# POLITECNICO DI TORINO

## Corso di laurea MAGISTRALE in INGEGNERIA ELETTRONICA(ELECTRONICS ENGINEERING)


## Tesi di laurea MAGISTRALE

## Sviluppo di IP su FPGA per la realizzazione di architetture modulari sulla scheda VirtLAB

**Relatore**

Prof. Massimo RUO ROCH

**Candidato**

**Gianfranco SARCIÀ**

**11 APRILE 2025**

**Sviluppo di IP su FPGA per la realizzazione di architetture modulari sulla scheda VirtLAB**

**Gianfranco Sarcià**

## Sommario

Questa tesi, intitolata "Sviluppo di IP su FPGA per la realizzazione di architetture modulari sulla scheda VirtLAB", ha come scopo quello di mostrare la realizzazione di un generatore di pattern digitali e un analizzatore di stati logici.

Nella prima sezione, viene presentato il progetto di un blocco di traduzione tra l'interfaccia SPI e l'interfaccia Avalon memory-mapped. Questo componente è stato successivamente collegato a una memoria, consentendo di eseguire operazioni di scrittura e lettura dei dati tramite un terminale configurato per l'interfaccia SPI.

La seconda sezione si concentra sullo sviluppo di un generatore di pattern digitali. Questo componente ha il compito di generare sequenze digitali a seconda delle impostazioni scelte dall'utente. Utilizzando il terminale SPI progettato nella prima sezione, è possibile infatti scrivere i registri di stato e controllo per la configurazione del generatore. Il generatore di pattern è fondamentale per stimolare altri componenti ed effettuare, ad esempio, verifiche sul comportamento di essi in funzione degli stimoli ricevuti.

Infine, la terza sezione mostra la realizzazione di un analizzatore di stati logici, progettato per campionare ed analizzare i segnali digitali in ingresso. Anche in questo progetto è stato utilizzato il terminale SPI per la configurazione dei registri di stato e controllo. L'analizzatore è in grado di acquisire e campionare i segnali in modo efficace, mostrando la presenza di eventuali glitch, e di salvarne il valore in memorie dedicate.

*A chi ha sempre creduto in me*

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Capitolo 1

# Introduzione

## 1.1 Introduzione alla scheda VirtLAB

### 1.1.1 Contesto storico e relative esigenze

A partire dalla primavera del 2020, la pandemia di SARS-CoV2 ha colpito la Cina, l'Italia ed il resto del mondo. Le contromisure di lockdown sono state necessarie per mitigare la diffusione del virus tra la popolazione, ma ciò ha comportato la sospensione didattica "in presenza" nelle scuole di ogni ordine e grado, comprese le università. Le difficoltà legate alla didattica sono state superate grazie all'utilizzo di internet, sfruttandone la capacità di effettuare videolezioni, videoconferenze e laboratori virtuali. I corsi di ingegneria elettronica sono candidati naturali nell'utilizzo di questi strumenti. Ad esempio, nei corsi di elettronica analogica e digitale, gli studenti progettano circuiti tramite linguaggi di descrizione dell'hardware(VHDL, Verilog, SystemC). i quali possono essere simulati utilizzando interfacce web verso strumenti di simulazione standard (SPICE, ModelSim). Ciò che però manca in questo approccio è il contatto con gli oggetti del mondo reale, fondamentali per l'acquisizione di competenze nel mondo ingegneristico. Sviluppare capacità progettuali richiede l'acquisizione delle seguenti abilità:

- Esplorazione dello spazio di progetto
- Simulazione del sistema progettato
- Verifica di conformità con il modello di alto livello precedentemente definito
- Verifica hardware
- Caratterizzazione del sistema hardware

I primi tre punti possono essere facilmente realizzati con le metodologie basate sul web descritte in precedenza. Tuttavia, gli ultimi due punti non erano ai tempi accessibili, pur essendo fondamentali. È necessario porre l'attenzione sul fatto che la simulazione non può sostituire completamente le misure effettuate sul circuito reale.

Inoltre, la capacità di diagnosticare guasti su circuiti reali è una competenza ingegneristica di alto valore che deve essere perseguita. Inoltre, richiede competenze sia nell'uso di strumenti di misurazioni reali, sia nello sviluppo personale di metodologie di ricerca dei guasti. Infine, i segnali reali sono solitamente diversi da quelli simulati, in quanto affetti da rumori o interferenze varie. Una possibile soluzione a queste esigenze sarebbe quella di dare agli studenti accesso a dispositivi fisici su cui svolgere le attività di laboratorio. In questo contesto esistono tre possibilità:

- Le università acquistano un "kit di laboratorio" e lo forniscono agli studenti. Questa soluzione però presenta problemi relativi ai costi e alle modalità di consegna e ritiro del materiale a fine corso.
- Le università richiedono agli studenti di acquistare di tasca propria il "kit di laboratorio". Anche in questo caso il problema principale è quello relativo al costo.
- Le università installano strumenti di misurazione in loco e forniscono allo studente la possibilità di interagire via internet. Tuttavia in questo caso lo studente non avrebbe accesso fisico ai componenti.

Il motivo principale per cui non è stata trovata una soluzione sul mercato è legato al fatto che questi kit non sono pensati per l'uso didattico. Pertanto, presentano delle specifiche elevate e di conseguenza un costo elevato. L'idea di base per risolvere questa problematica risiede quindi nel progettare un kit che abbia delle specifiche mirate alla risoluzione dei laboratori didattici con costi minimi. Proprio in questo contesto il Politecnico di Torino ha sviluppato la scheda "VirtLAB", in grado di soddisfare questi requisiti fondamentali. [1]

### 1.1.2 Architettura della scheda VirtLAB

La scheda VirtLAB è divisa in due sezioni principali:

- "Area Master"
- "Area User"

L'area "Master" è progettata per svolgere le funzioni di un tipico banco di laboratorio, integrando le funzionalità di un oscilloscopio a memoria digitale(DSO), un multimetro standard, un generatore di segnali analogici programmabili e un analizzatore di stati logici. L'area "User" è invece l'equivalente di una scheda prototipo sulla quale gli studenti possono svolgere l'esperimento stesso. La scheda può essere facilmente collegata ad un sistema di elaborazione (PC) dove verranno eseguite le attività di elaborazione ad alto costo. E' proprio l'integrazione di hardware e software in esecuzione sul dispositivo che permette di ridurre i costi al minimo.

L'architettura base della scheda è rappresentata nella figura 1.1.

Il microcontrollore della parte "master", un $STM32L496VETx$, rappresenta il controllore dell'intera scheda ed ha diversi modi di comunicare con gli altri componenti:

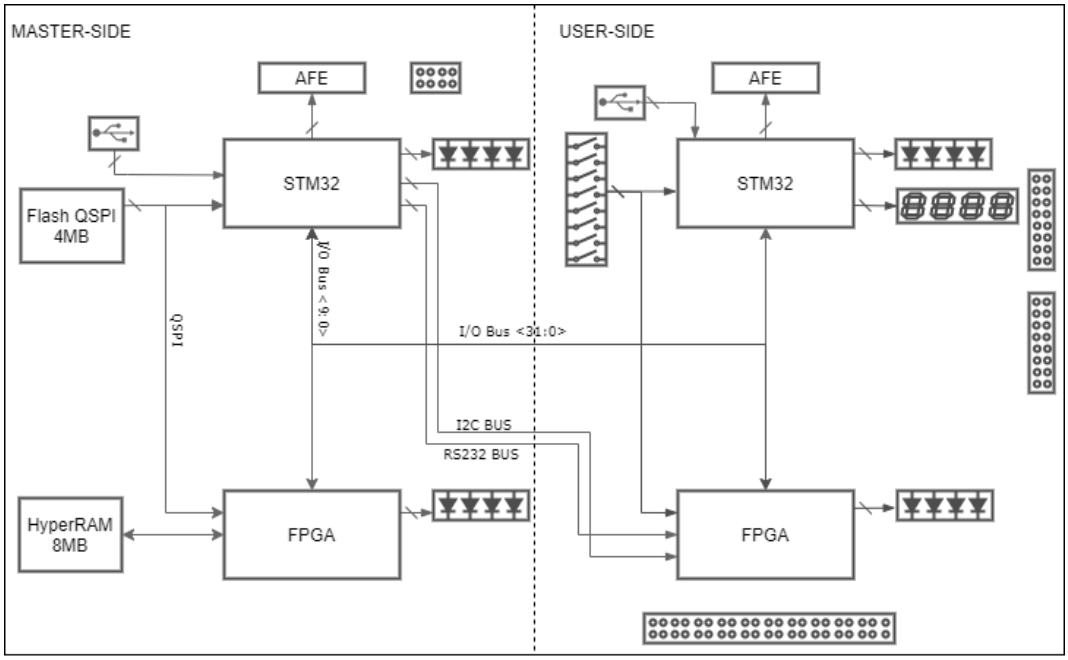


**Figura 1.1:** Schema a blocchi della VirtLAB

- Interfaccia $QSPI$ per la comunicazione con l'FPGA Master.
- Bus $I^2C$ o la comunicazione seriale $RS232$ per la comunicazione con la FPGA User.

Dell'apparecchiatura di test, il DSO (Digital Storage Oscilloscope) è stato implementato sfruttando il Front-End Analogico dell'$STM32$, mentre il generatore di segnali mediante il DAC (Digital to Analog Converter) presente nel $\mu C$ stesso.

L'FPGA "Master" è invece collegata al bus I/O a 32 bit, ed è quindi in grado di campionare segnali sul bus e inviarli al $\mu C$ master. I segnali campionati dall'FPGA Master e inviati a MCU-Master vengono trasferiti al PC sfruttando la connessione USB. In questo modo l'utente è in grado di osservare la forma d'onda dei segnali nel PC.

Il microcontrollore lato "User" è liberamente programmabile tramite $CubeMX$, quindi sfruttando la connessione USB e il PC. In questo modo, gli utenti sono in grado di caricare il file eseguibile sulla scheda e testare la funzionalità del progetto.

L'FPGA "User" è programmabile utilizzando il file .RBF sfruttando la connessione USB e l'interfaccia utente grafica per verificare la funzionalità.

Nella scheda VirtLab sono presenti 2 diverse memorie:

- Hyper RAM, in cui vengono memorizzati i campioni dell'analizzatore di stati logici e dell'oscilloscopio.
- Flash QSPI, che ricopre un ruolo fondamentale nel sistema. Ogni volta che la scheda VirtLab viene spenta, la configurazione dell'FPGA Master viene persa. All'avvio, il microcontrollore master trasferirà il contenuto della memoria all'interno dell'FPGA Master.

