Politecnico di Torino

DIPARTIMENTO DI ELETTRONICA E TELECOMUNICAZIONI
Corso di Laurea Magistrale in Ingegneria Elettronica

Sistemi Digitali Integrati
LABORATORIO 2

Progettazione di uno slave SPI

Prof. Massimo Ruo Roch

Laboratorio LED3

Author:
Bricco Letizia (s328719)

A.A. 2023/2024

Indice

Sommario

L'obiettivo del progetto è sviluppare e testare uno slave SPI in grado di effettuare transazioni di lettura e scrittura arbitrarie su un'opportuna interfaccia a registri.

Il circuito è stato realizzato rispettando rigorose specifiche di progetto per quanto riguarda sia il parallelismo dei segnali sia il regime di *timing* del sistema.

Inoltre, la *control unit* è stata progettata come una macchina a stati di Moore: in tal modo, il timing dei controlli risulta deterministico in quanto dipende unicamente dal clock di sistema.

La descrizione dell'architettura è stata implementata in VHDL in maniera gerarchica e il corretto comportamento è stato verificato sia mediante *testbench* sia su piattaforma fisica.

La simulazione è stata condotta utilizzando l'ambiente di sviluppo basato su Quartus-ModelSim e, per testare il blocco in maniera il più possibile completa, è stata automatizzata mediante uno script in linguaggio C++.

Per il test fisico, invece, ci si è avvalsi della scheda VirtLAB e, in particolare, il microcontrollore master STM32L496 è stato utilizzato come master SPI per la trasmissione e la ricezione dei dati.

1 Introduzione

Il protocollo **SPI** (***Serial Peripheral Interface***) è un protocollo di interfaccia seriale sincrona *full-duplex* proposto dalla Motorola come standard di comunicazione tra un microcontrollore e gli altri componenti di un circuito integrato.

Pur non essendo standardizzato, il suo utilizzo è molto frequente: ad esempio, una nota variante del protocollo SPI è il bus Microwire™ sviluppato dalla National Semiconductor.

La trasmissione avviene tra un dispositivo detto *master*, tipicamente il MCU, e uno o più chip periferici detti *slave*: il master controlla il bus, genera il segnale di clock, decide quando iniziare e terminare la comunicazione e con quale slave interagire; viceversa, lo slave è operativo solo quando selezionato dal master e si occupa di ricevere o trasmettere dati sul bus a seconda del comando ad esso impartito.

In ogni istante, il master può comunicare con uno solo degli slave, ma non ci sono limiti sulla lunghezza del messaggio da trasmettere, che è determinata unicamente dallo standard di comunicazione adottato dal costruttore.

Il protocollo si basa su **quattro segnali**, il cui nome può variare a seconda del costruttore; vengono qui utilizzati i nomi delle connessioni dell'IP sviluppata durante l'esperienza di laboratorio:

- **SCK** (***S**erial **C**lock*): generato dal master, è il segnale utilizzato per rendere sincrono il trasferimento dati.
- **nSS** (***S**lave **S**elect*): generato dal master per scegliere con quale slave vuole comunicare; si tratta di un segnale *attivo basso*, i.e., viene asserito quando subisce una transizione dal livello logico '1' al livello logico '0'; per limitare i consumi, tipicamente si fa in modo che SCK sia attivo solo quando nSS è asserito.
- **MOSI** (***M**aster **O**ut **S**lave **I**n*): è un'uscita per il master e un ingresso per lo slave, viene usato per la trasmisione di informazioni da master a slave.
- **MISO** (***M**aster **I**n **S**lave **O**ut*): è un ingresso per il master e un'uscita per lo slave, viene usato per la trasmissione di inforamzioni da slave a master.

Per quanto riguarda la frequenza di scambio dei dati, vi è un limite superiore che dipende dalle caratteristiche dei singoli dispositivi connessi sulla linea e dal loro numero: infatti, ogni slave aggiuntivo introduce una capacità parassita che aumenta il tempo di propagazione dei segnali.

La massima frequenza di funzionamento si ottiene facilmente dalle equazioni di timing:

$$\begin{cases} \dfrac{T_{\text{sck}}}{2} \geq t_{\text{out}}^{\text{m}} + t_{\text{p}}^{\text{bus}} + t_{\text{s}} \\ \dfrac{T_{\text{sck}}}{2} \geq t_{\text{out}}^{\text{s}} + t_{\text{p}}^{\text{bus}} + t_{\text{s}} \end{cases} \tag{1.1}$$

dove

1. $T_{\text{sck}} = 1/f_{\text{sck}}$ è la frequenza del SCK;
2. $t_{\text{out}}^{\text{m}}$ e $t_{\text{out}}^{\text{s}}$ sono i ritardi del master output e dello slave output rispetto al clock;
3. $t_{\text{p}}^{\text{bus}}$ è il tempo di propagazione lungo la linea, che dipende linearmente dalla capacità parassita;
4. t_{s} è il tempo di setup prima del campionamento.

2 Descrizione generale e funzionale

Questo paragrafo è dedicato ad una breve descrizione dei vincoli di progetto, della topologia del circuito e della derivazione dell'unità di controllo a partire dal timing del sistema.

2.1 Specifiche

Il protocollo SPI per la trasmissione dati può essere di quattro categorie, definite dal valore di due parametri di temporizzazione:

- **CPOL** (***C**lock **Pol**arity*): stabilisce il livello logico del serial clock negli istanti in cui nSS non è asserito;
- **CPHA** (***C**lock **Pha**se*): stabilisce se il campionamento e/o lo shift dei dati avvengono sul fronte di salita (0) o di discesa (1) del serial clock.

Il protocollo adottato per la progettazione della IP è quello convenzionalmente denotato come SPI_MODE_1, corrispondente ai valori CPOL $= 0$ (*clock polarity*) e CPHA $= 1$ (*clock phase*). Questo significa che:

1. Quando nSS non è asserito, la linea di SCK è posta al livello logico '0';
2. Master e slave campionano i dati in corrispondenza dei fronti di discesa di SCK;
3. Master e slave cambiano le proprie uscite in corrispondenza dei fronti di salita di SCK.

Inoltre, sono stati imposti dei vincoli sulla frequenza del clock di sistema CK e del serial clock SCK: si è assunto che la frequenza di CK fosse pari a $f_{\text{ck}} = 10\,\text{MHz}$ e si è assunto che il bus SPI avesse una frequenza di SCK pari a $f_{\text{sck}} \leq 1\,\text{MHz}$.

2.1.1 Protocollo di scrittura

Il protocollo adottato per effettuare una transazione di scrittura è il seguente:

1. Il master seleziona la periferica asserendo lo slave select nSS;

2. Sulla linea MOSI viene inviata una prima parola di 8 bit, che rappresenta il comando da eseguire (CMD), ed assume il valore

$$\text{CMD} = (32)_{10} = (20)_{16} = (00100000)_2 \,. \tag{2.1}$$

3. Subito dopo l'invio del comando, sulla linea MOSI viene inviata una seconda parola di 8 bit, che identificano l'indirizzo del registro in cui verrà memorizzato il dato (A);

4. Successivamente, sulla linea MOSI viene inviata un'ultima parola, questa volta codificata su 16 bit, che rappresenta il dato DIN da scrivere nel registro corrispondente all'indirizzo A;

5. Infine, la periferica viene deselezionata mediante il deasserimento di nSS;

6. Durante l'intera transazione, la linea MISO si trova in condizione di alta impedenza 'Z';

7. Il comando CMD, l'indirizzo A e il dato DIN vengono inviati serialmente sulla linea MOSI come un'unica parola di 32 bit e, oltre allo slave select, non sono previsti altri segnali per individuare gli istanti iniziale e finale della transazione.

Il protocollo adottato impone importanti vincoli sulla struttura delle interfacce a registri con cui lo slave SPI può interagire: dal momento che A è codificato su 8 bit, la periferica può inviare dati soltanto a $2^8 = 256$ locazioni differenti; inoltre, la dimensione massima delle parole che si possono memorizzare è 16 bit, il che limita la capacità massima della struttura di memoria a

$$(256 \times 16) \text{ bit} = 4 \,\text{kbit}. \tag{2.2}$$

Per una descrizione più dettagliata delle connessioni di I/O e della struttura della memoria, si rimanda al paragrafo 3.

2.1.2 Protocollo di lettura

Il protocollo adottato per effettuare una transazione di lettura è il seguente:

1. Il master seleziona la periferica asserendo lo slave select nSS;

2. Sulla linea MOSI viene inviata una prima parola di 8 bit, che rappresenta il comando da eseguire (CMD), ed assume il valore

$$\text{CMD} = (33)_{10} = (33)_{16} = (00100001)_2 \,. \tag{2.3}$$

3. Subito dopo l'invio del comando, sulla linea MOSI viene inviata una seconda parola di 8 bit, che identificano l'indirizzo del registro in cui verrà memorizzato il dato (A);

4. A questo punto, la linea MISO esce dalla condizione di alta impedenza e il MISO invia su di essa la parola da 16 bit contenuta nel registro corrispondente all'indirizzo A;

5. Infine, la periferica viene deselezionata mediante il deasserimento di nSS.

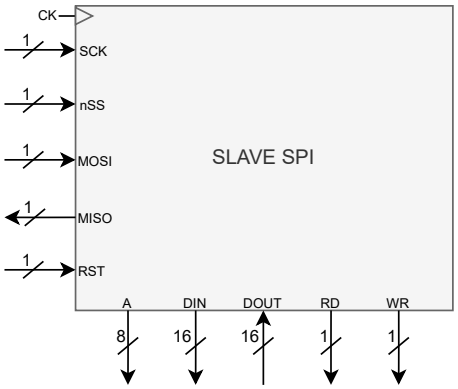


Figura 2.1: Slave SPI

2.2 Connessioni I/O

Lo schema delle porte di ingresso e di uscita della IP progettata, con i relativi parallelismi, è riportato in Figura 2.1. I segnali utilizzati sono i seguenti:

- **CK** (porta di ingresso, parallelismo 1 bit): clock di sistema con frequenza pari a 10 MHz, che fornisce la temporizzazione all'intero circuito; gli ingressi di tutti gli elementi sequenziali presenti nel circuito sono *positive edge triggered*, i.e., sono sensibili al fronte di salita di CK;
- **SCK** (porta di ingresso, parallelismo 1 bit): serial clock dell'interfaccia SPI con frequenza massima pari a 1 MHz. Come già detto, il protocollo SPI adottato impone che il campionamento dei dati avvenga sui fronti di discesa e che le uscite cambino sui fronti di salita;
- **nSS** (porta di ingresso, parallelismo 1 bit): slave select, i.e., bit di selezione della periferica che stabilisce gli istanti iniziale e finale di una transazione di scrittura o lettura;
- **MOSI** (porta di ingresso, parallelismo 1 bit): linea dati su cui vengono trasmessi serialmente i dati che il master invia allo slave; per evitare di realizzare una rete resistiva di *pull up*, non è prevista la possibilità che la linea vada in alta impedenza;
- **MISO** (porta di uscita, parallelismo 1 bit): linea dati su cui vengono trasmessi serialmente i dati che lo slave invia al master; essendo presente più di uno slave, è necessario che la linea esca dalla condizione di alta impedenza solo quando la periferica è selezionata e si sta effettuando un'operazione di lettura;
- **RST** (porta di ingresso, parallelismo 1 bit): segnale di reset asincrono che effettua il reset dello slave SPI, cancellando i dati che, attualmente, sono in fase di trasmissione o ricezione;
- **RD** (porta di uscita, parallelismo 1 bit): impulso di un colpo di CK che indica all'interfaccia a registri la necessità di effettuare una lettura; viene generato quando sul MOSI viene inviato il comando di lettura CMD $= (33)_{10}$;
- **WR** (porta di uscita, parallelismo 1 bit): impulso di un colpo di CK che indica all'interfaccia a registri la necessità di effettuare una scrittura; viene generato quando sul MOSI viene inviato il comando di lettura CMD $= (32)_{10}$;
- **A** (porta di uscita, parallelismo 8 bit): linea d'indirizzo, che indica il registro in cui si vuole scrivere il dato o dal quale lo si vuole leggere;

- **DIN** (porta di uscita, parallelismo 16 bit): bus dati utilizzato durante le operazioni di scrittura per inviare all'interfaccia a registri il dato da scrivere nell'indirizzo selezionato;
- **DOUT** (porta di ingresso, parallelismo 16 bit): bus dati utilizzato durante le operazioni di lettura per ricevere dall'interfaccia a registri il dato da restituire al master.

2.3 Progettazione di Execution Unit e Control Unit

L'architettura dello slave SPI progettato è mostrata in Figura A.1 (Appendice A), mentre la struttura della control unit può essere osservata nelle Figure B.1, B.2, B.3 e B.4 (Appendice B).

Come è possibile osservare, la topologia del datapath è piuttosto semplice ed è stata derivata dopo un'attenta analisi del timing di sistema, tenendo conto di due principi fondamentali:

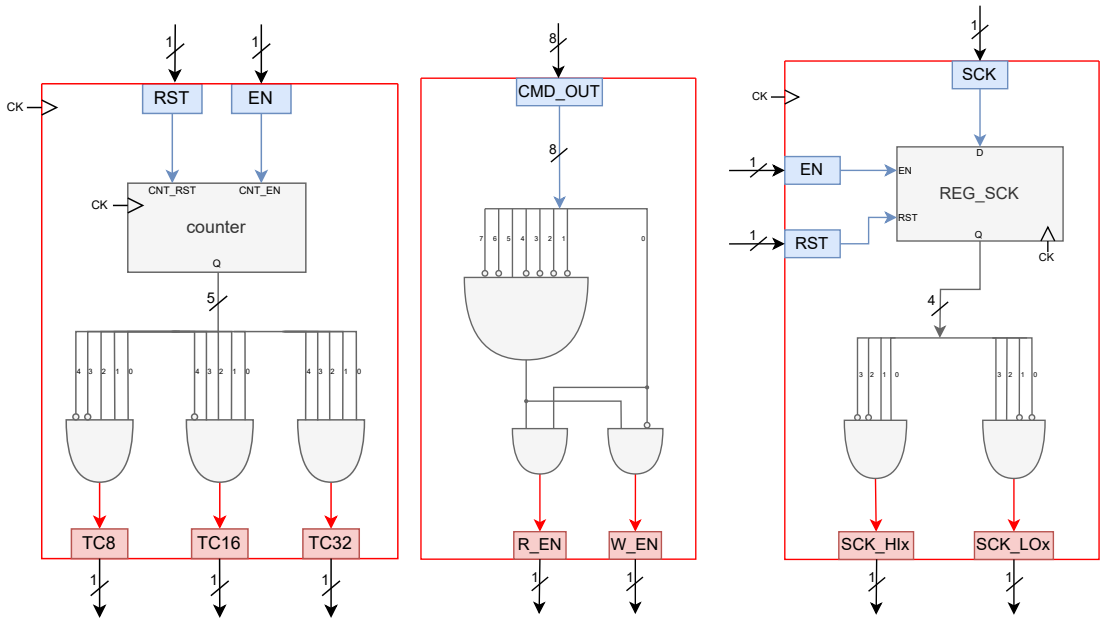
Ricezione da MOSI. Il comando CMD, l'indirizzo A e il dato da scrivere DIN, che vengono trasmessi dal master allo slave sulla linea MOSI, entrano nella IP sottoforma di uno stream seriale di bit e, per ricostruire correttamente le parole da 8 o 16 bit, è necessario avvalersi di un registro a scorrimento di tipo SIPO (*Serial In Parallel Out*);

Trasmissione su MISO. Il dato da leggere DOUT, che viene trasmesso dallo slave al master sulla linea MISO, entra nella IP come un vettore da 16 bit e deve essere "serializzato" mediante un registro a scorrimento di tipo PISO (*Parallel In Serial Out*).

