reg_in*. Inoltre è alzato il segnale di start del serializzatore in modo da far partire l'elaborazione nel ciclo successivo.

| Signal | Value |
|---|---|
| **reg_in_load** | **1** |
| reg_out_load | 0 |
| reg_words_load | 0 |
| reg_count_load | 0 |
| mux_count_rst | 0 |
| mux_rw_addr_sel | 00 |
| **ser_start** | **1** |
| conv_reset | 0 |
| par_set_out | 0 |
| o_en | 0 |
| o_we | 0 |
| o_done | 0 |

Tabella 2.5: Uscite nello stato READ_RAM.

### SERIALIZE

Nello stato serialize avviene tutto il calcolo della convoluzione del byte presente all'interno del registro *reg_in*. Il serializzatore continua finchè non ha serializzato tutto il registro *reg_in*, quando alzerà il segnale *ser_done*, che mette in pausa il convolutore. Il parallelizzatore, partito anch'esso con il segnale di *ser_start* sarà valido alla fine della convoluzione del byte. Quando il segnale di *ser_done* è alzato dopo 16 cicli di clock si passa alla sezione di caricamento in RAM.

| Signal | Value |
|---|---|
| reg_in_load | 0 |
| reg_out_load | 0 |
| reg_words_load | 0 |
| reg_count_load | 0 |
| mux_count_rst | 0 |
| mux_rw_addr_sel | 00 |
| ser_start | 0 |
| conv_reset | 0 |
| par_set_out | 0 |
| o_en | 0 |
| o_we | 0 |
| o_done | 0 |

Tabella 2.6: Uscite nello stato SERIALIZE.

### LOAD_FROM_PAR

In questo stato si carica dal registro interno del parallelizzatore al registro di uscita *reg_out* il primo byte da scrivere in memoria. Il byte corretto viene selezionato mandando al parallelizzatore il bit di comando del suo mux di uscita. Il primo byte viene selezionato con il segnale 0.

| Signal | Value |
|---|---|
| reg_in_load | 0 |
| **reg_out_load** | **1** |
| reg_words_load | 0 |
| reg_count_load | 0 |
| mux_count_rst | 0 |
| mux_rw_addr_sel | 00 |
| ser_start | 0 |
| conv_reset | 0 |
| **par_set_out** | **0** |
| o_en | 0 |
| o_we | 0 |
| o_done | 0 |

Tabella 2.7: Uscite nello stato LOAD_FROM_PAR.

### WRITE_RAM_1

Nel primo stato di scrittura viene richiesta alla RAM la scrittura del primo byte, presente già nel registro di uscita. Questo viene scrito all'indirizzo selezionato con il mux di indirizzo a 01. Inoltre sempre durante questo stato viene richiesto il caricamento del secondo byte in ingresso nel registro di out, in modo da averlo già caricato per il successivo ciclo di clock e richiedere già la sua scrittura.

| Signal | Value |
|---|---|
| reg_in_load | 0 |
| **reg_out_load** | **1** |
| reg_words_load | 0 |
| reg_count_load | 0 |
| mux_count_rst | 0 |
| **mux_rw_addr_sel** | **01** |
| ser_start | 0 |
| conv_reset | 0 |
| **par_set_out** | **1** |
| **o_en** | **1** |
| **o_we** | **1** |
| o_done | 0 |

Tabella 2.8: Uscite nello stato WRITE_RAM_1.

### WRITE_RAM_2

In questo stato, come nel precedente, viene richiesta la scrittura del registro di output, che in questo ciclo di clock avrà caricato il secondo byte presente nel parallelizzatore. Per il secondo byte viene scelto l'indirizzo selezionato 10 dal mux degli indirizzi.

| Signal | Value |
|---|---|
| reg_in_load | 0 |
| reg_out_load | 0 |
| reg_words_load | 0 |
| reg_count_load | 0 |
| mux_count_rst | 0 |
| **mux_rw_addr_sel** | **10** |
| ser_start | 0 |
| conv_reset | 0 |
| par_set_out | 0 |
| **o_en** | **1** |
| **o_we** | **1** |
| o_done | 0 |

Tabella 2.9: Uscite nello stato WRITE_RAM_2.