Per quanto riguarda le periferiche di I/O, sono presenti sia nel lato "master" che nel lato "user". In particolare, nel lato "user" sono presenti:

- 4 display a 7 segmenti connessi al µC;
- 4 led verdi connessi al µC;
- 4 led verdi connessi all'FPGA;
- 8 led rossi connessi direttamente ad 8 interruttori;

Nel lato "master" troviamo invece:

- 4 led verdi connessi al µC;
- 4 led verdi connessi all'FPGA;

Infine è presente un bus di I/O a 32 bit, connesso fisicamente a 32 pin, condiviso da entrambe le sezioni della scheda.

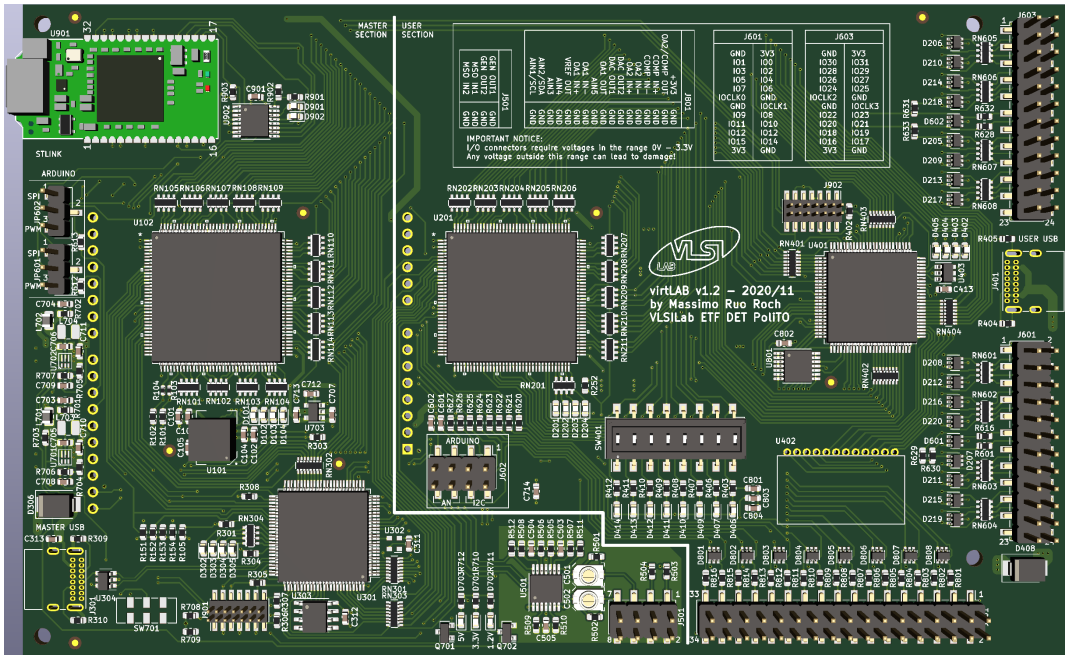


**Figura 1.2:** Scheda VirtLAB

La struttura fisica della scheda è mostrata nella figura 1.2 [2]

## 1.2 Obiettivi

Lo scopo di questa tesi è quello di implementare due IP sull'FPGA "Master" per la realizzazione di architetture modulari sulla scheda VirtLAB.

Il primo obiettivo consiste nella progettazione di un blocco di traduzione che consenta la comunicazione tra l'interfaccia SPI e l'interfaccia Avalon Memory-Mapped. Tale blocco risulta fondamentale per la comunicazione tra il $\mu C$, che si interfaccia all'esterno mediante l'utilizzo dell'SPI, e l'FPGA, che utilizza invece l'interfaccia

Avalon Memory-Mapped. Di fatto, questo blocco di traduzione è stato utilizzato per leggere e scrivere all'interno di una memoria RAM progettata mediante l'utilizzo del software *Platform Designer*. La lettura e la scrittura dei dati in memoria avviene mediante un terminale configurato per ricevere ed inviare i dati mediante l'interfaccia SPI.

Il "Bridge SPI to Avalon" funge da lavoro preliminare per il cuore di questa tesi, ovvero il progetto del generatore di pattern digitali e dell'analizzatore di stati logici.

Il lavoro relativo al generatore di pattern digitali prevede la progettazione di un modulo in grado di generare segnali digitali specifici a seconda dei requisiti applicativi. Utilizzando un terminale SPI, i registri di stato e controllo vengono scritti per la configurazione del generatore, che successivamente produrrà un'ampia varietà di pattern. Questa funzionalità è cruciale per testare e validare il comportamento di altri moduli nel sistema, garantendo così un elevato grado di versatilità nell'uso della FPGA.

Infine, l'ultimo obiettivo di questa tesi è la realizzazione dell'analizzatore di stati logici, progettato per monitorare e analizzare i segnali digitali in tempo reale. Anche in questo caso, sono state utilizzate tecniche simili a quelle utilizzate per il generatore di pattern, consentendo al terminale SPI di interagire con il modulo per la configurazione dei registri di stato e controllo. L'analizzatore è in grado di acquisire e visualizzare i dati in modo efficace, facilitando la diagnosi e il debug di sistemi digitali complessi.

# Capitolo 2

# Bridge da interfaccia SPI ad interfaccia Avalon Memory-Mapped

## 2.1 Descrizione dell'interfaccia SPI

L'interfaccia SPI (Serial Peripheral Interface) è una delle più utilizzate tra microcontrollori e circuiti integrati. L'SPI è un'interfaccia sincrona, full-duplex e descrive una comunicazione di tipo Master/Slave. I dati inviati dal master e dallo slave possono essere inviati simultaneamente e vengono sincronizzati sul fronte di salita o di discesa del clock.

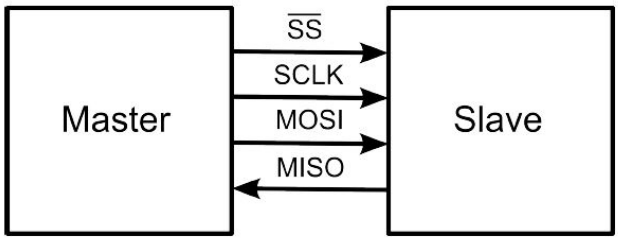


**Figura 2.1:** Schema base di collegamento tra periferiche con interfaccia SPI

Come si evince dalla figura 2.6, i dispositivi SPI utilizzano i seguenti segnali:

- Clock ($SCLK$)
- Chip Select o Slave Select ($CS$ o $SS$)
- Master Out, Slave In ($MOSI$)
- Master In, Slave Out ($MISO$)

Il dispositivo che genera il segnale di *clock* è chiamato master ed i dati trasmessi tra il master e lo slave vengono sincronizzati su questo clock. Le interfacce SPI presentano un solo Master, il quale può interfacciarsi con uno o più slave.

Il segnale di *chipselect*, proveniente dal master, viene utilizzato per selezionare lo slave con cui interfacciarsi. Questo è solitamente un segnale attivo basso e viene portato alto per disconnettere gli slave non utilizzati dal bus SPI.

I segnali di $MOSI$ e $MISO$ sono invece le linee di dato. Nello specifico il $MOSI$ trasmette i dati dal master verso lo slave, invece il $MISO$ trasmette i dati dallo slave al master. Il modo più semplice per collegare un master a più slave è quello rappresentato in figura 2.2:

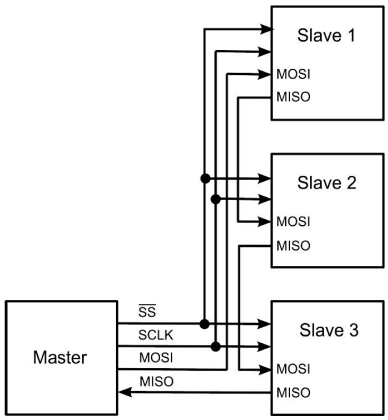


**Figura 2.2:** Architettura con un master e tre slave

Con questa architettura, è richiesto un *chipselect* individuale per ogni slave. Una volta che questo segnale viene abilitato dal master, il *clock* e i dati sulle linee $MOSI/MISO$ diventano disponibili per lo slave selezionato. Se venissero abilitati più slave contemporaneamente, i dati sulla linea $MISO$ verrebbero corrotti poiché il master non ne capirebbe la provenienza.

Per avviare una comunicazione SPI, il master deve inviare il segnale di *clock* e selezionare lo slave abilitando il segnale di *slaveselect* corrispondente. Durante la comunicazione SPI, i dati vengono trasmessi simultaneamente (spostati in serie sul bus $MOSI$) e ricevuti (letti o campionati sul bus $MISO$). Il fronte del clock seriale sincronizza la trasmissione e il campionamento dei dati.

L'interfaccia SPI fornisce all'utente la flessibilità di selezionare il fronte di salita o di discesa del clock per campionare o spostare i dati. La polarità e la fase del clock vengono selezionate dal master. Il bit CPOL imposta la polarità del clock durante lo stato di inattività, il quale è definito come il periodo in cui *ChipSelect* è alto e passa a basso all'inizio della trasmissione, e quando il *ChipSelect* è basso e passa a alto alla fine della trasmissione. Il bit CPHA seleziona invece la fase del clock. A seconda del valore del bit CPHA, viene utilizzato il fronte di salita o di discesa del *clock* per campionare o spostare i dati. Il master deve selezionare la polarità e la fase del *clock* in base ai requisiti dello slave. A seconda della selezione dei bit CPOL e CPHA, sono disponibili quattro modalità SPI, descritte nella tabella 2.1.

| **Modalità SPI** | **CPOL** | **CPHA** | **Polarità del Clock nello stato di Idle** | **Descrizione** |
|---|---|---|---|---|
| 0 | 0 | 0 | Logica Bassa | Dati campionati sul fronte di salita e trasmessi su quello di discesa |
| 1 | 0 | 1 | Logica Bassa | Dati campionati sul fronte di discesa e trasmessi su quello di salita |
| 2 | 1 | 0 | Logica Alta | Dati campionati sul fronte di discesa e trasmessi su quello di salita |
| 3 | 1 | 1 | Logica Alta | Dati campionati sul fronte di salita e trasmessi su quello di discesa |

**Tabella 2.1:** Modalità SPI con CPOL e CPHA

In Figura 2.3 è riportato il diagramma temporale nel caso in cui CPHA=1, mentre in Figura 2.4 il diagramma temporale nel caso in cui CPHA=0.