2.3.1 Execution Unit

Sulla base delle considerazioni del paragrafo precedente, è ora possibile analizzare nel dettaglio l'architettura:

- La memorizzazione di CMD, A (sia in lettura sia in scrittura) e DIN (in scrittura) avviene mediante tre shift register SIPO, denotati rispettivamente con CMD_SR, ADD_SR e DIN_SR: tali registri sono dotati di ingressi seriali collegati al MOSI e di uscite parallele a 16 bit; queste ultime, a loro volta, sono gli ingressi di tre registri, CMD_REG, ADD_REG e DIN_REG, che campionano il dato solo quando tutti i bit della parola di interesse sono stati caricati all'interno dei registri a scorrimento.
- Durante le operazioni di lettura, DOUT viene trasformato in una sequenza seriale di bit mediante il PISO DOUT_SR; la linea MISO non è collegata direttamente a DOUT_SR, bensì all'uscita del multiplexer EXIT_MUX, il cui segnale di selezione è denotato con S_MISO (1 bit):
  1. Se S_MISO $= 0$, il MISO è in alta impedenza;
  2. Se S_MISO $= 1$, sul MISO viene trasmessa la sequenza di bit generata da DOUT_SR.
- A tale scopo, nel circuito è stato inserito un contatore a 5 bit, mostrato in Figura 2.2a, che viene incrementato ad ogni campionamento del MOSI da parte dello slave e solleva un flag quando arriva a 7 (TC8), 15 (TC16) e 31 (TC32): tali valori corrispondono rispettivamente, alla ricezione del bit meno significativo di CMD, A e DIN.
- Quando si solleva il flag TC8, CMD viene memorizzato in CMD_REG ed entra nel blocco combinatorio CMD_BLOCK, che analizza il comando ricevuto:
  1. Se CMD $= (32)_{10}$ (comando di scrittura) viene sollevato il flag W_EN;
  2. Se CMD $= (33)_{10}$ (comadno di lettura) viene sollevato il flag R_EN.



(a) COUNT **(b)** CMD_BLOCK **(c)** EDGE

Figura 2.2: Rappresentazione grafica dei blocchi più significativi presenti nella Execution Unit della IP.

- Dal momento che $f_{sck} \leq 10 f_{ck}$, SCK viene interpretato da CK come un vero e proprio dato e non come un segnale di clock; per tale motivo, i disturbi potrebbero essere interpretati erroneamente come dei fronti. Questo problema può essere aggirato utilizzando la tecnica del *sovracampionamento* o *oversampling*, che prevede di campionare il segnale ad una frequenza maggiore di quella prevista dal criterio di Nyquist. All'interno della execution unit è stata pertanto prevista la presenza del blocco EDGE, mostrato in Figura 2.2c: quattro campioni di SCK, campionati con frequenza pari a f_{ck}, vengono memorizzati in un SIPO:

  1. Se l'uscita del SIPO è pari a 1100, si ha un fronte di discesa di SCK e viene sollevato il flag SCK_HIx;
  2. Se l'uscita del SIPO è pari a 0011, si ha un fronte di salita di SCK e viene sollevato il flag SCK_LOx.

2.3.2 Control Unit

Per quanto riguarda la struttura dell'unità di controllo, è importante sottolineare quanto segue:

- È previsto uno stato di **reset complessivo** della macchina, durante il quale vengono resettati gli shift register (CMD_SR, ADD_SR, DIN_SR, DOUT_SR), i registri (CMD_REG, ADD_REG, DIN_REG) e il contatore; si entra in tale stato se viene attivato il segnale di reset esterno oppure quando il comando trasmesso sul MOSI differisce da quelli previsti per le operazioni di scrittura o lettura;

- Come già detto, le operazioni di scrittura e lettura possono avvenire solo se lo slave è selezionato dal master mediante lo slave select nSS; pertanto, subito dopo lo stato di reset

è previsto lo stato WAIT_nSS, in cui tutti i segnali, tranne il reset del contatore, assumono i valori di default e la macchina resta in *idle* finché nSS non viene attivato;

- Dopo l'asserimento di nSS sono presenti due gruppi di stati, colorati in giallo e arancione nell'ASM chart, che governano la memorizzazione di CMD e A; in questo momento, l'evoluzione di stato non dipende da W_EN e R_EN, dunque il comportamento della macchina è lo stesso indipendentemente dal tipo di transazione che si vuole effettuare;
- Una volta processato l'indirizzo, l'evoluzione di stato è determinata dal valore logico di W_EN e R_EN:
  1. Se W_EN = 1 si entra in un gruppo di stati che gestiscono la memorizzazione di DIN e l'invio del dato all'interfaccia a registri (colorato in blu nell'ASM chart);
  2. Viceversa, se R_EN = 1 si entra nel gruppo di stati preposto alla serializzazione di DOUT e alla sua trasmissione sulla linea MISO (colorato in verde nell'ASM chart).
- Una volta inviato DIN alla memoria (in scrittura) o terminata la trasmissione di DOUT sul MISO (in lettura), si entra nello stato di DONE, in cui i segnali di controllo non cambiano rispetto ai valori di default e si attende il deasserimento di nSS per poi tornare allo stato di *idle* WAIT_nSS.

3 Test del funzionamento

La descrizione in VHDL dell'IP e dei *component* necessari a realizzarla è riportata nell'appendice D.

Per testare il funzionamento del blocco, è stata creata una top entity (si veda l'appendice D.2.3) all'interno della quale le porte dello slave SPI sono collegate ad una memoria 256×16, dotata delle seguenti connessioni I/O:

- **D** (porta di ingresso, parallelismo 16 bit): bus dati utilizzato durante le operazioni di scrittura per la memorizzazione di un dato; va collegato all'uscita DIN dell'interfaccia SPI;
- **Q** (porta di uscita, parallelismo 16 bit): bus dati utilizzato durante le operazioni di lettura per inviare il dato richiesto; va collegato all'ingresso DOUT dell'interfaccia SPI;
- **ADDR** (porta di ingresso, numero intero codificabile su 8 bit): indirizzo della cella da scrivere o leggere; la conversione tra `integer` e `std_logic_vector` va effettuata nella top level mediante la definizione di un opportuno segnale;
- **WR** (porta di ingresso, parallelismo 1 bit): linea seriale su cui viene inviato l'impulso di scrittura; va collegato alla port WR dell'SPI;
- **RD** (porta di ingresso, parallelismo 1 bit): linea seriale su cui viene inviato l'impulso di lettura; va collegato alla port RD dell'SPI;

Il comportamento del circuito progettato è stato verificato dapprima mediante simulazioni e, successivamente, su scheda. I paragrafi che seguono sono dedicati ad una breve descrizione di tutti i test effettuati.

3.1 Simulazione ModelSim automatizzata con C++

Le simulazioni sono state condotte in due step successivi mediante il software ModelSim:

1. Preliminarmente, è stata scritta una semplice testbench a scopo di *debug* (Appendice E.1), che effettua alcune transazioni di prova nei registri 1, 2 e 5 dell'interfaccia di memoria; in tal modo, è possibile verificare "ad occhio" che il comportamento della IP sia corretto;
2. Una volta conclusa la prima fase della simulazione, il processo di simulazione è stato automatizzato mediante uno script in C++ basato sui principi della programmazione ad oggetti e una testbench con I/O da file (Appendici E.2 e E.3).

Non essendo presenti particolari limitazioni sulla dinamica dei dati da scrivere/leggere, ad eccezione del parallelismo di 16 bit, la simulazione automatizzata consiste nella generazione casuale di 256 numeri compresi tra 0 e 65535, ognuno dei quali viene scritto in uno dei 256 registri e, successivamente, letto. Per automatizzare la simulazione sono state scritte due classi: **Converter**, per la conversione dei numeri tra decimale e binario, e **Simulation**, per eseguire la simulazione e controllare la correttezza dei risultati.

La classe Simulation è dotata dei seguenti metodi pubblici:

1. `Simulation::generate`, che genera tre file per effettuare, rispettivamente, le transazioni di scrittura, quelle di lettura e il confronto dei risultati;
2. `Simulation::run`, per avviare la simulazione ModelSim da linea di comando mediante una chiamata una chiamata del comando `system`, contenuto nella libreria `cstdlib`;
3. `Simulation::report`, per confrontare il file di output prodotto da ModelSim con quello di riferimento generato con il metodo `generate`, e stampare a video un feedback circa la correttezza o meno dei risultati; in caso di errori, vengono stampati anche i numeri delle righe del file dei comandi che hanno creato problemi.

L'esecuzione del programma di collaudo, in cui vengono richiamati tutti i metodi sopracitati, ha permesso di verificare la correttezza delle transazioni effettuate in tutti i registri dell'interfaccia.

3.2 Test su piattaforma fisica

Una volta constatata la correttezza dei risultati prodotti dalla testbench, il funzionamento dell'IP è stato verificato mediante la scheda VirtLAB.

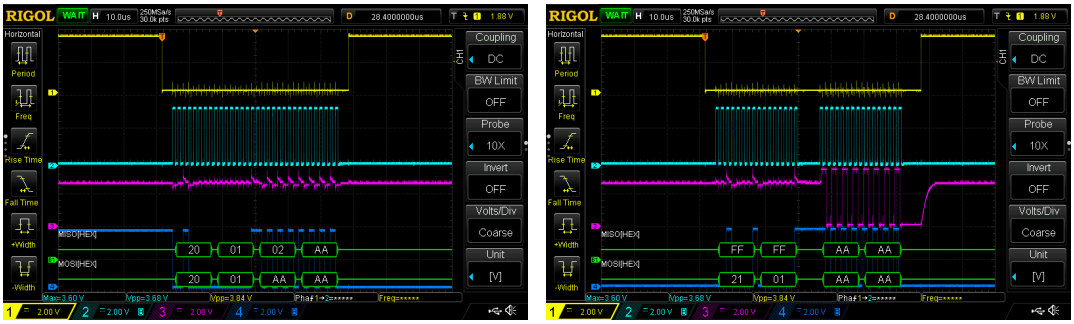
Innanzitutto, sul MCU user è stato caricato il file eseguibile `virtlab-user-spi-tester.elf`, che configura la porta USB utente come seriale virtuale.

A questo punto, dopo aver caricato sulla FPGA il file di configurazione prodotto dal sintetizzatore (`fpga-user.rbf`), è possibile effettuare delle transazioni di scrittura e lettura inviando opportuni comandi al MCU tramite seriale.

Come si può vedere dall'ouput dell'emulatore di terminale seriale, il corretto funzionamento è verificato: infatti, il dato letto da ciascun registro corrisponde a quello scritto per tutte le transazioni.

```
******************************
*   VirtLAB SPI tester v1.0   *
******************************

>w00aaaa
Writing aaaa to register 00
>w010603
Writing 0603 to register 01
```



(a) Scrittura **(b)** Lettura

Figura 3.1: Verifica del funzionamento dello slave SPI su scheda mediante misure con il DSO.

```
>w610322
Writing 0322 to register 61
>w680612
Writing 0612 to register 68
>wff1254
Writing 1254 to register ff
>w23fdc2
Writing fdc2 to register 23
>w367865
Writing 7865 to register 36
>wfe3196
Writing 3196 to register fe
>r00
Reading from register 00: aaaa
>r01
Reading from register 01: 0603
>r61
Reading from register 61: 0322
>r68
Reading from register 68: 0612
>rff
Reading from register ff: 1254
>r23
Reading from register 23: fdc2
>r36
Reading from register 36: 7865
>rfe
Reading from register fe: 3196
>
```

Le Figure 3.1a e 3.1b mostrano le misure effettuate collegando i pin di I/O del MCU su cui sono presenti i segnali dell'interfaccia SPI ai quattro canali dell'oscilloscopio digitale Rigol DS1054Z in dotazione in laboratorio. In particolare:

1. Il pin IO12 (nSS) è stato collegato al CH1 (in giallo);
2. Il pin IO13 (SCK) è stato collegato al CH2 (in azzurro);
3. Il pin IO14 (MISO) è stato collegato al CH3 (in rosa);
4. Il pin IO15 (MOSI) è stato collegato al CH4 (in blu).

L'oscilloscopio è stato configurato in modalità "SPI decoding" (pag. 8-15 del manuale e i segnali visualizzati sul display sono stati generati con una transazione di prova da terminale: in particolare, si è effettuata la scrittura del dato 'AAAA' all'indirizzo 1 e, successivamente, la lettura dello stesso registro.

Come ci si aspetta, durante la scrittura il MISO resta sempre in alta impedenza; in lettura, invece, esce da tale condizione solo durante la trasmissione del dato che, come si può vedere dai label in verde, avviene correttamente.

A Execution Unit

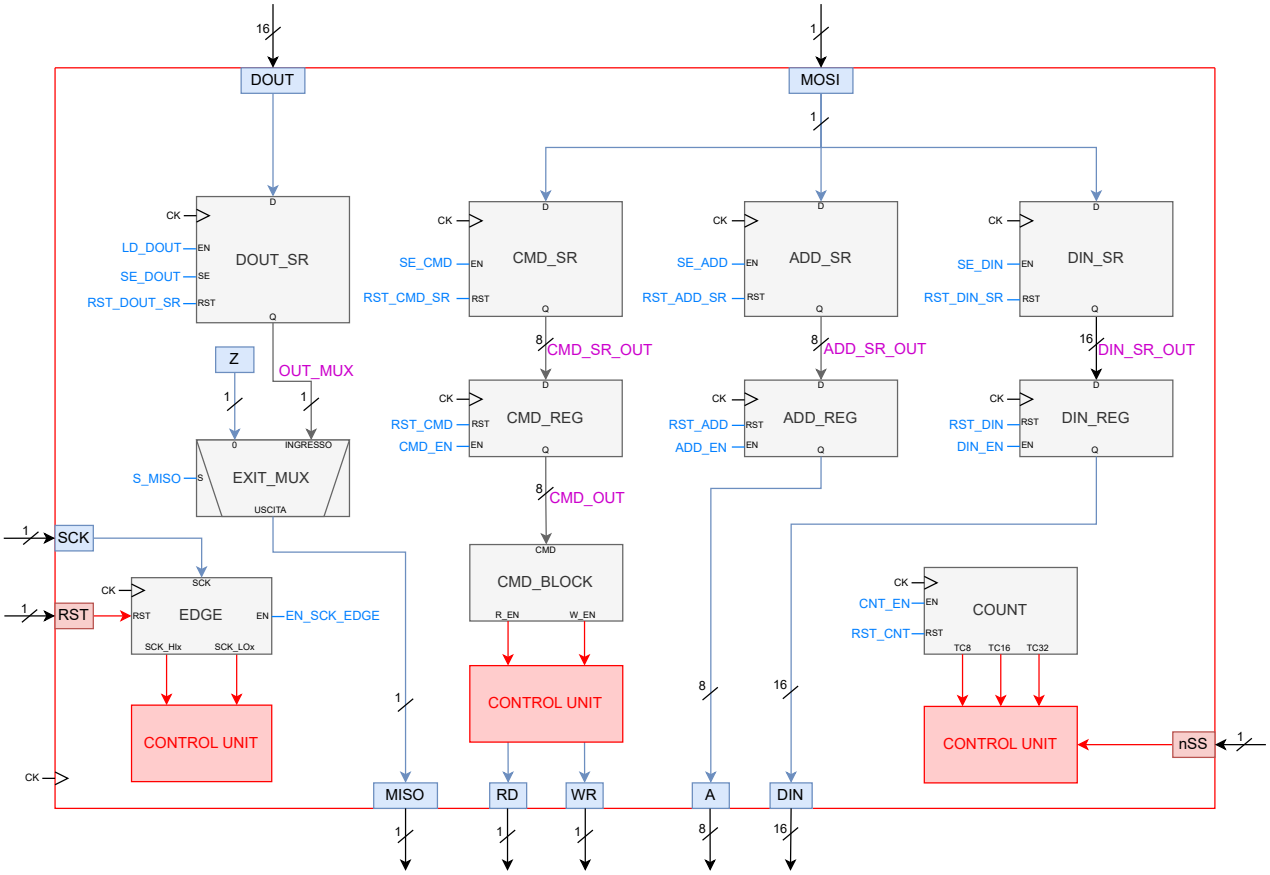


Figura A.1: Execution Unit dello slave SPI.