### WRITE_RAM_WAIT

Durante questo ciclo di clock la RAM sta completando il caricamento del secondo byte dello stream di uscita. Con questo stato di attesa si aspetta in modo da avere una RAM valida già in uscita da questo stato. Inoltre si incrementa il counter degli indirizzi in modo che il prossimo stato (READ_RAM_REQUEST) abbia il contatore aggiornato e possa svolgere la verifica dello stato dell'elaborazione.

| Signal | Value |
|---|---|
| reg_in_load | 0 |
| reg_out_load | 0 |
| reg_words_load | 0 |
| **reg_count_load** | **1** |
| mux_count_rst | 0 |
| mux_rw_addr_sel | 00 |
| ser_start | 0 |
| conv_reset | 0 |
| par_set_out | 0 |
| o_en | 0 |
| o_we | 0 |
| o_done | 0 |

Tabella 2.10: Uscite nello stato WRITE_RAM_WAIT.

**DONE**

Questo è lo stato di fine elaborazione. Esso porta il segnale *o_done* a alto. Si resta in questo stato fintantochè il segnale di start non viene riabbassato dall'utente. A quel punto la macchina tornerà nello stati iniziale (RESET) abbassando il segnale di done e rendendolo prondo per una nuova elaborazione, come da specifica.

| Signal | Value |
|---|---|
| reg_in_load | 0 |
| reg_out_load | 0 |
| reg_words_load | 0 |
| reg_count_load | 0 |
| mux_count_rst | 0 |
| mux_rw_addr_sel | 00 |
| ser_start | 0 |
| conv_reset | 0 |
| par_set_out | 0 |
| o_en | 0 |
| o_we | 0 |
| **o_done** | **1** |

Tabella 2.11: Uscite nello stato DONE.