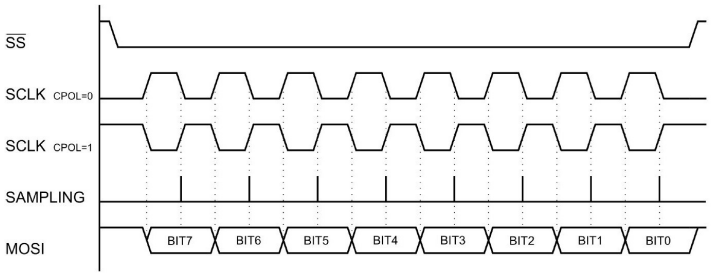


**Figura 2.3:** Diagramma temporale con CPHA=1

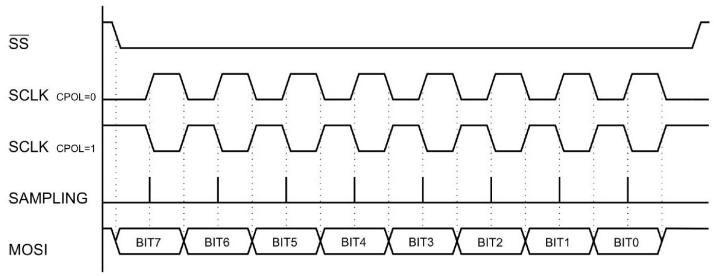


**Figura 2.4:** Diagramma temporale con CPHA=0

[3] [4]

## 2.2 Descrizione dell'interfaccia Avalon Memory-Mapped

L'interfaccia Avalon Memory-Mapped (Avalon MM) viene utilizzata per implementare un'interfaccia di lettura e scrittura tra componenti Master e Slave. I componenti che tipicamente utilizzano un'interfaccia Memory-Mapped sono:

- Microprocessori
- Memorie
- UART (Universal Asynchronous Receiver-Transmitter)
- DMA (Direct Memory Access)
- Timer

L'interfaccia Avalon Memory Mapped può essere più o meno complessa in funzione delle specifiche delle periferiche. Nel caso studiato, è stata utilizzata l'interfaccia base, in quanto il fine era la lettura e la scrittura di dati all'interno di una memoria. Il ruolo assegnato ai segnali definisce la tipologia di porte che il master e lo slave devono avere. Questa specifica è fondamentale, in quanto permette di non assegnare necessariamente tutti i segnali esistenti dell'interfaccia MM, ma solamente quelli necessari. I requisiti minimi per un'interfaccia Avalon MM sono il segnale *readdata* per un'interfaccia di sola lettura, oppure *writedata* e *write* per un'interfaccia di sola scrittura. La tabella 2.2 mostra i segnali base utilizzati da un'interfaccia MM.

| **Ruolo del segnale** | **Larghezza** | **Direzione** | **Descrizione** |
|---|---|---|---|
| *address* | 1-64 bit | M $\rightarrow S$ | Il segnale di *address* rappresenta l'indirizzo su cui leggere/scrivere i dati. |
| *read*/*read_n* | 1 | M $\rightarrow S$ | Utilizzato per indicare una lettura. Se presente, è obbligatoria la presenza del segnale *readdata* |
| *readdata* | 8,16,..,1024 | S $\rightarrow M$ | Il segnale *readdata* è un segnale che va dallo slave verso il master in risposta ad una richiesta di lettura. |
| *write*/*write_n* | 1 | M $\rightarrow S$ | Utilizzato per indicare una scrittura. Se presente, è obbligatoria la presenza del segnale *writedata* |
| *writedata* | 8,16,..,1024 | M $\rightarrow S$ | Il segnale *writedata* rappresenta il dato da scrivere durante una scrittura. La larghezza deve essere la stessa del segnale *readdata* se entrambi presenti |

| | | | |
|---|---|---|---|
| *waitrequest/ waitrequest_n* | 1 | M $\rightarrow$ *S* | Lo slave asserisce il segnale di *waitrequest* quando non disponibile ad assecondare la richiesta di lettura o scrittura. Questo segnale forza il master ad attendere fin quando l'interconnessione è disponibile a procedere con il trasferimento |

**Tabella 2.2:** Segnali basilari dell'interfaccia Avalon Memory Mapped

L'interfaccia Avalon Memory Mapped è di tipo sincrono. Di fatto, tutte le interfacce sono sincronizzate al *clock* associato. I segnali possono anche essere combinatori, a condizione che siano però pilotati dall'uscita del registro in modo sincrono al segnale di *clock*.

Come già accennato, un trasferimento è un'operazione di lettura o scrittura di una parola o uno o più simboli di dato. Questo può richiedere uno o più cicli di *clock* per essere completato. Sia il master che gli slave fanno parte del trasferimento: il master lo avvia e lo slave risponde.

Nel caso in cui il segnale *waitrequest* fosse presente, lo slave può bloccare l'interconnessione per tutti i cicli richiesti dall'asserzione del segnale. Lo slave generalmente riceve i segnali (*read*, *address*, *write* e *writedata*) dopo il fronte di salita del *clock*. Lo slave attiva il segnale *waitrequest* prima del fronte di salita del *clock* per ritardare i trasferimenti. Mentre *waitrequest* resta attivo, *address* e tutti gli altri segnali di controllo resteranno costanti. I trasferimenti vengono completati sul fronte di salita del primo colpo di *clock* dopo che lo slave ha disabilitato *waitrequest*. Non esiste alcun limite al tempo per cui l'interfaccia può rimanere in stallo. E' necessario quindi assicurarsi che uno slave non abiliti il segnale *waitrequest* a tempo indeterminato.

La figura 2.5 mostra le operazioni di lettura e scrittura utilizzando il segnale *waitrequest*.

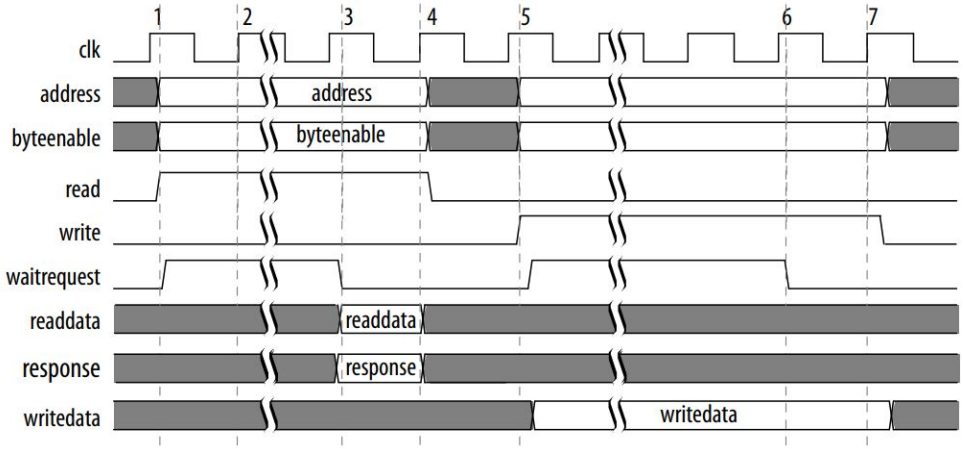


**Figura 2.5:** Timing relativo ad una lettura ed una scrittura con *waitrequest*

I numeri presenti in Figura 2.5 evidenziano le seguenti transizioni:

1. *Address* e *read* vengono attivati dopo il fronte di salita del *clock*. Lo slave attiva *waitrequest*, bloccando il trasferimento;
2. Il segnale *wairequest* viene campionato. Essendo attivo, il ciclo passa in uno stato di attesa. I segnali *address*, *read* e *write* rimangono costanti;
3. Lo slave disabilita *waitrequest* dopo il fronte di salita del *clock* e fornisce il segnale *readdata*;
4. Il master campiona *readdata* e *waitrequest*, adesso disabilitato, e completa il trasferimento;
5. *Address*, *write* e *writedata* vengono attivati dopo il fronte di salita del *clock*. Lo slave attiva *waitrequest*, bloccando il trasferimento;
6. Lo slave disabilita *waitrequest* dopo il fronte di salita del *clock*;
7. Lo slave campiona *writedata* terminando il trasferimento.

[5]

## 2.3 Architettura del componente "Bridge SPI to Avalon"

Lo scopo del progetto è quello di creare un "Bridge da SPI ad Avalon", cioè una sorta di blocco-traduttore che abbia in ingresso i segnali tipici dell'interfaccia SPI ed in uscita quelli tipici dell'interfaccia Avalon Memory-Mapped. L'obiettivo è quello di mettere in comunicazione il $\mu C$ con una memoria, affinché possa effettuare operazioni di lettura e scrittura su di essa. Considerando la teoria relativa all'interfaccia SPI e l'architettura della scheda VirtLAB, possiamo definire il $\mu C$ della scheda come il master di un sistema SPI, mentre il "Bridge SPI to Avalon" come lo slave con il quale il master si interfaccia. Per quanto riguarda l'interfaccia di uscita del "Bridge SPI to Avalon" invece, è possibile considerarla come un'interfaccia Master Avalon Memory-Mapped e la RAM il relativo slave.

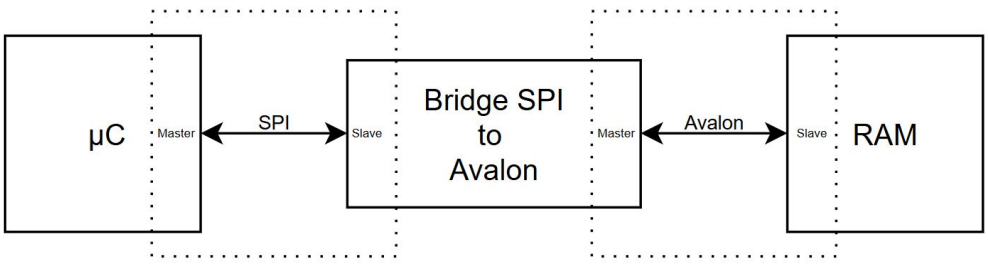


**Figura 2.6:** Rapporto master/slave tra $\mu C$, Bridge e RAM

### 2.3.1 Specifiche di progetto

Poiché l'interfaccia SPI non definisce il significato dei singoli bit trasmessi, è necessario definire il formato dell'informazione che transita sul bus $MOSI$ dell'SPI. Le istruzioni da realizzare sono:

- Scrittura: Il master SPI seleziona la periferica ed invia le seguenti informazioni sulla linea $MOSI$:
  - CMD: 8 bit, che rappresentano il comando da eseguire. Nel caso della scrittura, questi 8 bit assumono il valore 0x20 in esadecimale.
  - ADDR: 32 bit, che rappresentano l'indirizzo del registro che si desidera scrivere.
  - DATA: 32 bit, che rappresentano il dato che dovrà essere scritto all'indirizzo selezionato.