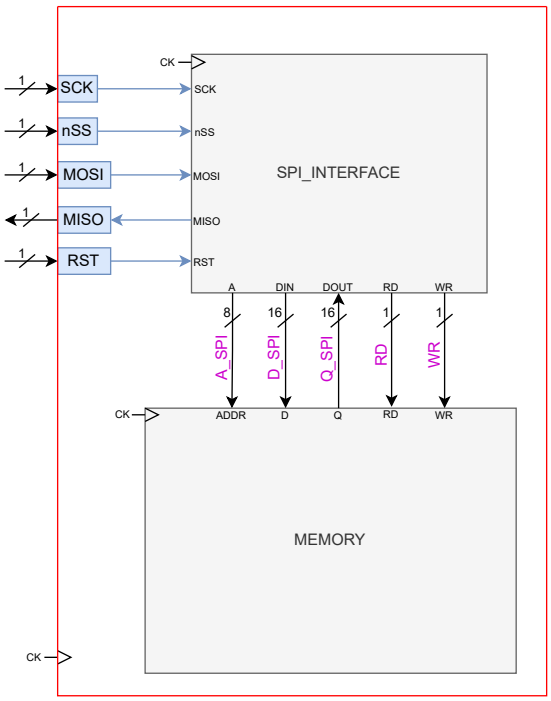


Figura A.2: Top level che mostra il collegamento tra lo slave SPI e l'interfaccia a registri.

B Control Unit

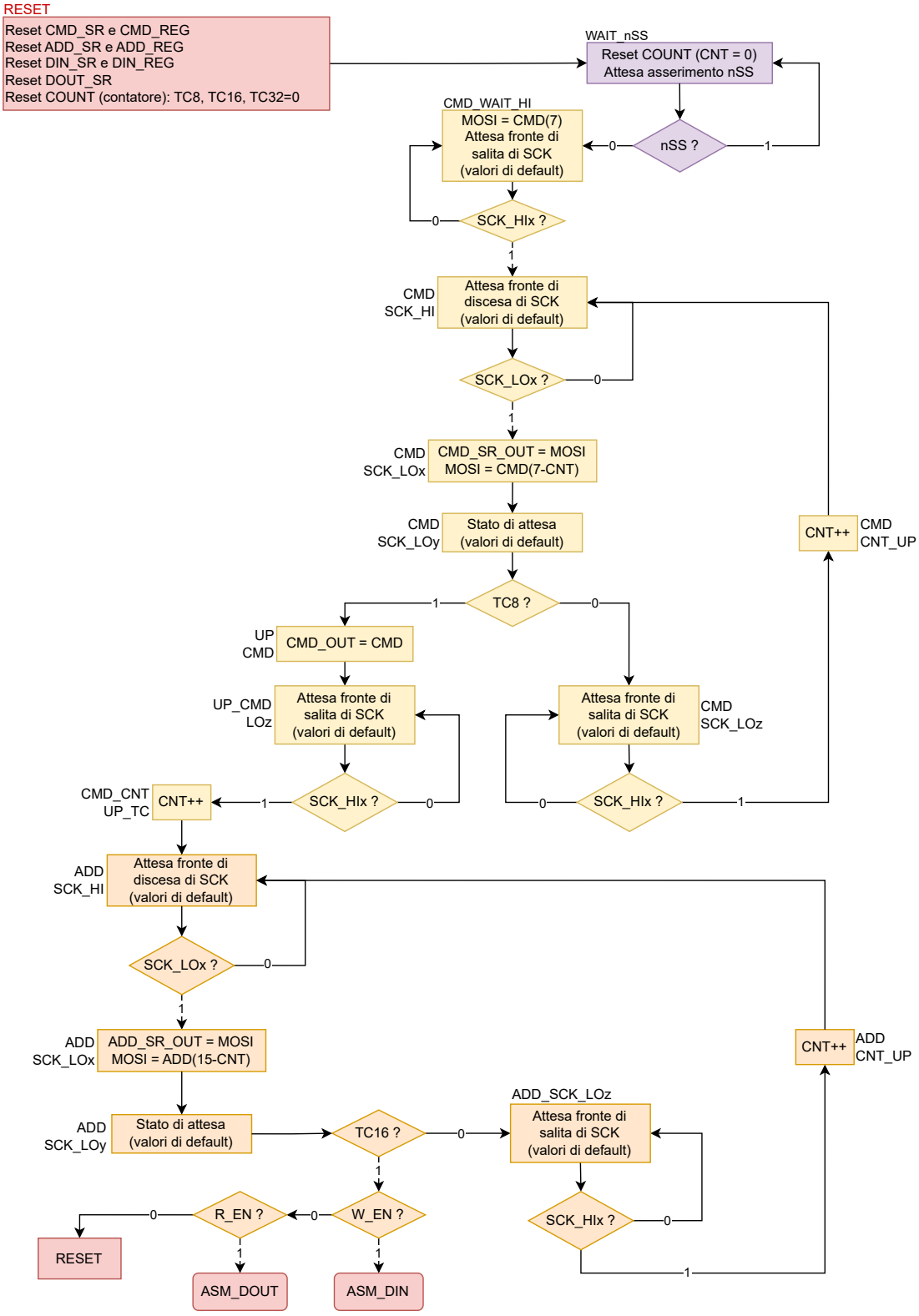


Figura B.1: ASM chart relativa alla trasmissione di comando e indirizzo sul MOSI

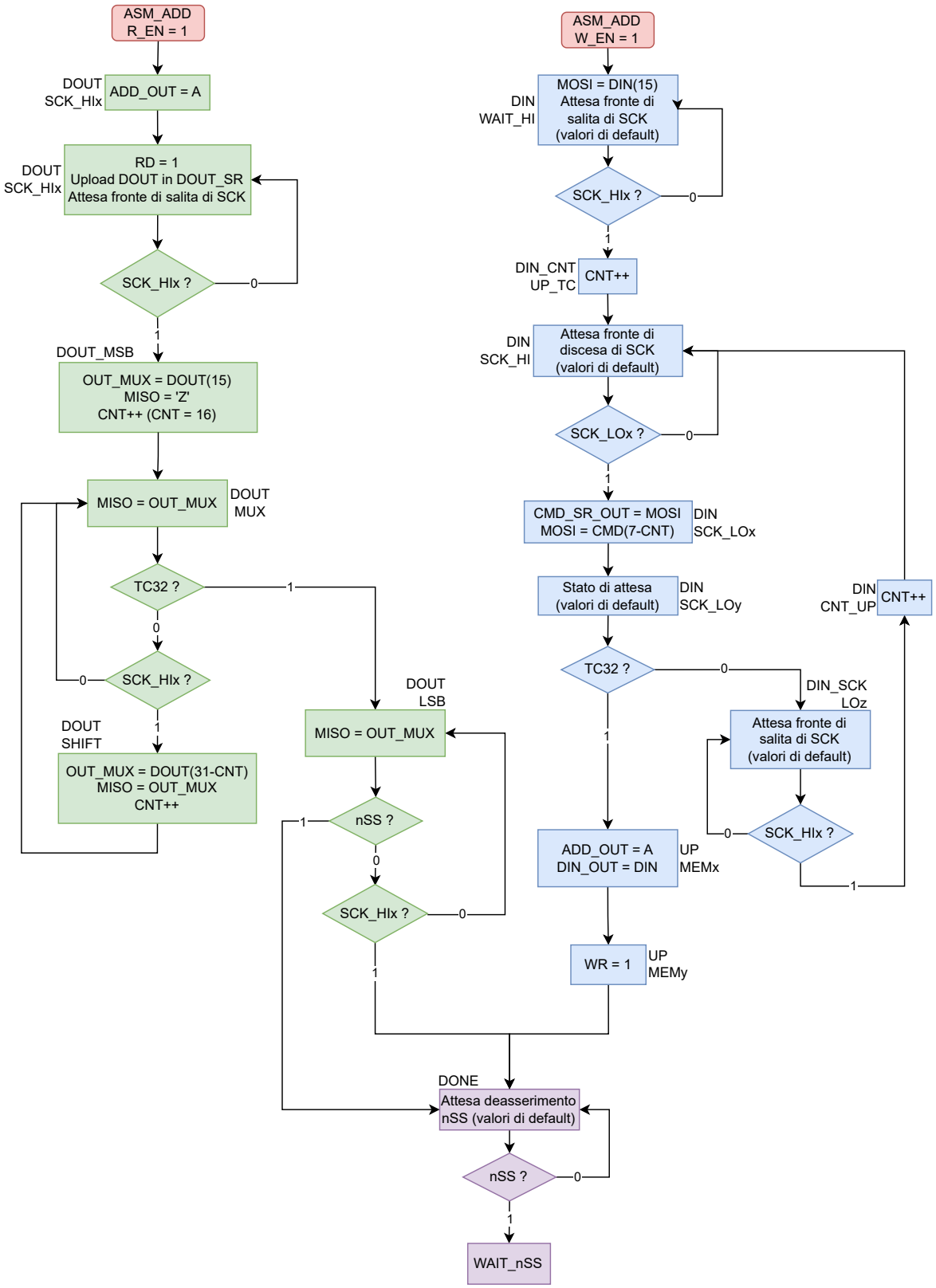


Figura B.2: ASM chart relativa alla trasmissione di DIN sul MOSI e di DOUT sul MISO

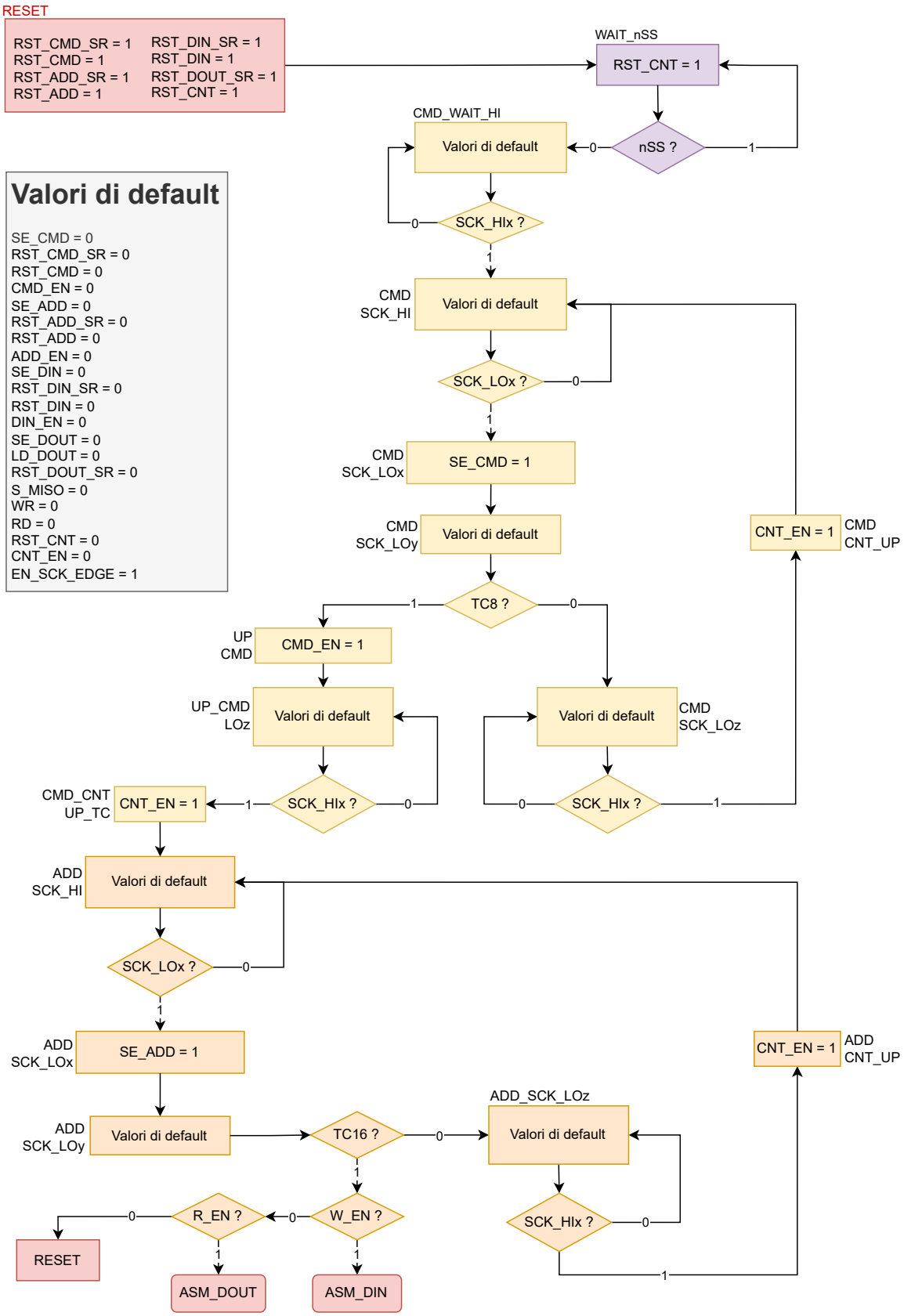


Figura B.3: Control ASM chart relativa alla trasmissione di comando e indirizzo sul MOSI