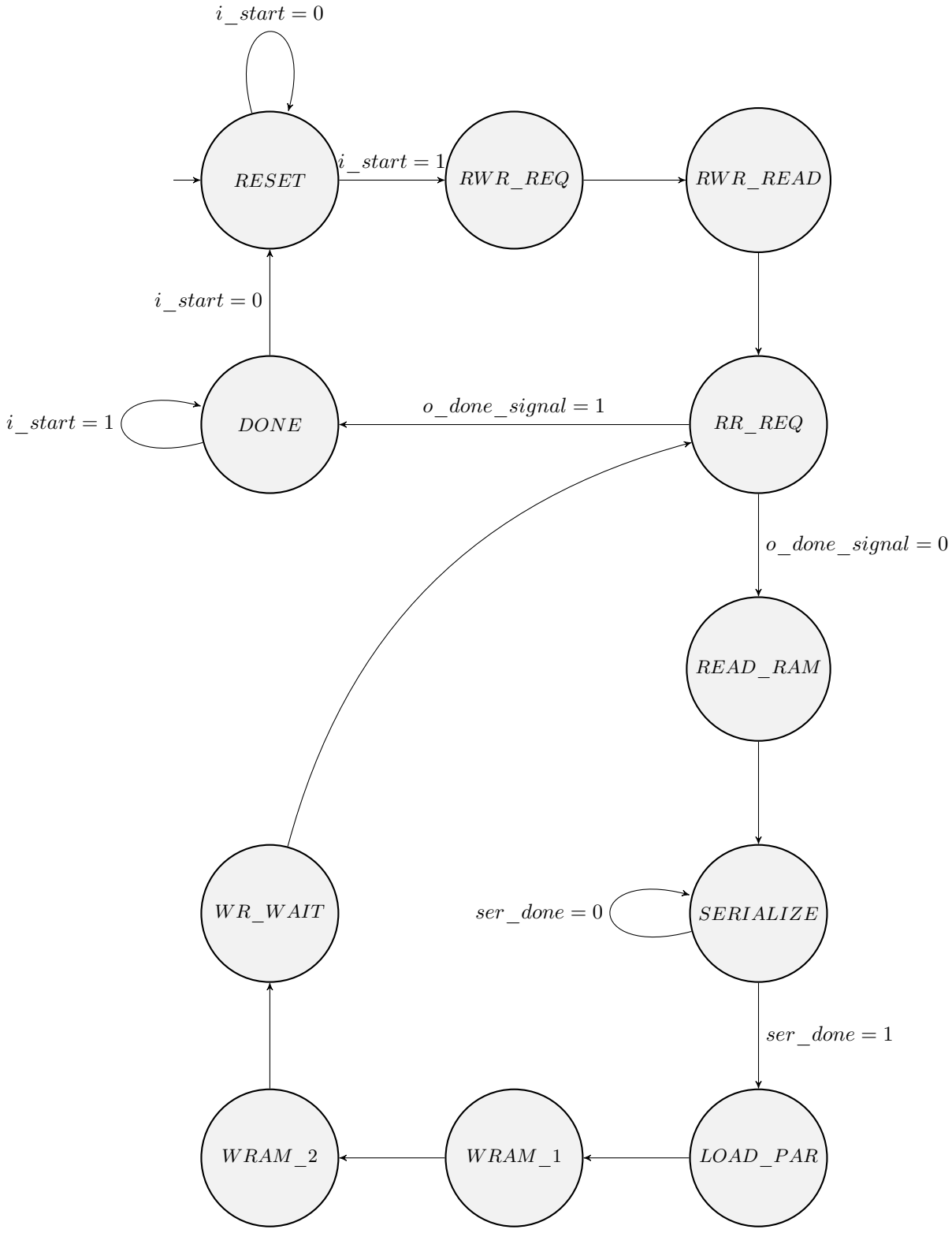



Figura 2.1: FSM Flow

# Capitolo 3

# Risultati sperimentali

Il dispositivo progettato è stato sintetizzato e testato con il tool Vivado di Xilinx. Vengono riportate ora le analisi sugli aspetti di sintesi del dispositivo e sui test effettuati su di esso.

## 3.1 Sintesi

La sintesi è eseguita con un constraint sul periodo di clock di **100ns** e con una fpga target **FPGA Artix 7** `xc7a200tfbg484-1`.

### 3.1.1 Componenti

Il report di sintesi riassume l'elenco dei componenti della FPGA utilizzati per sintetizzare il dispositivo.

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice LUTs* | 109 | 0 | 0 | 134600 | 0.08 |
| LUT as Logic | 109 | 0 | 0 | 134600 | 0.08 |
| LUT as Memory | 0 | 0 | 0 | 46200 | 0.00 |
| Slice Registers | 81 | 0 | 0 | 269200 | 0.03 |
| Register as Flip Flop | 81 | 0 | 0 | 269200 | 0.03 |
| Register as Latch | 0 | 0 | 0 | 269200 | 0.00 |
| F7 Muxes | 1 | 0 | 0 | 67300 | <0.01 |
| F8 Muxes | 0 | 0 | 0 | 33650 | 0.00 |

Si può notare come tutti i registri utilizzati siano **sincroni** (flip flop).

### 3.1.2 Timing

Il report di timing riassume i parametri della scheda in relazione al segnale di clock

```
Slack (MET) :             95.783ns  (required time - arrival time)
  Source:                 DATAPATH0/reg_count_reg[3]/C
                            {rise@0.000ns fall@50.000ns period=100.000ns}
```

```
Destination:            FSM_onehot_cur_state_reg[4]/D
                          {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:        4.066ns  (logic 1.496ns (36.793%)  route 2.570ns (63.207%))
Logic Levels:           4  (CARRY4=2 LUT4=1 LUT6=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
```

Si può notare come lo **slack** (parametro che indica il tempo tra il delay massimo della logica combinatoria e il prossimo rising-edge del ciclo di clock) sia particolarmente elevato. Questo a grazie al delay della logica combinatoria (**4.066ns**) che risulta più di un ordine di grandezza inferiore al periodo del ciclo di clock (100ns). Questo dato indica quindi che la scheda sarebbe in grado di girare a 10 volte la velocità attuale, ovvero con un ciclo di clock di periodo 10ns (100Mhz).

# Capitolo 4

# Conclusione