  Alla fine della trasmissione di questi bit, il master deseleziona la periferica.
- Lettura: Il master SPI seleziona la periferica ed invia le seguenti informazioni sulla linea $MOSI$:
  - CMD: 8 bit, che rappresentano il comando da eseguire. Nel caso della lettura, questi 8 bit assumono il valore 0x21 in esadecimale.
  - ADDR: 32 bit,che rappresentano l'indirizzo del registro che si desidera leggere.

  Una volta che il master SPI termina la trasmissione di questi dati, lo slave SPI invia 32 bit sulla linea $MISO$, ovvero il contenuto del registro selezionato. Alla fine della trasmissione di questi bit, il master deseleziona la periferica.

Una volta stabilito il significato dei bit che il master invia allo slave, il passo successivo consiste nel progettare l'architettura del componente, il cui compito è quello di suddividere i vari bit in ingresso in appositi registri ed inviarli alla memoria una volta conclusa la trasmissione del master SPI.

### 2.3.2 Datapath del componente "Bridge SPI to Avalon"

Per la realizzazione di un sistema più o meno complesso è necessario definire due blocchi fondamentali:

- Il datapath, che rappresenta l'insieme di unità di calcolo, come ad esempio unità di elaborazione (ALU) e registri, e definisce ciò che è in grado di fare una macchina;
- L'unità di controllo, la quale ha invece il compito di coordinare tutte le azioni necessarie per l'esecuzione delle istruzioni. [6]

Questi due blocchi sono strettamente legati in quanto l'uno lavora con i dati che riceve dall'altro. Solitamente, l'unità di controllo riceve dall'esterno il segnale di *start* ed altri segnali di controllo mentre il datapath riceve i dati da elaborare. In funzione di questi dati e dei segnali ricevuti dalla CU, invia a sua volta dei feedback all'unità di controllo, la quale modifica il suo comportamento in funzione di quanto ricevuto.

Anche nella realizzazione del "Bridge SPI to Avalon" sono stati progettati questi due blocchi. Il primo che andremo ad analizzare è il datapath, la cui struttura è mostrata in figura 2.7.

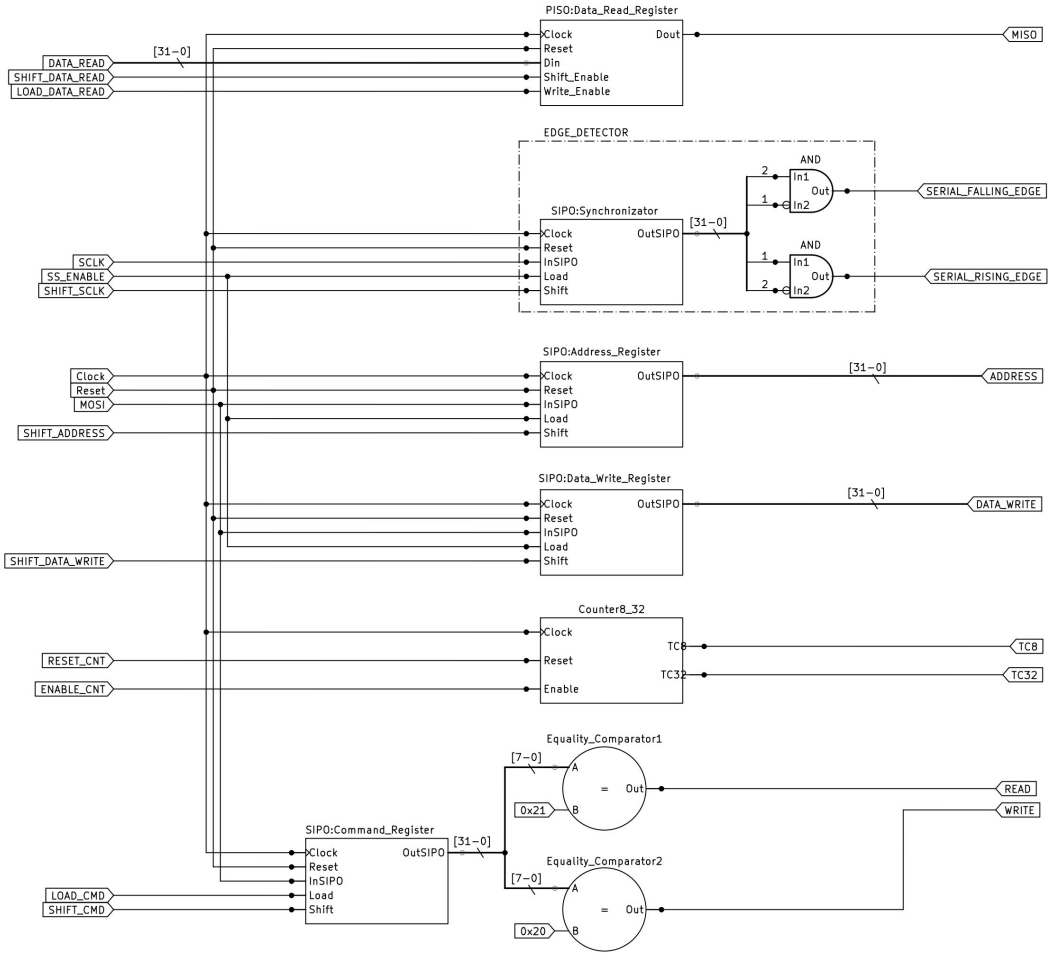


**Figura 2.7:** Datapath del "Bridge SPI To Avalon"

I blocchi funzionali che compongono il datapath sono:

- Registro SIPO (Serial-In-Parallel-Out) = Registro di 32 bit in cui il dato da memorizzare ($InSIPO$) entra in modo seriale, cioè un bit per ogni temporizzazione (*clock*), mentre l'uscita ($OutSIPO$) è in parallelo, cioè vengono mandati sul bus di uscita tutti e 32 bit insieme. Questo registro dispone del segnale di *reset*, necessario ad azzerare il contenuto presente al suo interno, un segnale di *Load*, il cui compito è quello di abilitare la scrittura del bit in ingresso, ed il segnale di $Shift$, ovvero il segnale che, se abilitato, fa traslare di una posizione i bit presenti al suo interno.

- Registro PISO (Parallel-In-Serial-Out) = Registro di 32 bit in cui il dato da memorizzare ($Din$) entra in modo parallelo, cioè 32 bit per ogni temporizzazione ($clock$), mentre l'uscita($Dout$) è seriale, cioè viene mandato in uscita un bit per volta. Questo registro dispone del segnale di $reset$, necessario ad azzerare il contenuto presente al suo interno, un segnale di $WriteEnable$, il cui compito è quello di abilitare la scrittura dei bit in ingresso, ed il segnale di $ShiftEnable$, ovvero il segnale che, se abilitato, fa traslare di una posizione i bit presenti al suo interno.

- Contatore = Contatore a 6 bit, in grado di contare da 0 a 63. Quando il segnale di $Enable$ è attivo, ogni colpo di $clock$ fa incrementare di 1 il contatore. Il segnale di $reset$ serve ad azzerare il contatore quando necessario. Le uscite, chiamate rispettivamente $TC8$ ($TerminalCount8$) e $TC32$ ($TerminalCount32$), rappresentano due flag che vengono attivate quando il contatore raggiunge un determinato valore.

- Comparatore di Uguaglianza = Un blocchetto che riceve come ingressi due valori e verifica se questi sono identici. Se la comparazione ha esito positivo, allora l'uscita sarà alta, altrimenti sarà bassa.

Una volta chiarita la funzione dei vari blocchi implementati, è necessario chiarire il significato dei segnali di ingresso e di uscita dell'architettura. Nelle tabelle 2.3 e 2.4 questi segnali vengono analizzati nel dettaglio.

| **Segnale** | **Bit** | **Provenienza** | **Descrizione** |
|---|---|---|---|
| $clock$ | 1 | Esterno | Segnale di temporizzazione del sistema |
| $reset$ | 1 | Esterno | Segnale di ripristino del sistema |
| $mosi$ | 1 | Esterno | $MasterOutSlaveIn$, segnale che trasporta, in modo seriale, i dati inviati dal $\mu C$ verso l'FPGA |
| $ss\_enable$ | 1 | Esterno | Segnale di abilitazione del blocco |
| $sck$ | 1 | Esterno | Segnale di temporizzazione dell'interfaccia SPI |
| $data\_read$ | 32 | Esterno | Segnale di dato indicante il valore letto in memoria |
| $reset\_cnt$ | 1 | CU | Segnale di ripristino del contatore |
| $enable\_cnt$ | 1 | CU | Segnale di abilitazione del contatore |
| $shift\_data\_read$ | 1 | CU | Segnale di abilitazione alla traslazione del dato salvato nel $Data\_Read\_Register$ |

| | | | |
|---|---|---|---|
| *shift_data_ _write* | 1 | CU | Segnale di abilitazione alla traslazione del dato salvato nel *Data_Write_Register* |
| *shift_address* | 1 | CU | Segnale di abilitazione alla traslazione del dato salvato nel *Address_Register* |
| *shift_cmd* | 1 | CU | Segnale di abilitazione alla traslazione del dato salvato nel *Command_Register* |
| *shift_sck* | 1 | CU | Segnale di abilitazione alla traslazione del dato salvato nel registro *Synchronizator* |
| *load_cmd* | 1 | CU | Segnale di abilitazione del *Command_Register* |
| *load_data_read* | 1 | CU | Segnale di abilitazione del *Read_Data_Register* |

**Tabella 2.3:** Significato dei segnali di input presenti nel datapath del Bridge SPI to Avalon

| **Segnale** | **Bit** | **Direzione** | **Descrizione** |
|---|---|---|---|
| *serial_falling_ _edge* | 1 | CU | Flag che segnala la presenza di un fronte di discesa del *sck* |
| *serial_rising_ _edge* | 1 | CU | Flag che segnala la presenza di un fronte di salita del *sck* |
| *miso* | 1 | Esterno | *MasterInSlaveOut*, segnale che trasporta, in modo seriale, i dati inviati dall'FPGA verso il $\mu C$ |
| *TC*8 | 1 | CU | Flag indicante il raggiungimento del valore 8 del contatore |
| *TC*32 | 1 | CU | Flag indicante il raggiungimento del valore 32 del contatore |
| *read* | 1 | CU | Segnale di abilitazione alla lettura in memoria |
| *write* | 1 | CU | Segnale di abilitazione alla scrittura in memoria |
| *address* | 32 | Esterno | Segnale indicante la locazione di memoria in cui effettuare un'operazione di lettura o scrittura |
| *data_write* | 32 | Esterno | Segnale di dato indicante il valore da scrivere in memoria |

**Tabella 2.4:** Significato dei segnali di output presenti nel datapath

A questo punto è possibile entrare più nel dettaglio dell'architettura e del suo funzionamento.