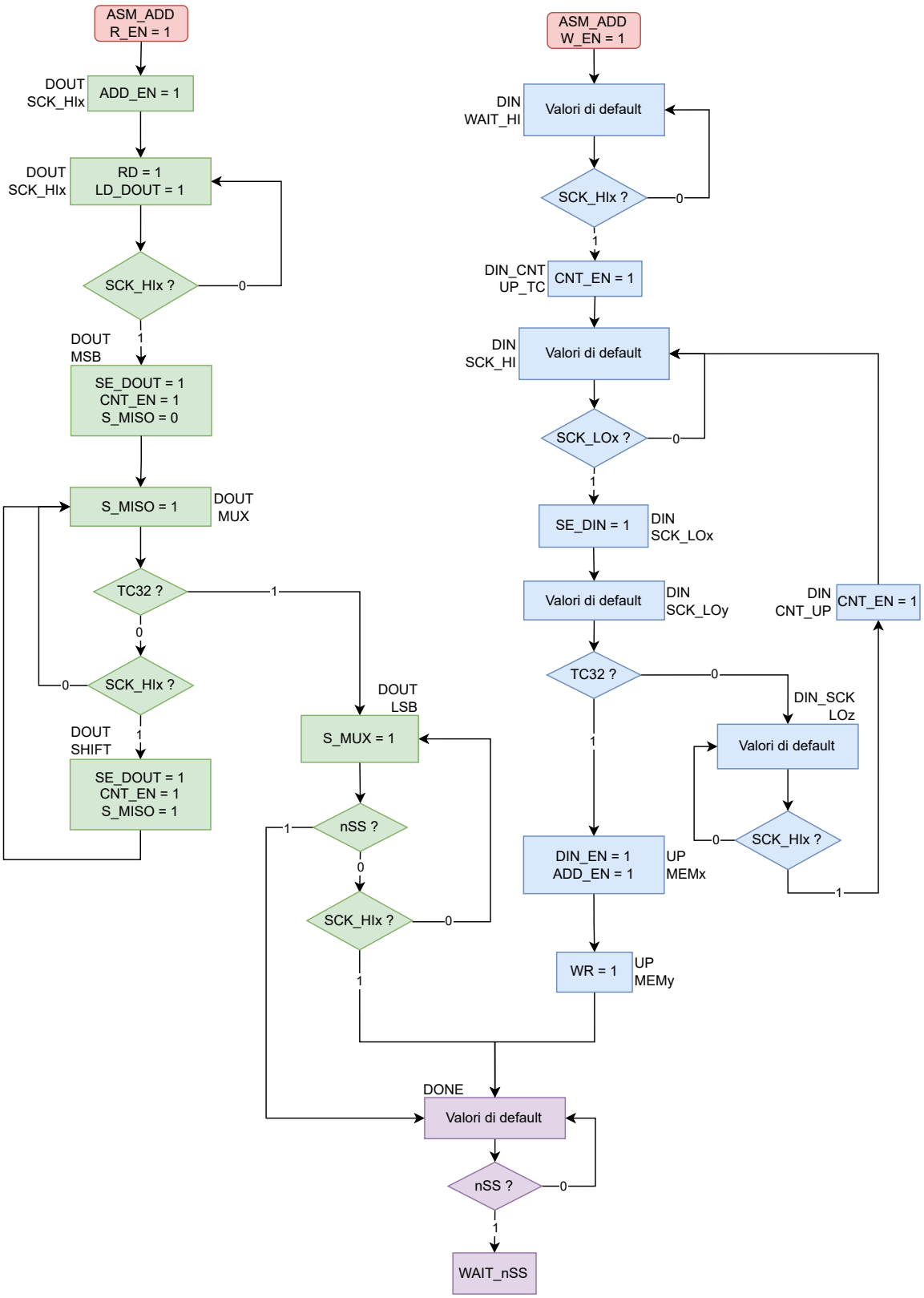
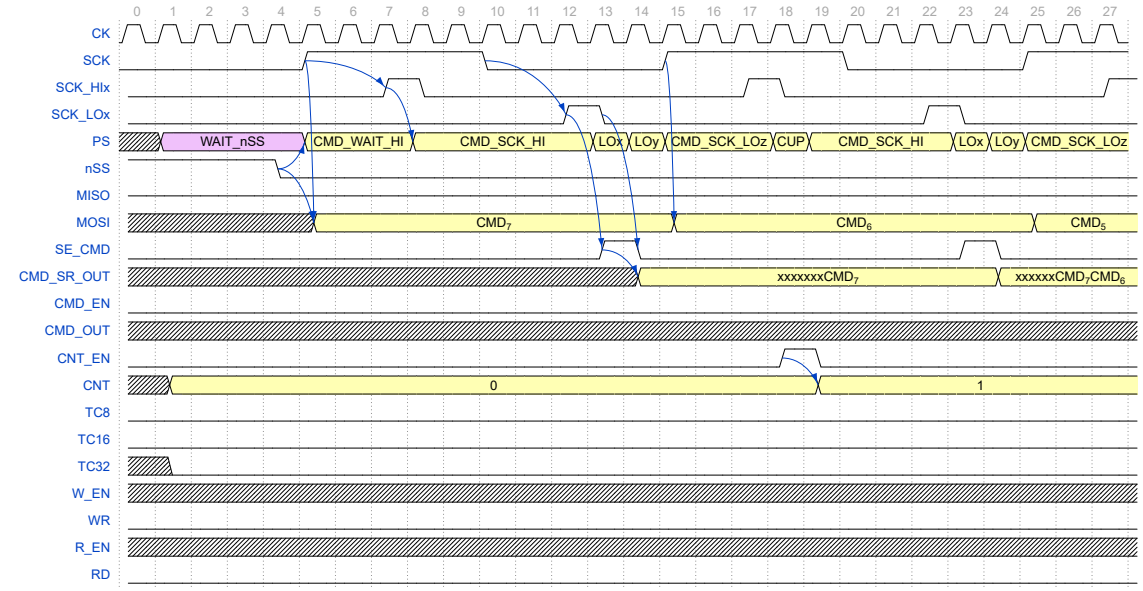
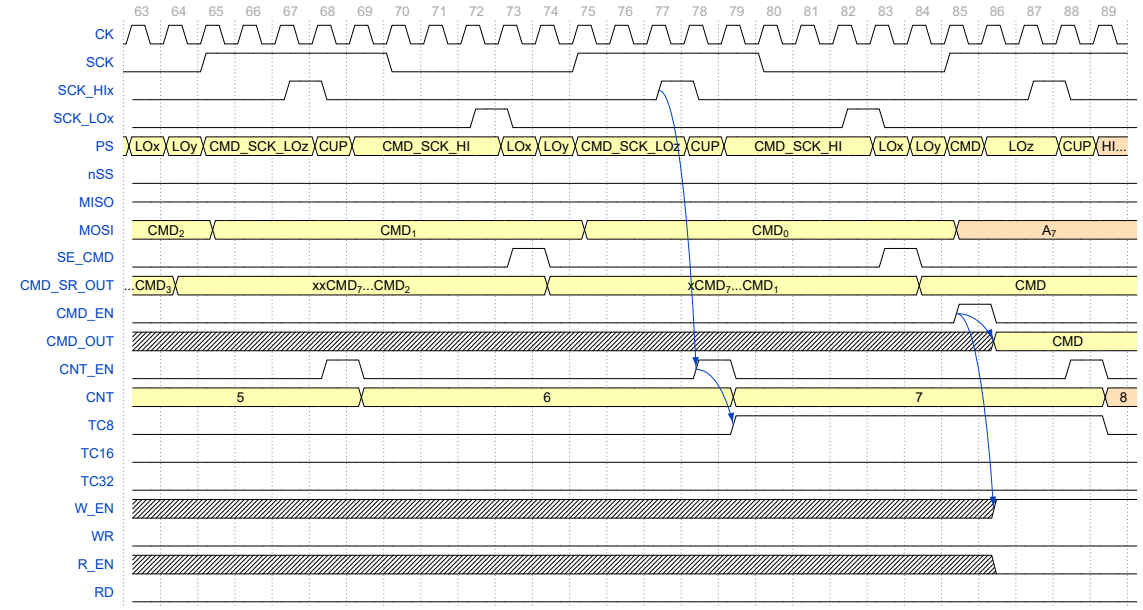
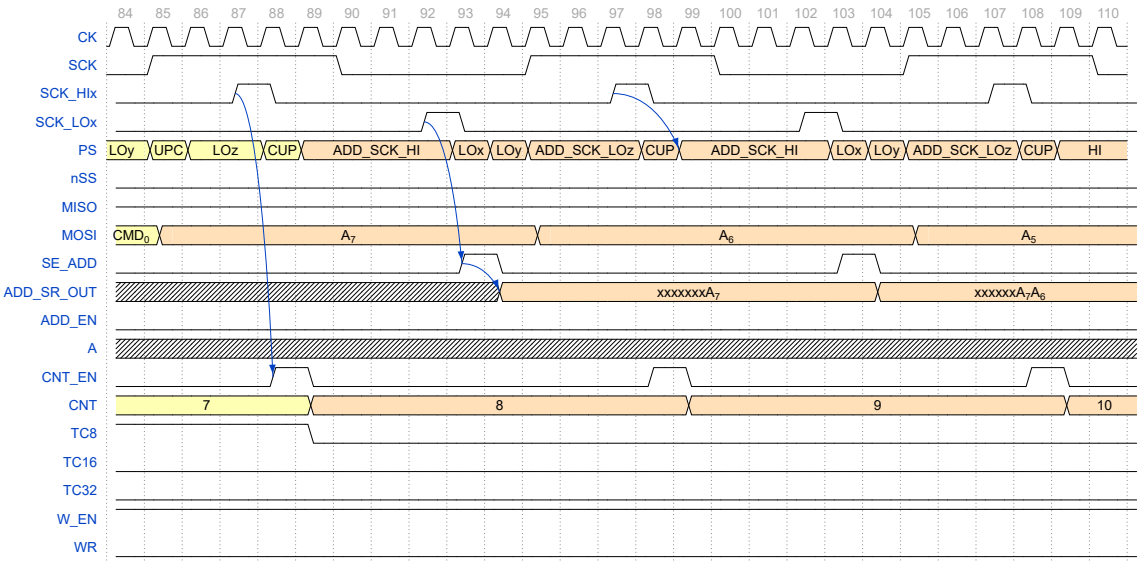


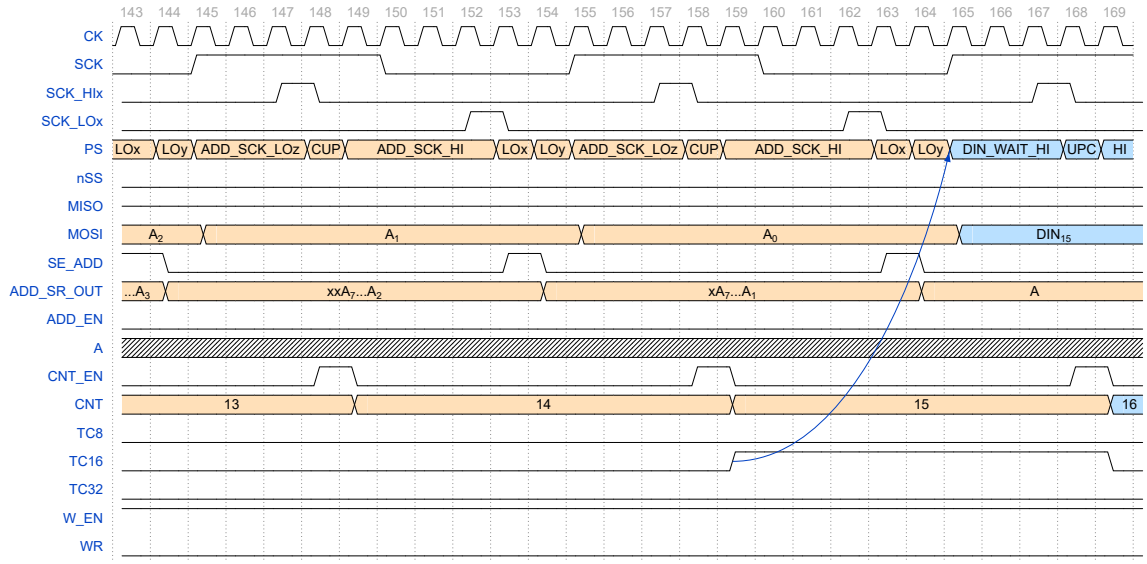
Figura B.4: Control ASM chart relativa alla trasmissione di DIN sul MOSI e di DOUT sul MISO