Come già spiegato nel paragrafo 2.1, quando il master invia una richiesta ad uno slave, deve per prima cosa selezionarlo e per fare ciò utilizza il segnale *ss_enable*. In questo modo, lo slave diventa sensibile ai segnali che riceve in ingresso. Il primo segnale che viene elaborato è *sck*, ossia il segnale di temporizzazione dell'interfaccia dell'SPI. Questo segnale ha una frequenza di funzionamento inferiore rispetto alla frequenza di *clk* a cui lavora il dispositivo. La prima task di questa architettura consiste nel sincronizzare l'unita di controllo a questo "clock alternativo". L'unità di sincronizzazione, chiamata nella figura 2.7 $EDGE\_DETECTOR$, è composta da un registro SIPO e da due porte AND. Il comportamento di questo blocco è semplice ed intuitivo. I segnali utilizzati sono:

- *Clock*, che si collega direttamente alla porta "clock" del SIPO. Questa rappresenta la frequenza di campionamento del serial clock;
- *Reset*, che si collega direttamente alla porta *reset* del SIPO;
- $SCLK$, che si collega alla porta $InSIPO$ del registro. Infatti ciò che viene salvato in questo registro sono i campioni del segnale di $SCK$, campionati alla frequenza di *clock*;
- $SS\_Enable$, collegato alla porta *Load* del SIPO. In questo modo il registro viene abilitato nello stesso momento in cui il master seleziona lo slave.
- $Shift\_sck$, collegata alla porta $Shift$ del SIPO. Questo segnale è sempre attivo dal momento in cui il master seleziona lo slave.

Quindi il segnale di *sck* viene campionato alla frequenza del segnale di *clock* ed i campioni vengono salvati in modo seriale all'interno del registro $Synchronizer$.

L'uscita è composta da 32 bit, ma gli unici che verranno utilizzati sono i bit presenti nella posizione 1 e 2. Questi due bit entreranno in due porte AND, le quali hanno un ingresso negato e uno no. Questo perché, ciò che è necessario cercare è il momento in cui il segnale passa da uno stato al suo opposto. Nel momento in cui il segnale passa da un valore logico alto ad un valore logico basso, si ha un fronte di discesa. Al contrario, se passa da un valore basso ad uno alto, si avrà un fronte di salita. Le espressioni booleane che sintetizzano ciò che è stato spiegato sono:

- $Fronte\_di\_salita <= (NOT\ \ Out\_SIPO(2)) \ \ AND \ \ Out\_SIPO(1)$;
- $Fronte\_di\_discesa <= (NOT\ \ Out\_SIPO(1)) \ \ AND \ \ Out\_SIPO(2)$;

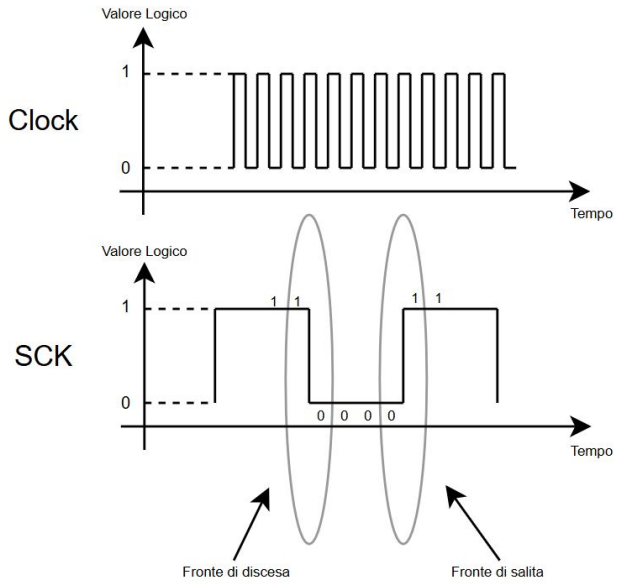


**Figura 2.8:** Fronte di discesa e di salita di *sck* rispetto al segnale di *clock*

In questo modo, in uscita dal blocco $EDGE\_DETECTOR$ avremo dei flag che indicano, in modo periodico tanto quanto il segnale *sck*, la presenza dei fronti del *sck*. Questi flag verranno indirizzati alla CU.

Una volta che l'architettura risulterà sincronizzata al *clock* dell'SPI, può cominciare il campionamento dei dati da elaborare. Come già accennato nel paragrafo 2.3.1, per effettuare una scrittura abbiamo di alcune informazioni, cioè il CMD, l'ADDRESS e il DATA. Tutte queste informazioni verranno inviate dal master sul segnale $MOSI$, il quale sarà collegato alla porta $InSIPO$ dei registri $Command\_Register$, $Address\_Register$ e $Data\_Write\_Register$.

Il primo dato fornito è il comando. Questo vuol dire che il segnale di abilitazione del $Command\_Register$ sarà attivo($Load\_Command$), così come il segnale $Shift\_Command$ per traslare di volta in volta i bit già memorizzati e scrivere sempre il nuovo dato nella posizione 0. Lo shift degli altri due registri sarà invece disabilitato. In questo modo, i bit che verranno inviati, verranno memorizzati solo nel $Command\_Register$. Insieme al registro in questione, verrà abilitato il $Counter8\_32$, il quale verrà aggiornato ogni volta che un bit verrà salvato nel registro. Memorizzati 8 bit, cioè la lunghezza del comando, il flag $TC8$ del contatore si attiverà e si passerà alla scrittura degli altri registri.

Affinché il comando abbia un significato per l'unità di controllo, dovrà prima essere decodificato. La decodifica verrà effettuata da due $Equality\_Comparator$, i quali riceveranno in ingresso gli 8 bit memorizzati nel registro e il codice associato all'operazione.

In particolare, il codice associato all'operazione di lettura sarà 0x21 (esadecimale)/00100001(binario), mentre il codice associato alla scrittura sarà 0x20(esadecimale) /00100000(binario). Se i bit in uscita del $Command\_Register$ corrisponderanno ad uno di questi valori, allora il segnale di $Read$ o di $Write$ si attiverà ed andrà alla CU,

la quale capirà l'operazione richiesta e seguirà rami diversi della macchina a stati.

Una volta completato l'invio dei bit relativi al comando, la CU resetterà il contatore per iniziare un nuovo conteggio per i bit dell'*address*. Infatti, il $MOSI$ continuerà a inviare bit come prima, con la differenza che i segnali $Load\_Command$ e $Shift\_Command$ saranno disattivati e verrà attivato il segnale di $Shift\_Address$. Il procedimento resta identico a quello spiegato per il comando, con l'unica eccezione che il flag $TC8$ del contatore verrà ignorato e verrà invece recepito il secondo flag, ovvero $TC32$. Infatti, mentre il comando è composto da 8 bit, l'indirizzo è composto da 32 bit.

Una volta conclusa l'acquisizione dell'indirizzo, la task successiva sarà l'acquisizione del dato da scrivere in memoria. Quindi verrà resettato nuovamente il contatore, verrà disattivato $Shift\_Address$ e verrà attivato $Shift\_Data\_Write$. Anche in questo caso, il dato da acquisire sarà di 32 bit, quindi la CU ignorerà $TC8$ e sarà sensibile a $TC32$.

Completata anche l'acquisizione del dato da scrivere, verranno posti in uscita i segnali di *write*, (proveniente direttamente dalla CU), *address* e $Data\_Write$ per effettuare l'operazione di scrittura in memoria.

Nel caso in cui l'operazione da svolgere fosse una lettura e non una scrittura, l'algoritmo cambia leggermente. Infatti l'acquisizione del comando e dell'indirizzo resteranno le stesse, ma in questo caso non sarà necessario acquisire $Data\_Write$. Al contrario, si effettuerà subito la lettura in memoria. Il dato che verrà letto in memoria ($Data\_Read$) dovrà però essere salvato nel registro $Data\_Read\_Register$. A differenza dei registri analizzati fin ora, questo è un registro PISO, cioè con ingresso in parallelo ed uscita in serie. Infatti, dalla lettura si acquisirà un dato di 32 bit e lo si salverà nel registro abilitando il segnale di $Load\_Data\_Read$. Una volta salvato il $Read\_Data$ nel registro e aver resettato il contatore, può iniziare la trasmissione del dato letto attivando lo $Shift\_Data\_Read$. Questa volta il dato non andrà dal master allo slave, ma dallo slave al master, in quanto l'operazione di lettura è una risposta ad un comando. Per questo motivo il dato letto verrà trasmesso sulla linea $MISO$.

## 2.4 Unità di controllo del componente "Bridge SPI to Avalon"

Una volta chiarito il funzionamento dell'architettura, è possibile analizzare l'unità di controllo del sistema. L'Unità di Controllo o Control Unit(CU) può essere considerata una sorta di "direttore d'orchestra" del sistema, in quanto detta i tempi per tutti i segnali.

Il metodo utilizzato per l'implementazione dell'unità di controllo è quello della FSM (Finite-State-Machine). La loro realizzazione fisica consiste in una rete per il calcolo delle uscite e dello stato futuro a partire dagli ingressi e dallo stato presente in cui si trova la macchina. L'FSM viene rappresentata come un circuito sequenziale

composto da un insieme di stati raggiungibili. Per ogni stato si descrivono le transizioni da e per esso, gli ingressi e le uscite. E' possibile utilizzare una rappresentazione grafica per descrivere l'evoluzione degli stati, chiamata State Transition Diagram (STD), o in gergo "pallogramma". Il pallogramma è composto da:

- $Stati \rightarrow Nodi \rightarrow Circonferenze$
- $Transizioni \rightarrow Vertici \rightarrow Frecce$

  Nota la struttura di una macchina a stati finiti, è possibile analizzare quella progettata per il progetto del "Bridge SPI to Avalon".

  Come si può notare nella figura 2.9, il pallogramma presenta 20 stati differenti. Vediamo nel dettaglio il significato di ognuno di essi e come avvengono i passaggi da uno stato al successivo.

  Supponendo che all'accensione del dispositivo venga asserito un segnale di reset ($CLR = 1$), il primo stato in cui entra la macchina è $s0\_reset$, che comporta l'azzeramento del contatore e dei registri presenti nell'architettura. Anche se non presenti in figura, ogni stato è collegato con $s0\_reset$ per evitare che la macchina vada in stallo in caso di malfunzionamento. In questo modo, attivando il segnale $CLR$, si ritorna nello stato di reset iniziale.

  Nel colpo di clock successivo alla disattivazione di $clr$, la CU si sposterà nello stato $s1\_idle$, ovvero uno stato di attesa in cui non avvengono modifiche dei segnali.

  Non appena verrà campionato il segnale $SS = 0$, la CU si sposterà in $s2\_clk$, ovvero lo stato in cui avviene la sincronizzazione con i fronti del segnale $sck$ già trattato nel paragrafo 2.3.2.

  Una volta agganciato il fronte di salita si entrerà nello stato $s3\_opcode\_samp$, in cui avviene il campionamento del primo bit del $CMD$ e l'incremento del contatore.

  Una volta campionato, si passerà allo stato $s4\_wait\_1$ da cui si dirameranno due differenti strade. Se il flag del contatore $TC8$ è attivo, lo stato successivo in cui andrà la macchina sarà $s5\_wait\_2$, poiché saranno stati campionati 8 bit di $CMD$. Se il flag non è attivo, la macchina attenderà nello stato attuale, fin quando non verrà rilevato un fronte di salita di $SCK$. A quel punto ritornerà nello stato $s3\_opcode\_sample$ per campionare i bit del $CMD$ mancanti.

  Una volta campionati tutti i bit relativi al $CMD$, dallo stato $s5\_wait_2$ si andrà allo stato $s6\_address\_samp$ a condizione che almeno uno tra $RD$ e $WR$ sia attivo e che venga campionato un $SRE = 1$.

  In $s6\_address\_samp$ verranno campionati i bit di indirizzo. In funzione dell'operazione eseguire, si potrà andare in due stati differenti. Se $RD = 0$ e $WR = 1$ la macchina passerà allo stato $s7_wait3\_wr$, mentre se $RD = 1$ e $WR = 0$ passerà allo stato $s13\_wait3\_rd$. Se entrambi i segnali sono uguali si tornerà allo stato $s1\_idle$, in quanto non sarà possibile stabilire quale operazione eseguire.

  Analizziamo prima il ramo relativo all'operazione di scrittura. Una volta in $s7\_wait3\_wr$, si controllerà il secondo flag del contatore, ovvero $TC32$. Se questo non sarà attivo, vorrà dire che non tutti e 32 i bit di indirizzo saranno stati campionati e quindi al primo $SRE = 1$ si ritornerà allo stato $s6\_address\_samp$. Se invece il

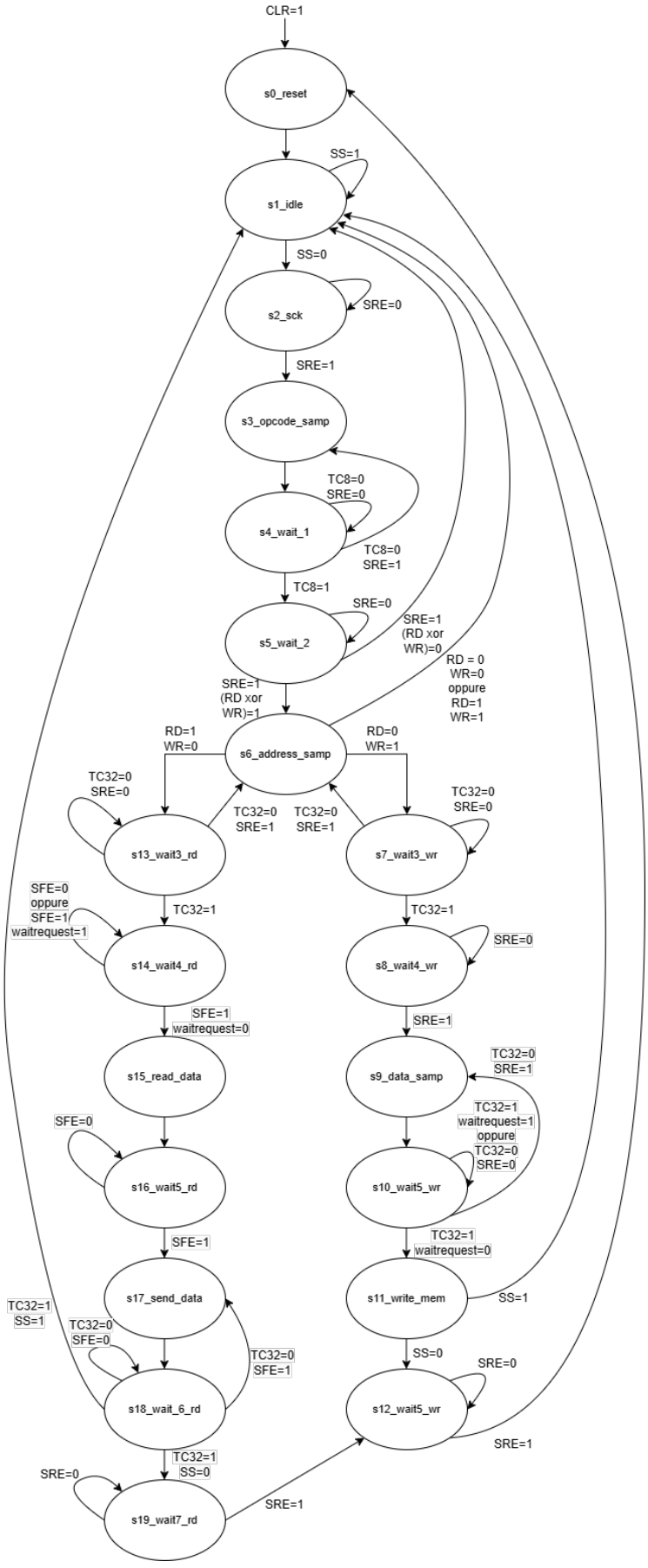


**Figura 2.9:** FSM del "Bridge SPI to Avalon"

flag sarà attivo, vorrà dire che tutti i bit di indirizzo saranno stati campionati e si potrà proseguire verso lo stato $s8\_wait4\_wr$.

Da li si passerà allo stato $s9\_data\_samp$, dove avverrà il campionamento dei bit del dato da scrivere. Dopo il campionamento dei bit di dato, si passerà allo stato $s10\_wait5\_wr$. Anche in questo caso si osserverà il flag del contatore $TC32$. Se non sarà attivo, al primo $SRE = 1$ si ritornerà allo stato $s9\_data\_samp$, se sarà attivo invece, verrà controllato un ulteriore segnale. Se $waitrequest = 1$, lo slave non sarà momentaneamente disponibile ad eseguire l'operazione richiesta. In tal caso si resterà in attesa nello stato $s10\_wait5\_wr$. Se invece $waitrequest = 0$, si passerà allo stato $s11\_write\_mem$, ovvero lo stato in cui avverrà la scrittura in memoria.

Una volta terminata l'operazione di scrittura, l'operazione sarà conclusa e lo stato successivo sarà $s1\_idle$ se $SS = 1$, altrimenti si andrà nello stato $s12\_wait5\_wr$ in attesa di un $SRE = 1$. Dopo di che si andrebbe nello stato $s0\_reset$ per pulire i registri ed il contatore per evitare problematiche varie nelle operazioni successive.

Per quanto riguarda invece l'operazione di lettura, una volta in $s13\_wait3\_rd$ verrà controllato $TC32$. Se questo non sarà attivo, vorrà dire che non tutti e 32 i bit di indirizzo saranno stati campionati e quindi al primo $SRE = 1$ si ritornerà allo stato $s6\_address\_samp$. Se invece il flag è attivo, vorrà dire che tutti i bit di indirizzo saranno stati campionati e si potrà proseguire verso lo stato $s14\_wait4\_rd$.

A partire da questo momento, la CU si sincronizzerà con il fronte di discesa del $SCK$ perché sul fronte di salita verranno campionati i bit e traslati nel registro interno, mentre sul nuovo fronte di discesa verrà posto il nuovo bit sulla linea MISO. Dallo stato $s14\_wait4\_rd$, se $SFE = 1$ e $waitrequest = 0$, si uscirà e si andrà nello stato $s15\_read\_data$ in cui avverrà la lettura del dato in memoria.

Dopo di ciò, si passerà allo stato $s16\_wait5\_rd$ e si attenderà il primo $SFE = 1$ per iniziare l'invio dei dati sulla linea $MISO$. L'invio dei bit avverrà nello stato $s17\_send\_data$. Nello stato $s18\_wait\_6\_rd$ avverrà invece il controllo del flag del contatore. Se $TC32 = 0$ al primo $SFE = 1$ si tornerà allo stato $s17\_send\_data$. Se $TC32 = 1$ si controllerà il segnale $SS$. Se uguale ad 1, lo stato successivo diventerà $s1\_idle$, altrimenti si passerà nello stato $s19\_wait7\_rd$, dal quale ci si potrà spostare al primo $SRE = 1$ in direzione $s12\_wait5\_wr$.

## 2.5 Architettura del sistema "Bridge SPI to Avalon"

Una volta definita l'architettura del "Bridge SPI to Avalon Component", è stato utilizzato il software $Platform\ Designer$ per collegare il componente progettato con una memoria RAM. Essendo già definita in una libreria del software, non è stato necessario progettarne e definirne la struttura. E' stato invece necessario definirne le specifiche, ovvero la grandezza, la modalità di funzionamento e l'interfaccia utilizzata.

Per la creazione del blocco "Bridge SPI to Avalon" è stato creato un nuovo componente personalizzato, al cui interno sono stati inseriti i codici sviluppati per il progetto. Anche in questo caso è stato necessario definire le interfacce di ingresso ed uscita affinché risultassero compatibili con gli standard utilizzati.