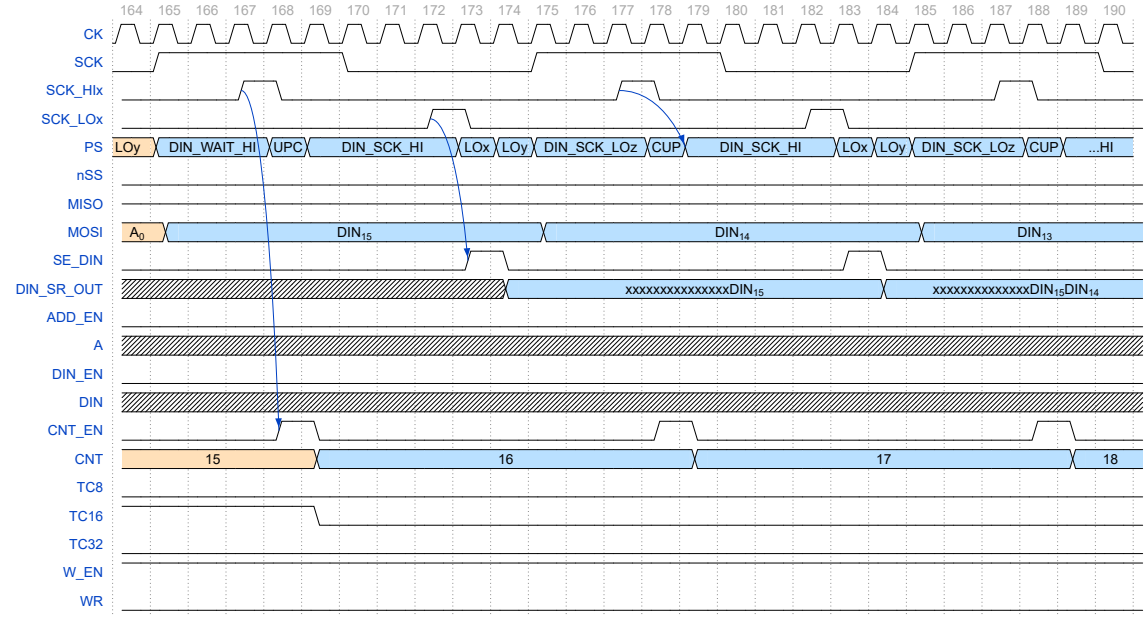
C Timing diagram

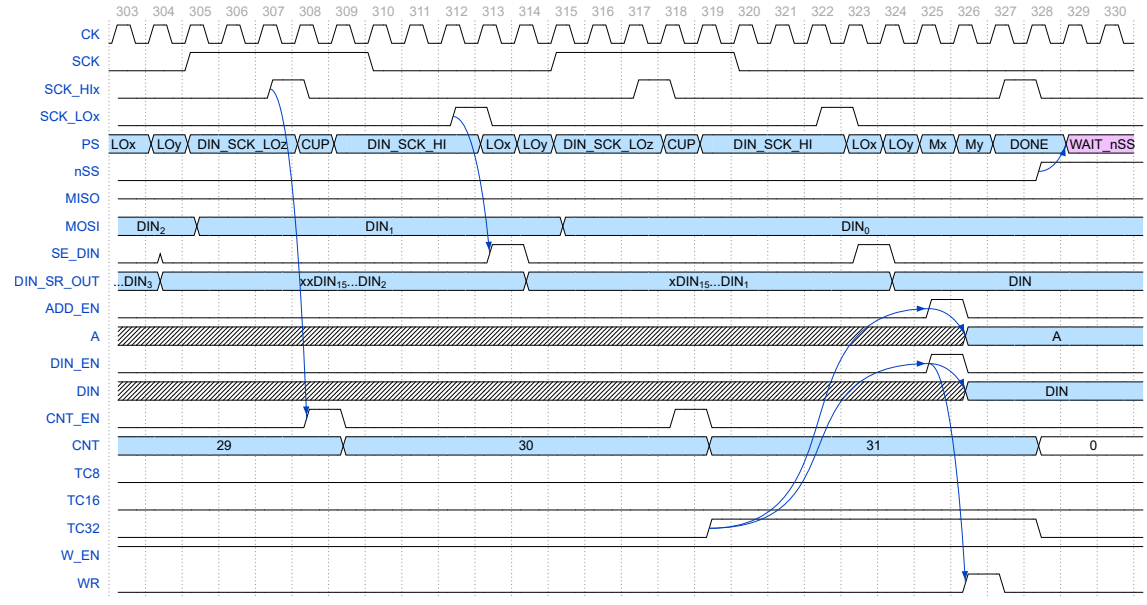


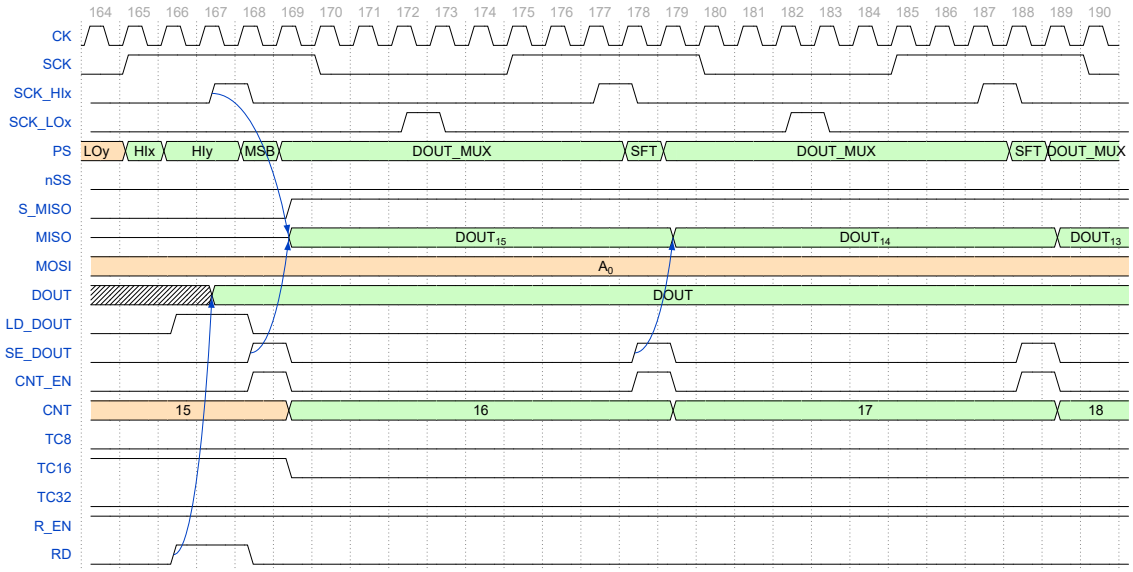


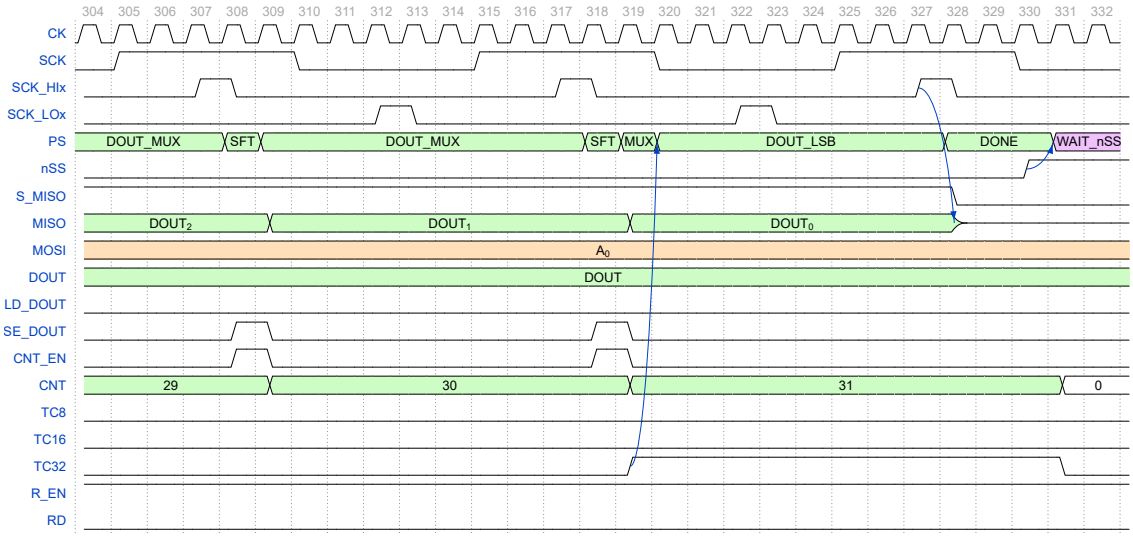



143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169
CK
SCK
SCK_HIx
SCK_LOx
PS
nSS
MISO
MOSI
SE_ADD
ADD_SR_OUT
ADD_EN
A
CNT_EN
CNT
TC8
TC16
TC32
W_EN
WR
LOx LOy ADD_SCK_LOz CUP ADD_SCK_HI LOx LOy ADD_SCK_LOz CUP ADD_SCK_HI LOx LOy DIN_WAIT_HI UPC HI
A2 A1 A0 DIN15
...A3 xxA7...A2 xA7...A1 A
13 14 15 16


164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190
CK
SCK
SCK_HIx
SCK_LOx
PS
nSS
MISO
MOSI
SE_DIN
DIN_SR_OUT
ADD_EN
A
DIN_EN
DIN
CNT_EN
CNT
TC8
TC16
TC32
W_EN
WR
LOy DIN_WAIT_HI UPC DIN_SCK_HI LOx LOy DIN_SCK_LOz CUP DIN_SCK_HI LOx LOy DIN_SCK_LOz CUP ...HI
A0 DIN15 DIN14 DIN13
xxxxxxxxxxxxxxDIN15
xxxxxxxxxxxxxxDIN15DIN14
15 16 17 18


303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330
CK
SCK
SCK_HIx
SCK_LOx
PS
nSS
MISO
MOSI
SE_DIN
DIN_SR_OUT
ADD_EN
A
DIN_EN
DIN
CNT_EN
CNT
TC8
TC16
TC32
W_EN
WR
LOx LOy DIN_SCK_LOz CUP DIN_SCK_HI LOx LOy DIN_SCK_LOz CUP DIN_SCK_HI LOx LOy Mx My DONE WAIT_nSS
DIN2 DIN1 DIN0
...DIN3 xxDIN15...DIN2 xDIN15...DIN1 DIN
A
DIN
29 30 31 0


164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190
CK
SCK
SCK_HIx
SCK_LOx
PS
LOy
HIx
HIy
MSB
DOUT_MUX
SFT
DOUT_MUX
SFT
DOUT_MUX
nSS
S_MISO
MISO
DOUT15
DOUT14
DOUT13
MOSI
A0
DOUT
DOUT
LD_DOUT
SE_DOUT
CNT_EN
CNT
15
16
17
18
TC8
TC16
TC32
R_EN
RD


304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332
CK
SCK
SCK_HIx
SCK_LOx
PS
DOUT_MUX
SFT
DOUT_MUX
SFT
MUX
DOUT_LSB
DONE
WAIT_nSS
nSS
S_MISO
MISO
DOUT2
DOUT1
DOUT0
MOSI
A0
DOUT
DOUT
LD_DOUT
SE_DOUT
CNT_EN
CNT
29
30
31
0
TC8
TC16
TC32
R_EN
RD

D Descrizione dell'hardware

D.1 Component

D.1.1 Registro

```
--*******************************************************************************
--* Registro con parallelismo di ingresso e uscita pari a N bit (generic)
--*******************************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity reg is
  generic (N : integer);
  port (
    d            : in std_logic_vector(N - 1 downto 0);
    clk, rst, en : in std_logic;
    q            : out std_logic_vector(N - 1 downto 0)
  );
end reg;

architecture structure of reg is
begin

  process (clk, rst)
  begin
    if (rst = '1') then --reset asincrono
      q <= (others => '0');
    elsif (clk'event and clk = '1') then --fronte di salita del clock
      if (en = '1') then
        q <= d;
      end if;
    end if;
  end process;

end structure;
```

D.1.2 SIPO

```
--*******************************************************************************
--* SIPO con parallelismo di uscita pari a N bit (generic)
--*******************************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity SIPO is
  generic (N : integer);
  port (
    clk : in std_logic;
    rst : in std_logic;
    en  : in std_logic;
    d   : in std_logic;
```

```
    q   : out std_logic_vector(N - 1 downto 0)
  );
end SIPO;

architecture structure of SIPO is

  signal data : std_logic_vector(N - 1 downto 0);

begin

  process (clk, rst)
  begin
    if (rst = '1') then -- reset attivo alto
      data <= (others => '0');
    elsif (clk'event and clk = '1') then -- fronte di salita del clock
      if en = '1' then
        data <= data(N - 2 downto 0) & d;
      end if;
    end if;
  end process;

  q <= data;

end structure;
```

D.1.3 PISO

```
--*****************************************************************************
--* PISO con parallelismo di ingresso pari a N bit (generic)
--*****************************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity PISO is
  generic (N : integer := 16);
  port (
    clk : in std_logic;
    se  : in std_logic; -- shift enable
    rst : in std_logic;
    en  : in std_logic; -- load enable
    d   : in std_logic_vector(N - 1 downto 0);
    q   : out std_logic
  );
end PISO;

architecture structure of PISO is

  signal data : std_logic_vector(N - 1 downto 0);

begin
  process (CLK, RST)
  begin
    if (RST = '1') then                 -- reset attivo alto
      data <= (others => '0');            -- reset
      q    <= 'Z';                        -- uscita in alta impedenza
```

```
    elsif (clk'event and clk = '1') then -- fronte di salita del clock
      if (EN = '1') then                    -- load
        data <= d;
      elsif (EN = '0' and SE = '1') then            -- shift
        q                    <= data(15);             -- mando fuori il MSB
        data(N - 1 downto 1) <= data(N - 2 downto 0); -- shifto di 1 bit verso sx
        data(0)              <= '0';                  -- appendo uno zero a dx (LSB)
      end if;
    end if;
  end process;

end structure;
```

D.1.4 Rilevatore dei fronti di SCK

```
--*******************************************************************************
--* Rilevatore dei fronti di SCK con sovracampionamento
--* Fronte di discesa = 1100
--* Fronte di salite = 0011
--*******************************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity clock_edge is
  generic (N : integer := 4);
  port (
    sck          : in std_logic;
    clk, en, rst : in std_logic;
    sck_lox      : out std_logic;
    sck_hix      : out std_logic
  );
end clock_edge;

architecture structure of clock_edge is

  component SIPO is
    generic (N : integer);
    port (
      clk : in std_logic;
      rst : in std_logic;
      en  : in std_logic;
      d   : in std_logic;
      q   : out std_logic_vector(N - 1 downto 0)
    );
  end component;

  signal edge : std_logic_vector(3 downto 0);

begin
  REG_SCK : SIPO
  generic map(N => N)
  port map(clk => clk, rst => rst, en => en, d => sck, q => edge);

  SCK_LOx <= (edge(3) and edge(2)) and (not(edge(1)) and not(edge(0)));
  SCK_HIx <= (not(edge(3)) and not(edge(2))) and (edge(1) and edge(0));
```

```
43
44  end structure;
```

D.1.5 Rilevatore del comando *read*/*write*

```
1   --*********************************************************************************
2   --* Rilevatore del comando di scrittura o lettura
3   --* Scrittura = 00100000
4   --* Lettura = 00100001
5   --*********************************************************************************
6
7   library ieee;
8   use ieee.std_logic_1164.all;
9   use ieee.numeric_std.all;
10
11  entity command is
12    port (
13      cmd  : in std_logic_vector(7 downto 0);
14      w_en : out std_logic;
15      r_en : out std_logic
16    );
17  end command;
18
19  architecture structure of command is
20
21  begin
22    w_en <= (not(cmd(7)) and not(cmd(6)) and cmd(5) and not(cmd(4)) and not(cmd(3)) and
      not(cmd(2)) and not(cmd(1))) and not(cmd(0));
23    r_en <= (not(cmd(7)) and not(cmd(6)) and cmd(5) and not(cmd(4)) and not(cmd(3)) and
      not(cmd(2)) and not(cmd(1))) and cmd(0);
24
25  end structure;
```

D.1.6 Contatore a 5 bit

```
1   --*********************************************************************************
2   --* Contatore su 5 bit con rilevatore di 7 (TC8), 15 (TC16) e 31 (TC32)
3   --*********************************************************************************
4
5   library ieee;
6   use ieee.std_logic_1164.all;
7   use ieee.numeric_std.all;
8
9   entity counter is
10    port (
11      en, rst, clk    : in std_logic;
12      tc8, tc16, tc32 : out std_logic
13    );
14  end counter;
15
16  architecture structure of counter is
17
18    signal Q : unsigned(4 downto 0);
19
20  begin
21    process (clk, en, rst)
```

```
  begin
    if (rst = '1') then -- reset attivo alto
      Q <= (others => '0');
    elsif (clk'event and clk = '1') then -- fronte di salita del clock
      if (en = '1') then
        Q <= Q + 1;
      end if;
    end if;
  end process;

  tc8  <= Q(0) and Q(1) and Q(2) and not(Q(3)) and not(Q(4)); --7=00111
  tc16 <= Q(0) and Q(1) and Q(2) and Q(3) and not(Q(4));      --15=01111
  tc32 <= Q(0) and Q(1) and Q(2) and Q(3) and Q(4);           --31=11111

end structure;
```

D.1.7 Multiplexer a due vie per l'alta impedenza

```
--*******************************************************************************
--* Multiplexer a due vie con ingressi e uscita su 1 bit
--* s=0: l'uscita va in alta impedenza
--* s=1: trasmettiamo in uscita il valore presente in ingresso
--*******************************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mux_z is
  port (
    ingresso : in std_logic; --input
    s        : in std_logic; --selettore
    uscita   : out std_logic --output
  );
end mux_z;

architecture structure of mux_z is

begin
  uscita <= 'Z' when s = '0' --alta impedenza (s=0)
    else
    ingresso; --trasmissione dato (s=1)

end structure;
```