Una volta definiti nella loro interezza il componente e la memoria, sono stati effettuati i collegamenti tra di essi. A differenza di quanto fatto in precedenza, cioè collegare manualmente i vari segnali di input e output dei vari blocchi, i collegamenti sono stati fatti sfruttando il software. Infatti, essendo l'interfaccia di uscita del componente compatibile con l'interfaccia di ingresso della memoria, è stato sufficiente collegare il nodo corrispondente alle due interfacce. I collegamenti effettuati per tutto il sistema sono mostrati in figura 2.10

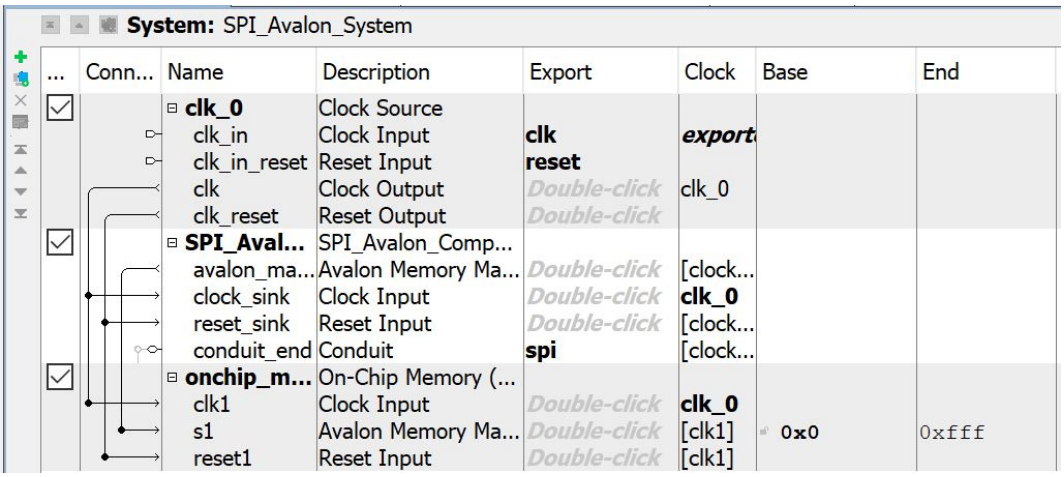


**Figura 2.10:** Collegamenti effettuati tramite il software Platform Designer

Questi collegamenti hanno portato alla creazione del sistema nella sua totalità. Da notare come l'interfaccia SPI sia stata definita mediante dei *conduit* in quanto input del sistema stesso.

I componenti che compongono il sistema sono mostrati in figura 2.11

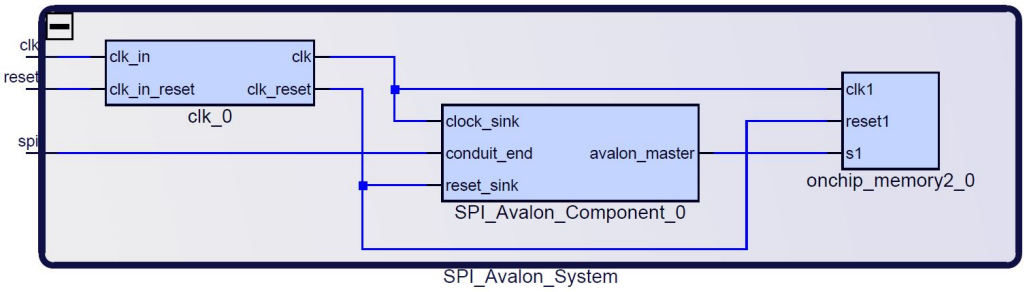


**Figura 2.11:** Sistema creato dal software Platform Designer

## 2.6 Timing del sistema "Bridge SPI to Avalon"

Un timing diagram o diagramma temporale è uno specifico tipo di diagramma di interazione, dove l'attenzione è focalizzata sui vincoli di tempo. Infatti sono usati per esplorare il comportamento di un oggetto in un dato periodo di tempo. Il tempo viene incrementato da sinistra a destra e il comportamento dei segnali è mostrato in

compartimenti separati disposti verticalmente. [7] Lo sviluppo di questi diagrammi è uno dei passi fondamentali nella progettazione elettronica.

A livello temporale viene progettato prima della definizione dell'unità di controllo, la quale verrà progettata proprio in funzione di questi diagrammi. In questo documento vengono analizzati in una sezione successiva alla descrizione della CU, solo per rendere più chiara l'esposizione e mostrare più chiaramente l'uguaglianza con le simulazioni sostenute in seguito.

Il primo timing diagram che verrà analizzato, mostra l'andamento dei segnali nel momento in cui viene resettato il sistema, viene selezionato lo slave e vengono campionati i primi due bit di CMD e l'ultimo. In particolare in questo diagramma viene inviato un CMD relativo all'operazione di scrittura.

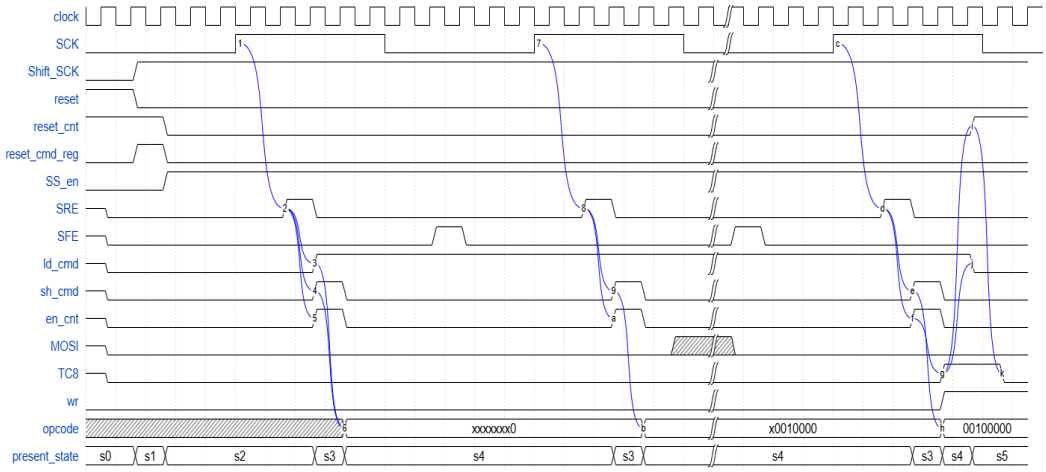


**Figura 2.12:** Timing diagram rappresentante il campionamento del segnale di comando per l'operazione di scrittura

L'andamento dei segnali e la dipendenza tra di essi viene efficacemente mostrata nella figura 2.12. In particolare si osservi come tutti i segnali dipendano dal fronte di salita del $SCK$. Una volta individuato il fronte, si attivano i segnali $ld\_cmd$, $sh\_cmd$ e $en\_cnt$. Questo avviene per ogni fronte individuato. Quando viene campionato l'ultimo bit di $CMD$, il contatore attiva $TC8$ che resetta il contatore ed abbassa il segnale di $ld\_cmd$.

Finito il campionamento del $CMD$, inizia il campionamento dell'$address$ e successivamente il campionamento del $data\_write$. L'andamento dei segnali è mostrato nei timing diagram presenti in figura 2.13 e figura 2.14.

L'algoritmo di campionamento dei due dati è praticamente identico. L'unica cosa che cambia è ovviamente il registro a cui sono riferiti i vari segnali di $enable$ e $shift$.

Una volta concluso il campionamento del $data\_write$, sono disponibili tutti i dati necessari per la scrittura in memoria, che, come abbiamo ampiamente spiegato, avviene seguendo l'interfaccia Avalon Memory-Mapped. Il timing diagram che mostra la scrittura è mostrato in figura 2.15

Conclusa l'operazione di scrittura, andiamo ad analizzare l'operazione di lettura. Per quanto riguarda il campionamento di $CMD$ e $Address$, il timing diagram è identico. L'unico cambiamento risulta essere nel valore dei bit campionati in quanto

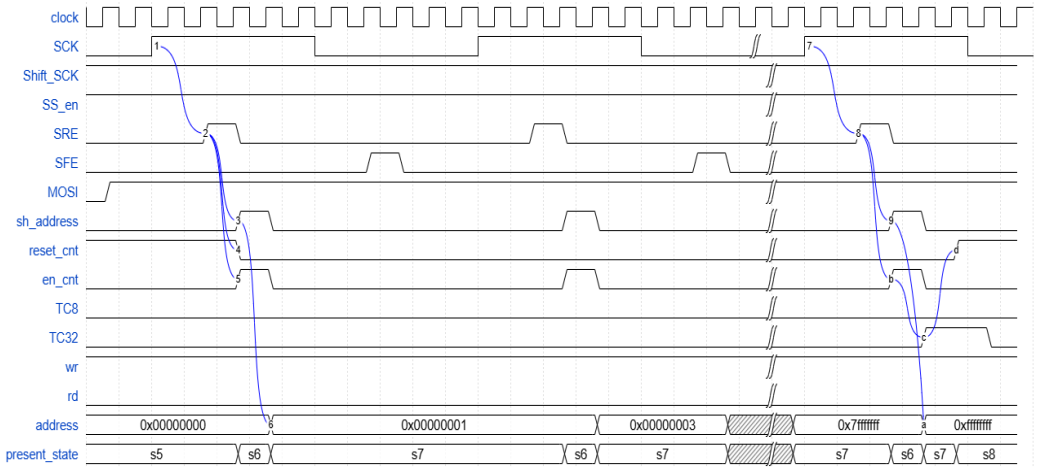


**Figura 2.13:** Timing diagram rappresentante il campionamento del segnale di indirizzo

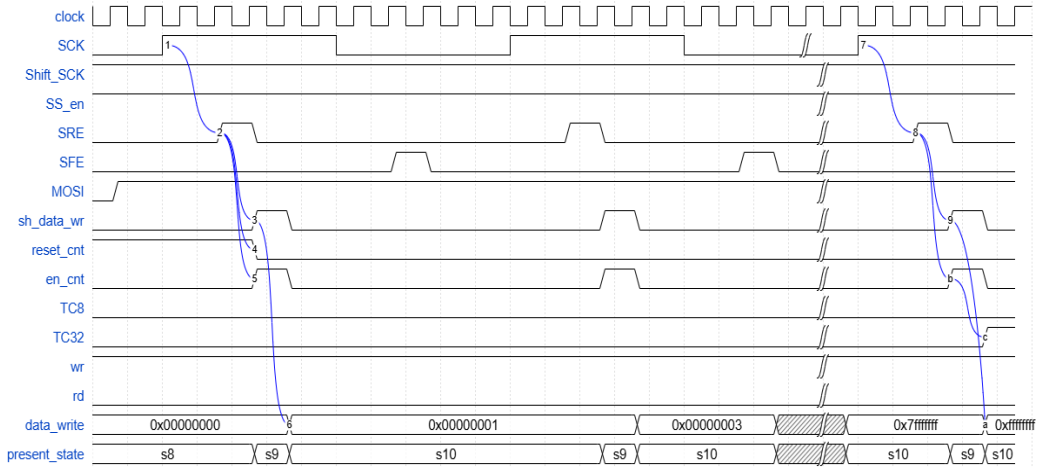


**Figura 2.14:** Timing diagram rappresentante il campionamento del segnale da scrivere in memoria

il comando di lettura è codificato in modo differente da quello di scrittura. Andiamo quindi ad osservare cosa accade dopo aver terminato il campionamento dell'ultimo bit di *address*.