D.2 Progetto completo

D.2.1 Register file

```
--*******************************************************************************
--* Memoria di tipo register file con capacità 256xN
--* Gli indirizzi cono codificati su 8 bit (quindi abbiamo 256 locazioni 0-255)
--* Il parallelismo dei dati è pari a N bit (generic)
--*******************************************************************************

library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity MEM is
  generic (N : integer := 16);
  port (
    CK   : in std_logic;
    WR   : in std_logic;                          --write (attivo alto)
    RD   : in std_logic;                          --read (attivo alto)
    ADDR : in integer range 0 to 255;             --8 bit
    D    : in std_logic_vector(N - 1 downto 0); --16 bit
    Q    : out std_logic_vector(N - 1 downto 0) --16 bit
  );
end MEM;

architecture structure of MEM is

  type ram_array is array (0 to 255) of std_logic_vector (N - 1 downto 0);
  signal ram : ram_array;

begin

  process (CK)
  begin
    if (CK'event and CK = '1') then -- fronte di salita del CK
      if (WR = '1') then          -- scrittura
        ram(ADDR) <= D;
      elsif (RD = '1') then       -- lettura
        Q <= ram(ADDR);
      end if;
    end if;
  end process;

end structure;
```

D.2.2 Slave SPI

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity spi is
  port (
    CK, SCK : in std_logic;                     -- main clock e clock di sistema
    nSS     : in std_logic;                     -- slave select (attivo basso)
    RST     : in std_logic;                     -- reset (attivo alto)
    MOSI    : in std_logic;                     -- Master Out Slave In
    MISO    : out std_logic;                    -- Master In Slave Out
    RD, WR  : out std_logic;                    -- segnali di controllo per memoria
    A       : out std_logic_vector(7 downto 0);  -- indirizzo di memoria
    DIN     : out std_logic_vector(15 downto 0); -- ingresso memoria (uscita per spi)
    DOUT    : in std_logic_vector(15 downto 0)   -- uscita memoria (ingresso per spi)
  );
end spi;

architecture structure of spi is

```

```
21    --*****************************************************************************
22    --* Elenco degli stati
23    --*****************************************************************************
24
25    type state_type is (
26      RESET, WAIT_nSS,
27      -- trasmissione CMD su MOSI ------------------------------------------------
28      CMD_WAIT_HI, CMD_SCK_HI, CMD_SCK_LOx, CMD_SCK_LOy, CMD_SCK_LOz, CMD_CNT_UP, UP_CMD,
        UP_CMD_LOz, CMD_CNT_UP_TC,
29      -- trasmissione ADD su MOSI ------------------------------------------------
30      ADD_SCK_HI, ADD_SCK_LOx, ADD_SCK_LOy, ADD_SCK_LOz, ADD_CNT_UP,
31      -- trasmissione DIN su MOSI ------------------------------------------------
32      DIN_WAIT_HI, DIN_CNT_UP_TC, DIN_SCK_HI, DIN_SCK_LOx, DIN_SCK_LOy, DIN_SCK_LOz,
        DIN_CNT_UP,
33      -- scrittura DIN in memoria ------------------------------------------------
34      UP_MEMx, UP_MEMy,
35      -- trasmissione DOUT su MISO -----------------------------------------------
36      DOUT_SCK_HIx, DOUT_SCK_HIy, DOUT_SHIFT, DOUT_MUX, DOUT_LSB, DOUT_MSB, DONE
37    );
38    signal PS, NS : state_type; -- present state (PS) e next state (NS)
39
40    --*****************************************************************************
41    --* Definizione segnali interni (N.B. I SEGNALI DI CONTROLLO SONO TUTTI ATTIVI ALTI)
42    --*****************************************************************************
43
44    signal SE_CMD, RST_CMD_SR, RST_CMD, CMD_EN : std_logic; -- comando
45    signal SE_ADD, RST_ADD_SR, RST_ADD, ADD_EN : std_logic; -- indirizzo
46    signal SE_DIN, RST_DIN_SR, RST_DIN, DIN_EN : std_logic; -- dato in scrittura
47    signal SE_DOUT, LD_DOUT, RST_DOUT_SR       : std_logic; -- dato in lettura
48    signal RST_CNT, CNT_EN, TC8, TC16, TC32    : std_logic; -- contatore
49    signal SCK_LOx, SCK_HIx, EN_SCK_EDGE       : std_logic; -- rilevatori fronti di SCK
50    signal S_MISO                              : std_logic; -- selettore mux di uscita
51
52    signal CMD_SR_OUT : std_logic_vector(7 downto 0);
53    signal CMD_OUT    : std_logic_vector(7 downto 0);
54    signal R_EN, W_EN : std_logic;
55    signal ADD_SR_OUT : std_logic_vector(7 downto 0);
56    signal DIN_SR_OUT : std_logic_vector(15 downto 0);
57    signal OUT_MUX    : std_logic;
58
59    --*****************************************************************************
60    --* Dichiarazione component
61    --*****************************************************************************
62
63    -- registro con ingressi e uscite su N bit
64    component reg is
65      generic (N : integer);
66      port (
67        d            : in std_logic_vector(N - 1 downto 0);
68        clk, rst, en : in std_logic;
69        q            : out std_logic_vector(N - 1 downto 0)
70      );
71    end component;
72
73    -- shift register SIPO con uscite su N bit
74    component SIPO is
75      generic (N : integer);
76      port (
77        clk : in std_logic;
```

```
78        rst : in std_logic;
79        en  : in std_logic;
80        d   : in std_logic;
81        q   : out std_logic_vector(N - 1 downto 0)
82      );
83    end component;
84
85    -- registro parallel in serial out con ingressi su N bit
86    component PISO is
87      generic (N : integer := 16);
88      port (
89        clk : in std_logic;
90        se  : in std_logic;
91        rst : in std_logic;
92        en  : in std_logic;
93        d   : in std_logic_vector(N - 1 downto 0);
94        q   : out std_logic
95      );
96    end component;
97
98    -- contatore a 5 bit con rilevatore di 7, 15, 31
99    component counter is
100     port (
101       en, rst, clk    : in std_logic;
102       tc8, tc16, tc32 : out std_logic
103     );
104   end component;
105
106   -- rilevatore del comando di scrittura o lettura
107   component command is
108     port (
109       cmd  : in std_logic_vector(7 downto 0);
110       w_en : out std_logic;
111       r_en : out std_logic
112     );
113   end component;
114
115   -- rilevatore dei fronti di SCK
116   component clock_edge is
117     generic (N : integer := 4);
118     port (
119       sck          : in std_logic;
120       clk, en, rst : in std_logic;
121       sck_lox      : out std_logic;
122       sck_hix      : out std_logic
123     );
124   end component;
125
126   -- multiplexer a due vie che collega l'ingresso all'uscita oppure la mette in Z
127   component mux_z is
128     port (
129       ingresso : in std_logic; -- input
130       s        : in std_logic; -- selettore
131       uscita   : out std_logic -- output
132     );
133   end component;
134
135   --*****************************************************************************
136   --* Architecture
```

```
137   --*****************************************************************************
138
139 begin
140
141   --* STEP 1: ASM dei controlli ********************************************************
142   controlASM : process (PS, nSS, SCK_LOx, SCK_HIx, TC8, TC16, TC32, W_EN, R_EN)
143   begin
144
145     ------------------------------------------------------------------------------
146     -- Valori di default -----------------------------------------------------------
147     SE_CMD     <= '0';
148     RST_CMD_SR <= '0';
149     RST_CMD    <= '0';
150     CMD_EN     <= '0';
151     --
152     SE_ADD     <= '0';
153     RST_ADD_SR <= '0';
154     RST_ADD    <= '0';
155     ADD_EN     <= '0';
156     --
157     SE_DIN     <= '0';
158     RST_DIN_SR <= '0';
159     RST_DIN    <= '0';
160     DIN_EN     <= '0';
161     --
162     SE_DOUT     <= '0';
163     LD_DOUT     <= '0';
164     RST_DOUT_SR <= '0';
165     S_MISO      <= '0';
166     --
167     WR <= '0';
168     RD <= '0';
169     --
170     RST_CNT <= '0';
171     CNT_EN  <= '0';
172     --
173     EN_SCK_EDGE <= '1';
174     ------------------------------------------------------------------------------
175
176     case PS is
177
178       when RESET => -- resetto la macchina
179         RST_CMD_SR  <= '1';
180         RST_CMD     <= '1';
181         RST_ADD_SR  <= '1';
182         RST_ADD     <= '1';
183         RST_DIN_SR  <= '1';
184         RST_DIN     <= '1';
185         RST_DOUT_SR <= '1';
186         RST_CNT     <= '1';
187         -------------------
188         NS <= WAIT_nSS;
189
190         ---- ASM_CMD -----------------------------------------------------------
191         -- Il master invia gli 8 bit di CMD sul MOSI
192       when WAIT_nSS => -- reset contatore (TC8, TC16, TC32 = 0), aspetto asserimento di
          nSS
193         RST_CNT <= '1';
194         -------------------
```

```
195           if (nSS = '0') then
196             NS <= CMD_WAIT_HI;
197           else
198             NS <= WAIT_nSS;
199           end if;
200
201         when CMD_WAIT_HI => -- valori di default, aspetto fronte di salita SCK
202           if (SCK_HIx = '1') then
203             NS <= CMD_SCK_HI;
204           else
205             NS <= CMD_WAIT_HI;
206           end if;
207
208         when CMD_SCK_HI => -- valori di default, aspetto fronte di discesa SCK per
            campionare il MOSI
209           if (SCK_LOx = '1') then
210             NS <= CMD_SCK_LOx;
211           else
212             NS <= CMD_SCK_HI;
213           end if;
214
215         when CMD_SCK_LOx => -- CMD_SR campiona MOSI = CMD(7-CNT)
216           SE_CMD <= '1';
217           -------------------
218           NS <= CMD_SCK_LOy;
219
220         when CMD_SCK_LOy => -- stato di attesa, valori di default
221           if (TC8 = '1') then
222             NS <= UP_CMD;
223           else
224             NS <= CMD_SCK_LOz;
225           end if;
226
227         when CMD_SCK_LOz => -- valori di default, aspetto fronte di salita SCK per
            incrementare il contatore
228           if (SCK_HIx = '1') then
229             NS <= CMD_CNT_UP;
230           else
231             NS <= CMD_SCK_LOz;
232           end if;
233
234         when CMD_CNT_UP => -- incremento contatore con TC8=0
235           CNT_EN <= '1';
236           -------------------
237           NS <= CMD_SCK_HI;
238
239         when UP_CMD => -- memorizzo CMD in CMD_REG
240           CMD_EN <= '1';
241           -------------------
242           NS <= UP_CMD_LOz;
243
244         when UP_CMD_LOz => -- valori di default, aspetto fronte di salita SCK per
            incrementare il contatore
245           if (SCK_HIx = '1') then
246             NS <= CMD_CNT_UP_TC;
247           else
248             NS <= UP_CMD_LOz;
249           end if;
250
```

```
251         when CMD_CNT_UP_TC => -- incremento contatore con TC8=1
252           CNT_EN <= '1';
253           -------------------
254           NS <= ADD_SCK_HI;
255
256           ---- ASM_ADD ------------------------------------------------------------
257           -- Il master invia gli 8 bit di indirizzo sul MOSI
258         when ADD_SCK_HI => -- valori di default, aspetto fronte di discesa SCK per
            campionare il MOSI
259           if (SCK_LOx = '1') then
260             NS <= ADD_SCK_LOx;
261           else
262             NS <= ADD_SCK_HI;
263           end if;
264
265         when ADD_SCK_LOx => -- ADD_SR campiona MOSI = ADD(15-CNT)
266           SE_ADD <= '1';
267           -------------------
268           NS <= ADD_SCK_LOy;
269
270         when ADD_SCK_LOy => -- stato di attesa, valori di default
271           if (TC16 = '1') then
272             if (W_EN = '1') then -- scrittura
273               NS <= DIN_WAIT_HI;
274             elsif (R_EN = '1') then -- lettura
275               NS <= DOUT_SCK_HIx;
276             else
277               NS <= RESET;
278             end if;
279           else
280             NS <= ADD_SCK_LOz;
281           end if;
282
283         when ADD_SCK_LOz => -- valori di default, aspetto fronte di salita SCK per
            incrementare il contatore
284           if (SCK_HIx = '1') then
285             NS <= ADD_CNT_UP;
286           else
287             NS <= ADD_SCK_LOz;
288           end if;
289
290         when ADD_CNT_UP => -- incremento contatore con TC16=0
291           CNT_EN <= '1';
292           -------------------
293           NS <= ADD_SCK_HI;
294
295           ---- ASM_DIN ------------------------------------------------------------
296           -- Il master invia i 16 bit di DIN sul MOSI
297         when DIN_WAIT_HI => -- valori di default, aspetto fronte di salita SCK per
            incrementare il contatore
298           if (SCK_HIx = '1') then
299             NS <= DIN_CNT_UP_TC;
300           else
301             NS <= DIN_WAIT_HI;
302           end if;
303
304         when DIN_CNT_UP_TC => -- incremento il contatore con TC16=1
305           CNT_EN <= '1';
306           -------------------
```

```
307           NS <= DIN_SCK_HI;
308
309         when DIN_SCK_HI => -- valori di default, aspetto fronte di discesa SCK per
            campionare il MOSI
310           if (SCK_LOx = '1') then
311             NS <= DIN_SCK_LOx;
312           else
313             NS <= DIN_SCK_HI;
314           end if;
315
316         when DIN_SCK_LOx => -- DIN_SR campiona MOSI = DIN(31-CNT)
317           SE_DIN <= '1';
318           -------------------
319           NS <= DIN_SCK_LOy;
320
321         when DIN_SCK_LOy => -- stato di attesa, valori di default
322           if (TC32 = '0') then
323             NS <= DIN_SCK_LOz;
324           else
325             NS <= UP_MEMx;
326           end if;
327
328         when DIN_SCK_LOz => -- valori di default, aspetto fronte di salita SCK per
            incrementare il contatore
329           if (SCK_HIx = '1') then
330             NS <= DIN_CNT_UP;
331           else
332             NS <= DIN_SCK_LOz;
333           end if;
334
335         when DIN_CNT_UP => -- incremento il contatore con TC32=0
336           CNT_EN <= '1';
337           -------------------
338           NS <= DIN_SCK_HI;
339
340         when UP_MEMx => -- memorizzo A in ADD_REG e DIN in DIN_REG
341           DIN_EN <= '1';
342           ADD_EN <= '1';
343           -------------------
344           NS <= UP_MEMy;
345
346         when UP_MEMy => -- invio il segnale di scrittura
347           WR <= '1';
348           -------------------
349           NS <= DONE;
350
351           ---- ASM_DOUT ------------------------------------------------------------
352           -- Lo slave invia i 16 bit di DOUT sul MISO
353         when DOUT_SCK_HIx => -- memorizzo A in ADD_REG
354           ADD_EN <= '1';
355           -------------------
356           NS <= DOUT_SCK_HIy;
357
358         when DOUT_SCK_HIy => -- invio il segnale di lettura, carico DOUT nel PISO, aspetto
            fronte di salita SCK per iniziare a mandare i dati sul MISO
359           RD      <= '1';
360           LD_DOUT <= '1';
361           -------------------
362           if (SCK_HIx = '1') then
```

```
363             NS <= DOUT_MSB;
364           else
365             NS <= DOUT_SCK_HIy;
366           end if;
367
368         when DOUT_MSB => -- invio MSB di DOUT sull'ingresso 0 del mux, ma lascio ancora il
            MISO in Z
369           SE_DOUT <= '1';  -- OUT_MUX = DOUT(15)
370           CNT_EN  <= '1';  -- CNT++
371           S_MISO  <= '0';  -- MISO in Z
372           -------------------
373           NS <= DOUT_MUX;
374
375         when DOUT_SHIFT =>
376           SE_DOUT <= '1'; -- OUT_MUX = DOUT(31-CNT)
377           CNT_EN  <= '1'; -- CNT++
378           S_MISO  <= '1'; -- dati su MISO
379           -------------------
380           NS <= DOUT_MUX;
381
382         when DOUT_MUX => -- invio dati su MISO; aspetto fronte di salita di SCK per fare
            un nuovo shift o TC32 per terminare la transazione
383           S_MISO <= '1';
384           -------------------
385           if (TC32 = '0' and SCK_HIx = '1') then
386             NS <= DOUT_SHIFT;
387           elsif (TC32 = '0' and SCK_HIx = '0') then
388             NS <= DOUT_MUX;
389           else
390             NS <= DOUT_LSB;
391           end if;
392
393         when DOUT_LSB => -- aspetto fronte di salita di SCK o deasserimento nSS per andare
            in DONE
394           S_MISO <= '1';
395           -------------------
396           if (nSS = '1' or SCK_HIx = '1') then
397             NS <= DONE;
398           else
399             NS <= DOUT_LSB;
400           end if;
401
402         when DONE => -- stato di done (aspetto il deasserimento di nSS)
403           if (nSS = '1') then
404             NS <= WAIT_nSS;
405           else
406             NS <= DONE;
407           end if;
408
409         when others =>
410           NS <= RESET;
411
412       end case;
413     end process controlASM;
414
415     --* STEP 2: transizioni di stato ************************************************
416     transitionsFSM : process (CK, RST)
417     begin
418       if (RST = '1') then -- reset asincrono attivo alto
```

```
419         PS <= RESET;
420       elsif (CK'event and CK = '1') then -- fronte di salita del CK
421         PS <= NS;
422       end if;
423     end process transitionsFSM;
424 
425     --* STEP 3: datapath ****************************************************************
426 
427     CMD_SR : SIPO
428     generic map(N => 8)
429     port map(clk => CK, en => SE_CMD, rst => RST_CMD_SR, d => MOSI, q => CMD_SR_OUT);
430 
431     CMD_REG : reg
432     generic map(N => 8)
433     port map(clk => CK, en => CMD_EN, rst => RST_CMD, d => CMD_SR_OUT, q => CMD_OUT);
434 
435     ADD_SR : SIPO
436     generic map(N => 8)
437     port map(clk => CK, en => SE_ADD, rst => RST_ADD_SR, d => MOSI, q => ADD_SR_OUT);
438 
439     ADD_REG : reg
440     generic map(N => 8)
441     port map(clk => CK, en => ADD_EN, rst => RST_ADD, d => ADD_SR_OUT, q => A);
442 
443     DIN_SR : SIPO
444     generic map(N => 16)
445     port map(clk => CK, en => SE_DIN, rst => RST_DIN_SR, d => MOSI, q => DIN_SR_OUT);
446 
447     DIN_REG : reg
448     generic map(N => 16)
449     port map(clk => CK, en => DIN_EN, rst => RST_DIN, d => DIN_SR_OUT, q => DIN);
450 
451     DOUT_SR : PISO
452     generic map(N => 16)
453     port map(clk => CK, en => LD_DOUT, se => SE_DOUT, rst => RST_DOUT_SR, d => DOUT, q =>
    OUT_MUX);
454 
455     COUNT : counter
456     port map(clk => CK, en => CNT_EN, rst => RST_CNT, tc8 => TC8, tc16 => TC16, tc32 =>
    TC32);
457 
458     CMD_BLOCK : command
459     port map(cmd => CMD_OUT, w_en => W_EN, r_en => R_EN);
460 
461     EDGE : clock_edge
462     generic map(N => 4)
463     port map(clk => CK, sck => SCK, rst => RST, en => EN_SCK_EDGE, sck_lox => SCK_LOx,
    sck_hix => SCK_HIx);
464 
465     EXIT_MUX : mux_z
466     port map(ingresso => OUT_MUX, S => S_MISO, uscita => MISO);
467 
468 end structure;
```

D.2.3 Top level

```
1 library ieee;
```

```
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity top_level is
6    port (
7      CK   : in std_logic;
8      SCK  : in std_logic;
9      nSS  : in std_logic;
10     RST  : in std_logic;
11     MOSI : in std_logic;
12     MISO : out std_logic
13   );
14 end entity;
15
16 architecture structure of top_level is
17
18   --*****************************************************************************
19   --* Definizione segnali interni
20   --*****************************************************************************
21
22   signal RD, WR    : std_logic;                     -- controlli
23   signal A_SPI     : std_logic_vector(7 downto 0);  -- memory address (vector)
24   signal A_SPI_INT : integer range 0 to 255;        -- memory address (integer)
25   signal D_SPI     : std_logic_vector(15 downto 0); -- ingresso memoria
26   signal Q_SPI     : std_logic_vector(15 downto 0); -- uscita memoria
27
28   --*****************************************************************************
29   --* Dichiarazione component
30   --*****************************************************************************
31
32   component spi is
33     port (
34       CK, SCK : in std_logic;
35       nSS     : in std_logic;
36       RST     : in std_logic;
37       MOSI    : in std_logic;
38       MISO    : out std_logic;
39       RD, WR  : out std_logic;
40       A       : out std_logic_vector(7 downto 0);
41       DIN     : out std_logic_vector(15 downto 0);
42       DOUT    : in std_logic_vector(15 downto 0)
43     );
44   end component;
45
46   component MEM is
47     generic (N : integer := 16);
48     port (
49       CK   : in std_logic;
50       WR   : in std_logic;
51       RD   : in std_logic;
52       ADDR : in integer range 0 to 255;
53       D    : in std_logic_vector(N - 1 downto 0);
54       Q    : out std_logic_vector(N - 1 downto 0)
55     );
56   end component;
57
58 begin
59
60   A_SPI_INT <= to_integer(unsigned(A_SPI));
```

```
61
62    SPI_INTERFACE : spi
63    port map(
64      CK   => CK,
65      SCK  => SCK,
66      nSS  => nSS,
67      RST  => RST,
68      A    => A_SPI,
69      DIN  => D_SPI,
70      DOUT => Q_SPI,
71      RD   => RD,
72      WR   => WR,
73      MOSI => MOSI,
74      MISO => MISO
75    );
76
77    MEMORY : mem
78    generic map(N => 16)
79    port map(
80      CK   => CK,
81      WR   => WR,
82      RD   => RD,
83      ADDR => A_SPI_INT,
84      D    => D_SPI,
85      Q    => Q_SPI
86    );
87
88  end structure;
```

E Test

E.1 Testbench a scopo di *debug*

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity tb_spi_simple is
6   end tb_spi_simple;
7
8   architecture behavioral of tb_spi_simple is
9
10    -- segnali di appoggio
11    signal clock   : std_logic := '0';              -- main clock
12    signal s_clock : std_logic := '0';              -- system clock
13    signal nSS_spi : std_logic := '1';              -- slave select
14    signal s_mosi  : std_logic := '1';              -- MOSI seriale
15    signal s_miso  : std_logic := 'Z';              -- MISO seriale
16    signal MOSI_wr : std_logic_vector(31 downto 0); -- vettore dati per WR
17    signal MOSI_rd : std_logic_vector(15 downto 0); -- vettore dati per RD
18    signal reset   : std_logic;                     -- reset esterno
19
20    -- dichiarazione UUT
21    component top_level is
22      port (
23        CK   : in std_logic;
24        SCK  : in std_logic;
```

```
25        nSS  : in std_logic;
26        RST  : in std_logic;
27        MOSI : in std_logic;
28        MISO : out std_logic
29      );
30    end component;
31
32  begin
33
34    -- istanza UUT
35    TB_SPI : top_level
36    port map(
37      CK => clock, SCK => s_clock,
38      nSS => nSS_spi, RST => reset,
39      MOSI => s_mosi, MISO => s_miso
40    );
41
42    -- Process CK (clock FPGA, periodo 200 ns, f=5 MHz)
43    CK_process : process
44    begin
45      wait for 50 ns;
46      clock <= not clock;
47    end process CK_process;
48
49    -- Process di lettura e scrittura
50    RD_WR_process : process
51    begin
52      reset <= '1';
53      wait for 0.1 ns;
54      reset <= '0';
55
56      -- PRIMA SCRITTURA: w01090b
57      -- CMD 00100000 '20' (w)
58      -- ADD 00000001 '01'
59      -- DIN 0000100100001011 '090b'
60      -- 00100000 00000001 0000100100001011
61      MOSI_wr <= "00100000" & "00000001" & "0000100100001011";
62      nSS_spi <= '0';
63      wait for 100 ns;
64
65      for i in 0 to 31 loop --invio dati su mosi
66        s_mosi  <= MOSI_wr(31 - i);
67        s_clock <= '1';
68        wait for 500 ns;
69        s_clock <= '0';
70        wait for 500 ns;
71      end loop;
72
73      wait for 200 ns;
74      nSS_spi <= '1';
75      wait for 400 ns;
76
77      -- SECONDA SCRITTURA: w02aaaa
78      -- CMD 00100000 '20' (w)
79      -- ADD 00000010 '02'
80      -- DIN 1010101010101010 'aaaa'
81      -- 00100000 00000010 1010101010101010
82      MOSI_wr <= "00100000" & "00000010" & "1010101010101010";
83      nSS_spi <= '0';
```

```
84      wait for 100 ns;
85
86      for i in 0 to 31 loop --invio dati su mosi
87        s_mosi  <= MOSI_wr(31 - i);
88        s_clock <= '1';
89        wait for 500 ns;
90        s_clock <= '0';
91        wait for 500 ns;
92      end loop;
93
94      wait for 200 ns;
95      nSS_spi <= '1';
96      wait for 200 ns;
97
98      -- TERZA SCRITTURA: w0509ef
99      -- CMD 00100000 '20' (w)
100     -- ADD 00000101 '05'
101     -- DIN 1010101010101010 '09ef'
102     -- 00100000 00000101 0000100111101111
103     MOSI_wr <= "00100000" & "00000101" & "0000100111101111";
104     nSS_spi <= '0';
105     wait for 100 ns;
106
107     for i in 0 to 31 loop
108       s_mosi  <= MOSI_wr(31 - i);
109       s_clock <= '1';
110       wait for 500 ns;
111       s_clock <= '0';
112       wait for 500 ns;
113     end loop;
114
115     wait for 200 ns;
116     nSS_spi <= '1';
117     wait for 400 ns;
118
119     -- PRIMA LETTURA: r02
120     -- CMD 00100001 '21' (r)
121     -- ADD 00000010 '02'
122     -- 00100001 00000010
123     MOSI_rd <= "00100001" & "00000010";
124     nSS_spi <= '0';
125     wait for 1 us;
126
127     for i in 0 to 15 loop --invio dati su mosi
128       s_mosi  <= MOSI_rd(15 - i);
129       s_clock <= '1';
130       wait for 500 ns;
131       s_clock <= '0';
132       wait for 500 ns;
133     end loop;
134
135     for i in 0 to 18 loop -- aspetto il dato
136       s_clock <= '1';
137       wait for 500 ns;
138       s_clock <= '0';
139       wait for 500 ns;
140     end loop;
141
142     nSS_spi <= '1';
```

```
143       wait for 200 ns;
144
145       -- SECONDA LETTURA: r01
146       -- CMD 00100001 '21' (r)
147       -- ADD 00000001 '01'
148       -- 00100001 00000001
149       MOSI_rd <= "00100001" & "00000001";
150       nSS_spi <= '0';
151       wait for 1 us;
152
153       for i in 0 to 15 loop
154         s_mosi  <= MOSI_rd(15 - i);
155         s_clock <= '1';
156         wait for 500 ns;
157         s_clock <= '0';
158         wait for 500 ns;
159       end loop;
160
161       for i in 0 to 18 loop
162         s_clock <= '1';
163         wait for 500 ns;
164         s_clock <= '0';
165         wait for 500 ns;
166       end loop;
167
168       nSS_spi <= '1';
169       wait for 200 ns;
170
171       -- TERZA LETTURA: r05
172       -- CMD 00100001 '21' (r)
173       -- ADD 00000101 '05'
174       -- 00100001 00000101
175       MOSI_rd <= "00100001" & "00000101";
176       nSS_spi <= '0';
177       wait for 1 us;
178
179       for i in 0 to 15 loop
180         s_mosi  <= MOSI_rd(15 - i);
181         s_clock <= '1';
182         wait for 500 ns;
183         s_clock <= '0';
184         wait for 500 ns;
185       end loop;
186
187       for i in 0 to 18 loop
188         s_clock <= '1';
189         wait for 500 ns;
190         s_clock <= '0';
191         wait for 500 ns;
192       end loop;
193
194       nSS_spi <= '1';
195       wait;
196
197     end process RD_WR_process;
198
199   end architecture;
```

E.2 Testbench con I/O da file

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity tb_spi_complete is
end tb_spi_complete;

architecture behavioral of tb_spi_complete is

    -- definizione segnali interni
    signal clock   : std_logic := '0';              -- main clock
    signal s_clock : std_logic := '0';              -- system clock
    signal nSS_spi : std_logic := '1';              -- slave select
    signal reset   : std_logic := '0';              -- reset esterno
    signal s_mosi  : std_logic := '1';              -- MOSI seriale
    signal s_miso  : std_logic := 'Z';              -- MISO seriale
    signal MOSI_wr : std_logic_vector(31 downto 0); -- vettore dati per WR
    signal MOSI_rd : std_logic_vector(15 downto 0); -- vettore dati per RD
    signal MISO_rd : std_logic_vector(15 downto 0); -- vettore restituito su MISO

    -- definizione file di I/O
    file file_WRITE  : text;
    file file_READ   : text;
    file file_OUTPUT : text;

    -- dichiarazione UUT
    component top_level is
        port (
            CK, SCK : in std_logic;
            nSS     : in std_logic;
            RST     : in std_logic;
            MOSI    : in std_logic;
            MISO    : out std_logic
        );
    end component;

begin

    -- istanza UUT
    TB_SPI : top_level
    port map(
        CK => clock, SCK => s_clock,
        RST => reset, nSS => nSS_spi,
        MOSI => s_mosi, MISO => s_miso
    );

    -- Process CK (clock FPGA, periodo 100 ns, f=10 MHz)
    CK_process : process
    begin
        wait for 50 ns;
        clock <= not clock;
    end process CK_process;

    -- Process di scerittura e lettura
```

```
    WR_RD_process : process
        variable v_WLINE : line; -- riga file comandi di scrittura
        variable v_RLINE : line; -- riga file comandi di lettura
        variable v_OLINE : line; -- riga file di output risultati
        variable v_wMOSI : std_logic_vector(31 downto 0);
        variable v_rMOSI : std_logic_vector(15 downto 0);
        variable cnt_bit : integer := 15;

    begin

        wait for 100 ns;
        reset <= '1';
        wait for 100 ns;
        reset <= '0';

        -- Apro file di I/O in modalità di lettura/scrittura
        file_open(file_WRITE, "input_wr.txt", read_mode);
        file_open(file_READ, "input_rd.txt", read_mode);
        file_open(file_OUTPUT, "output_results.txt", write_mode);

        -- Leggo da input_wr.txt
        while not endfile(file_WRITE) loop
            readline(file_WRITE, v_WLINE);
            read(v_WLINE, v_wMOSI); -- get data in

            -- scrittura
            MOSI_wr <= v_wMOSI;
            nSS_spi <= '0';
            wait for 100 ns;

            for i in 0 to 31 loop
                s_mosi  <= MOSI_wr(31 - i);
                s_clock <= '1';
                wait for 500 ns;
                s_clock <= '0';
                wait for 500 ns;
            end loop;

            wait for 200 ns;
            nSS_spi <= '1';
            wait for 5 us;

        end loop;

        -- Leggo da input_rd.txt
        while not endfile(file_READ) loop
            readline(file_READ, v_RLINE);
            read(v_RLINE, v_rMOSI); -- get data in

            -- lettura
            MOSI_rd <= v_rMOSI;
            nSS_spi <= '0';
            wait for 1 us;

            for i in 0 to 15 loop
                s_mosi  <= MOSI_rd(15 - i);
                s_clock <= '1';
                wait for 500 ns;
                s_clock <= '0';
```

```
                wait for 500 ns;
            end loop;

            for i in 0 to 18 loop
                s_clock <= '1';
                if (s_miso /= 'Z') then
                    MISO_rd(cnt_bit) <= s_miso;
                    cnt_bit := cnt_bit - 1;
                end if;
                wait for 500 ns;
                s_clock <= '0';
                wait for 500 ns;
            end loop;

            cnt_bit := 15;
            nSS_spi <= '1';
            wait for 2 us;

            -- Scrivo in output_results.txt
            write(v_OLINE, MISO_rd, right, 16);
            writeline(file_OUTPUT, v_OLINE);

        end loop;

        -- Chiudo i file di I/O
        file_close(file_WRITE);
        file_close(file_READ);
        file_close(file_OUTPUT);

        wait;
    end process WR_RD_process;

end architecture;
```

E.3 Automatizzazione della simulazione con C++

E.3.1 Classe Converter

```
#include <iostream>
#include <fstream> // per il file processing
#include <string>  // per creazione e manipolazione stringhe
#include <cstring> // per manipolazione di stringhe C-like
#include <cmath>   // per l'elevazione a potenza con pow(a,b)

#include "Converter.hpp"

using namespace std;

// Costruttore
Converter::Converter(int n, int r)
    : number{n}, result{} {}

// Distruttore
Converter::~Converter() {}

// Converte un numero da intero a binario (con parallelismo a n bit) e scrive il
↪  risultato in un file
```

```
void Converter::intToBin(string conv_number, unsigned int n, ofstream &outFile)
{
    int digit;                  // singola cifra del numero da convertire
    number = stoi(conv_number); // numero (intero) da convertire

    // Sfruttiamo l'overloading dell'operatore >> (shift right)
    // Ad ogni iterazione shiftiamo number a dx di i posizioni (ovvero calcoliamo number
    ↪  % 2^i) e mettiamo il risultato in bitwise and con 1
    for (int i = n - 1; i >= 0; i--)
    {
        digit = (number >> (i)) & 1;
        outFile << digit;
    }
}

// Converte un numero da binario a intero
int Converter::binToInt(string conv_number)
{
    result = 0;
    int bit_number = conv_number.length();
    for (int i = 0; i < bit_number; i++)
    {
        string bit = conv_number.substr(i, 1); // estraggo l'i-esimo bit
        int bit_int = stoi(bit);               // trasformo il bit in un numero intero
        result += pow(2, bit_number - 1 - i) * bit_int;
    }
    return result;
}
```

E.3.2 Classe Simulation

```
#include <iostream>
#include <fstream>    // per file processing
#include <sstream>    // per trattare le stringhe come stream di dati
#include <filesystem> // per verificare l'esistenza dei file
#include <cstdlib>    // per usare comandi shell
#include <string>     // per manipolazione stringhe
#include <cstring>    // per manipolazione stringhe C-like
#include <vector>     // per manipolazione vettori
#include <cmath>      // per funzioni matematiche
#include <iomanip>    // per formattazione I/O

#include "Converter.hpp"
#include "Simulation.hpp"

using namespace std;

// Costruttore
Simulation::Simulation(unsigned int c)
    : correct{c} {}

// Distruttore
Simulation::~Simulation() {}

// Genero file per effettuare scritture e letture
// @param wFName = nome file contenente comandi di scrittura
// @param rFName = nome file contenente comandi di lettura
```

```
27 // @param ref_FName = nome file contenente i dati generati (per confronto con i
   ↪  risultati di Modelsim)
28 void Simulation::generate(string wFName, string rFName, string ref_FName)
29 {
30     Converter C; // oggetto convertitore
31 
32     // check esistenza file (se non esistono li creo con una chiamata a system)
33     if (!filesystem::exists(wFName))
34         system(("touch " + wFName).c_str());
35     if (!filesystem::exists(rFName))
36         system(("touch " + rFName).c_str());
37     if (!filesystem::exists(ref_FName))
38         system(("touch " + ref_FName).c_str());
39 
40     // apro i file
41     ofstream wF(wFName);
42     ofstream rF(rFName);
43     ofstream ref_F(ref_FName);
44 
45     // scrivo i comandi di scrittura e lettura generando casualmente i numeri di 16 bit
   ↪  da memorizzare
46     if (wF && rF && ref_F) // se l'apertura è andata a buon fine
47     {
48         for (int add = 0; add < 256; add++)
49         {
50             int din = rand() % 65536; // dato da scrivere
51 
52             // scrivo comandi di scrittura
53             C.intToBin(to_string(32), 8, wF);   // comando di scrittura
54             C.intToBin(to_string(add), 8, wF);  // indirizzo
55             C.intToBin(to_string(din), 16, wF); // dato
56             wF << endl;
57 
58             // scrivo file di riferimento
59             C.intToBin(to_string(din), 16, ref_F);
60             ref_F << endl;
61 
62             // scrivo comandi di lettura
63             C.intToBin(to_string(33), 8, rF);  // comando di lettura
64             C.intToBin(to_string(add), 8, rF); // indirizzo
65             rF << endl;
66         }
67 
68         // chiudo i file
69         wF.close();
70         rF.close();
71         ref_F.close();
72     }
73 }
74 
75 // Esecuzione simulazione mediante chiamata a system
76 void Simulation::run(string fileCompilazione)
77 {
78     system(("vsim -c -do " + fileCompilazione).c_str()); // lancio la simulazione
79 }
80 
81 // Controlla la correttezza dei risultati
82 unsigned int Simulation::report(string risultati_tb, string risultati_ref)
83 {
```

```
84      string line_tb, line_ref; // righe dei due file
85      int cnt_lines_tb = 0;     // contatore di riga del file generato dalla tb
86      int cnt_lines_ref = 0;    // contatore di riga del file di riferimento
87      int tot_correct = 0;      // numero totale di righe corrette all'interno del file
        ↪  generato dalla tb
88
89      // apro i file in lettura
90      ifstream tbF(risultati_tb);
91      ifstream ref_F(risultati_ref);
92
93      while (tbF.good() && ref_F.good())
94      {
95          // estraggo una riga da ognuno dei due file
96          if (getline(tbF, line_tb) && getline(ref_F, line_ref))
97          {
98              // se i risultati della tb e del file di riferimento sono uguali incremento
                ↪  il contatore di righe corrette
99              if (line_tb == line_ref)
100             {
101                 tot_correct++;
102             }
103             // se i risultati sono diversi, esco dal ciclo (non ho più bisogno di
                ↪  controllare le righe restanti)
104             else
105             {
106                 cout << "Errore durante la transazione effettuata all'indirizzo " <<
                    ↪  cnt_lines_tb << endl;
107                 break;
108             }
109
110             // incremento i contatori di riga
111             cnt_lines_tb++;
112             cnt_lines_ref++;
113         }
114     }
115
116     // chiudo i file
117     tbF.close();
118     ref_F.close();
119
120     // stabilisco il valore del flag che mi dice se la simulazione è andata a buon fine
121     if ((cnt_lines_tb == cnt_lines_ref) && (tot_correct == cnt_lines_ref))
122         correct = 1;
123     else
124         correct = 0;
125
126     return correct;
127 }
```

E.3.3 Main

```
1  #include <iostream>
2  #include <fstream>    // per file processing
3  #include <sstream>    // per trattare le stringhe come stream di dati
4  #include <filesystem> // per verificare l'esistenza dei file
5  #include <cstdlib>    // per usare comandi shell
6  #include <string>     // per manipolazione stringhe
```

```cpp
#include <cstring>    // per manipolazione stringhe C-like
#include <vector>     // per manipolazione vettori
#include <cmath>      // per funzioni matematiche
#include <iomanip>    // per formattazione I/O

#include "Converter.hpp"
#include "Simulation.hpp"

using namespace std;

int main(int argc, char **argv)
{

    int ret = 0;
    Simulation Simulator;

    /*****************************************************************************/
    /*        Inizializzazione degli oggetti necessari alla gestione dei file    */
    /*****************************************************************************/

    const string tbFileName = "tb_spi_complete.vhd";       // testbench
    const string compileFileName = "compile.do";           // file con le info per la
    ↪  simulazione
    const string writeFileName = "input_wr.txt";           // file con i comandi di
    ↪  scrittura
    const string readFileName = "input_rd.txt";            // file con i comandi di
    ↪  lettura
    const string ref_oFileName = "output_results_ref.txt"; // file dati per confronto
    const string tb_oFileName = "output_results.txt";      // file di output generato
    ↪  dalla testbench

    // check esistenza testbench
    if (!filesystem::exists(tbFileName))
    {
        cerr << "Errore! La testbench " << tbFileName << " non esiste." << endl;
        ret = 1;
    }

    // check esistenza file per la compilazione
    if (!filesystem::exists(compileFileName))
    {
        cerr << "Errore! Il file per la compilazione " << compileFileName << " non
        ↪  esiste." << endl;
        ret = 1;
    }

    /*****************************************************************************/
    /*                            Simulazione automatizzata                      */
    /*****************************************************************************/

    // generazione file di scrittura
    Simulator.generate(writeFileName, readFileName, ref_oFileName);

    // simulazione automatizzata
    cout << endl
         << "*************************************************************" <<
         ↪  endl;
    cout << "Inizio Simulazione Modelsim" << endl;
    Simulator.run(compileFileName);
```

```
60      cout << "Fine Simulazione Modelsim" << endl;
61      cout << "**********************************************************************" <<
        ↪  endl
62           << endl;
63
64      // controllo risultati
65      int simulazione_corretta = Simulator.report(ref_oFileName, tb_oFileName);
66      if (simulazione_corretta == 1)
67      {
68          cout << "OK! Tutte le transazioni di lettura e scrittura sono andate a buon
            ↪  fine." << endl;
69          cout << "Verosimilmente l'interfaccia SPI funziona correttamente! :)" << endl
70               << endl;
71      }
72      else
73      {
74          cout << endl
75               << "Non tutte le transazioni di lettura hanno prodotto risultati uguali ai
                 ↪  dati trasmessi in scrittura." << endl;
76          cout << "L'interfaccia SPI non funziona. :(" << endl
77               << endl;
78      }
79
80      return ret; // punto di uscita dal programma
81  }
```

E.4 Test su VirtLab

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4   library lpm;
5   use lpm.lpm_components.all;
6   library altera_mf;
7   use altera_mf.altera_mf_components.all;
8
9   entity user is
10    port (
11      -- Main clock inputs
12      mainClk : in std_logic;
13      slowClk : in std_logic;
14      -- Main reset input
15      reset : in std_logic;
16      -- MCU interface (UART, I2C)
17      mcuUartTx : in std_logic;
18      mcuUartRx : out std_logic;
19      mcuI2cScl : in std_logic;
20      mcuI2cSda : inout std_logic;
21      -- Logic state analyzer/stimulator
22      lsasBus : inout std_logic_vector(31 downto 0);
23      -- Dip switches
24      switches : in std_logic_vector(7 downto 0);
25      -- LEDs
26      leds : out std_logic_vector(3 downto 0)
27    );
28  end user;
29
```

```
30  architecture behavioural of user is
31
32    signal clk     : std_logic;
33    signal pllLock : std_logic;
34
35    signal lsasBusIn  : std_logic_vector(31 downto 0);
36    signal lsasBusOut : std_logic_vector(31 downto 0);
37    signal lsasBusEn  : std_logic_vector(31 downto 0) := (others => '0');
38
39    signal mcuI2cDIn  : std_logic;
40    signal mcuI2CDOut : std_logic;
41    signal mcuI2cEn   : std_logic := '0';
42
43    component myAltPll
44      port (
45        areset : in std_logic := '0';
46        inclk0 : in std_logic := '0';
47        c0     : out std_logic;
48        locked : out std_logic
49      );
50    end component;
51
52    component top_level is
53      port (
54        CK   : in std_logic;
55        SCK  : in std_logic;
56        nSS  : in std_logic;
57        RST  : in std_logic;
58        MOSI : in std_logic;
59        MISO : out std_logic
60      );
61    end component;
62
63  begin
64
65    --*************************************************************************
66    --* Main clock PLL
67    --*************************************************************************
68
69    myAltPll_inst : myAltPll port map(
70      areset => reset,
71      inclk0 => mainClk,
72      c0     => clk,
73      locked => pllLock
74    );
75
76    --*************************************************************************
77    --* LEDs
78    --*************************************************************************
79
80    -- accendo il led 3 per verificare il corretto caricamento del .rbf
81    leds(3) <= '1';
82
83    -- collego i led restanti ai rispettivi switch
84    leds(2 downto 0) <= switches(2 downto 0);
85
86    --*************************************************************************
87    --*    lsasBus  : inout std_logic_vector( 31 downto 0 )
88    --*************************************************************************
```

```
89
90    lsasBusIn      <= lsasBus;
91    lsasbusEn (14) <= '1';
92
93    lsasBus_tristate :
94    process (lsasBusEn, lsasBusOut) is
95    begin
96      for index in 0 to 31 loop
97        if lsasBusEn(index) = '1' then
98          lsasBus(index) <= lsasBusOut (index);
99        else
100         lsasBus(index) <= 'Z';
101       end if;
102     end loop;
103   end process;
104
105   spi : top_level port map(
106     CK   => mainClk,
107     SCK  => lsasbus(13),
108     nSS  => lsasbus(12),
109     RST  => switches(0),
110     MOSI => lsasbus(15),
111     MISO => lsasbusOut(14)
112   );
113
114 end behavioural;
```