La differenza rispetto al caso della scrittura consiste nel momento in cui si fa l'accesso in memoria. Mentre nella scrittura prima campionavamo il dato da scrivere e poi si accedeva in memoria per la scrittura, nella lettura si accede in memoria subito dopo aver terminato il campionamento dell'*address*. Una volta letto il dato presente in memoria, lo si salva nell'apposito registro e successivamente lo si manda in modo seriale sul segnale $MISO$.

## 2.7 Simulazione del comportamento del "Bridge SPI to Avalon"

Una volta concluso il progetto in tutte le sue parti, è necessario procedere con la simulazione del sistema. Per farlo è stato utilizzato il software "Altera-Modelsim",

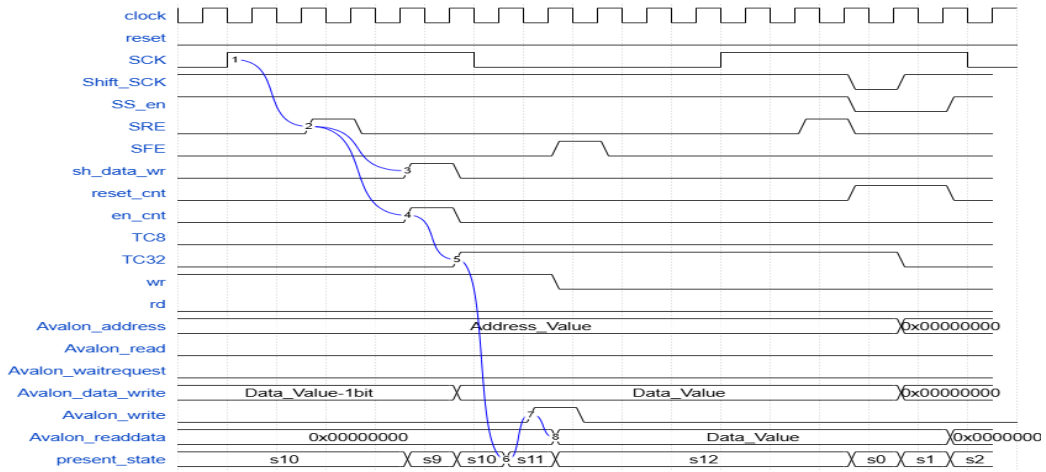


**Figura 2.15:** Timing diagram rappresentante la scrittura in memoria

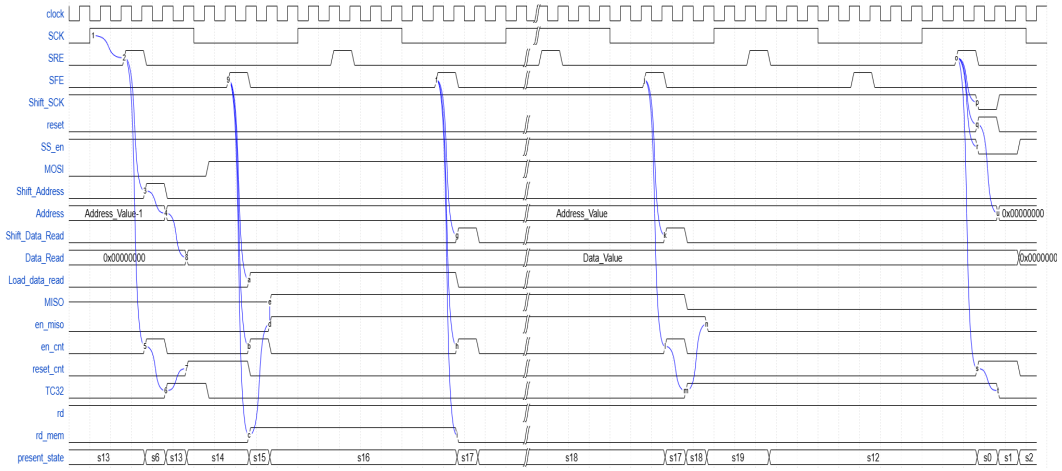


**Figura 2.16:** Timing diagram rappresentante la lettura del dato in memoria

il quale, tramite l'elaborazione di un "testbench" ha simulato il comportamento dell'architettura. Per chiarezza verranno mostrati inizialmente le simulazioni relative ai punti critici analizzati nel paragrafo 2.6, e successivamente la simulazione del sistema nel suo insieme.

La figura 2.17 mostra la sincronizzazione al $SCK$ ed il campionamento dei primi bit del CMD di scrittura.

La figura 2.18 invece mostra il campionamento dell'ultimo bit di $CMD$.

Una volta terminato il campionamento del $MOSI$ per l'acquisizione del comando, si passa al campionamento del $MOSI$ per l'acquisizione dell'indirizzo. I primi bit campionati sono mostrati nella figura 2.19.

La figura 2.20 invece mostra il campionamento dell'ultimo bit di $Address$.

Una volta terminato il campionamento del $MOSI$ per l'acquisizione dell'indirizzo, si passa al campionamento del $MOSI$ per l'acquisizione del dato da scrivere in memoria. I primi bit campionati sono mostrati nella figura 2.21.

La figura 2.22 invece mostra il campionamento dell'ultimo bit di $Data\_Write$.

Una volta terminato il campionamento dei dati, si effettua la scrittura in memoria

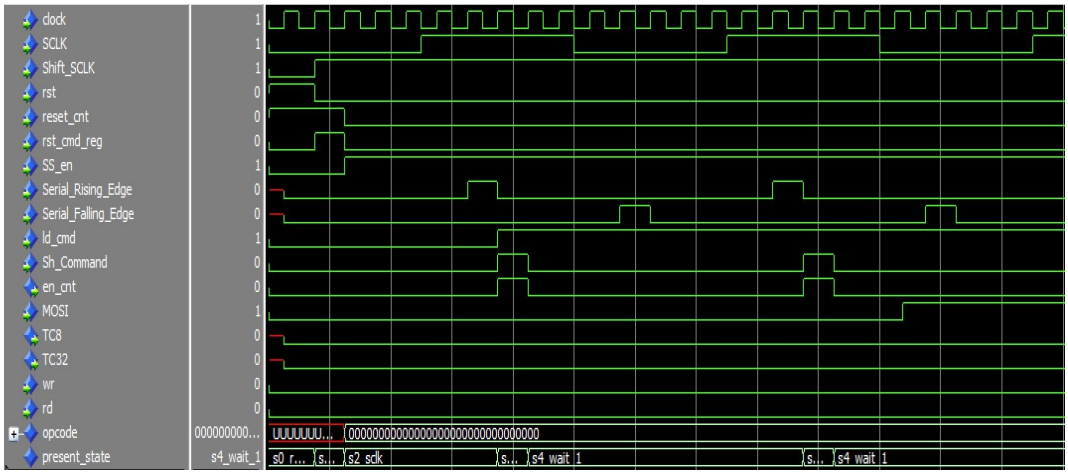


**Figura 2.17:** Simulazione del campionamento del primo bit del segnale di comando per l'operazione di scrittura

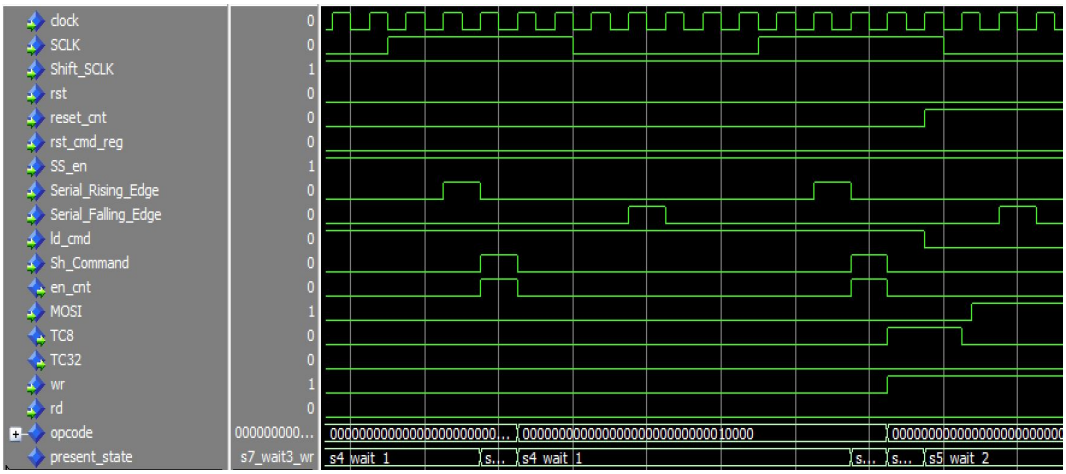


**Figura 2.18:** Simulazione del campionamento dell'ultimo bit del segnale di comando per l'operazione di scrittura

utilizzando proprio i dati appena acquisiti. La figura 2.23 mostra l'andamento dei segnali proprio in questo caso.

In questo modo è terminata l'operazione di scrittura in memoria. Come già sottolineato in occasione della spiegazione dei timing, per l'operazione di lettura possiamo trascurare il campionamento del comando e dell'indirizzo in quanto risulterebbero visivamente uguali a quelli presentati per la scrittura. Ovviamente ciò che cambierebbe sarebbero i valori campionati del *MOSI* in quanto il codice relativo all'operazione di lettura ha un codice diverso rispetto a quello di scrittura. Osserviamo la simulazione alla fine del campionamento dell'indirizzo in figura 2.24.

A differenza di quanto visto per la scrittura, non appena viene campionato l'ultimo bit di indirizzo, si accede alla memoria e si ottiene il dato presente all'indirizzo appena ricevuto. Dopo di ché questo dato viene trasmesso in modo seriale sul *MISO*. In figura 2.25 viene mostrato il comportamento dei segnali alla fine della trasmissione del dato letto.

A questo punto possiamo vedere come appaiono i segnali nel loro insieme.

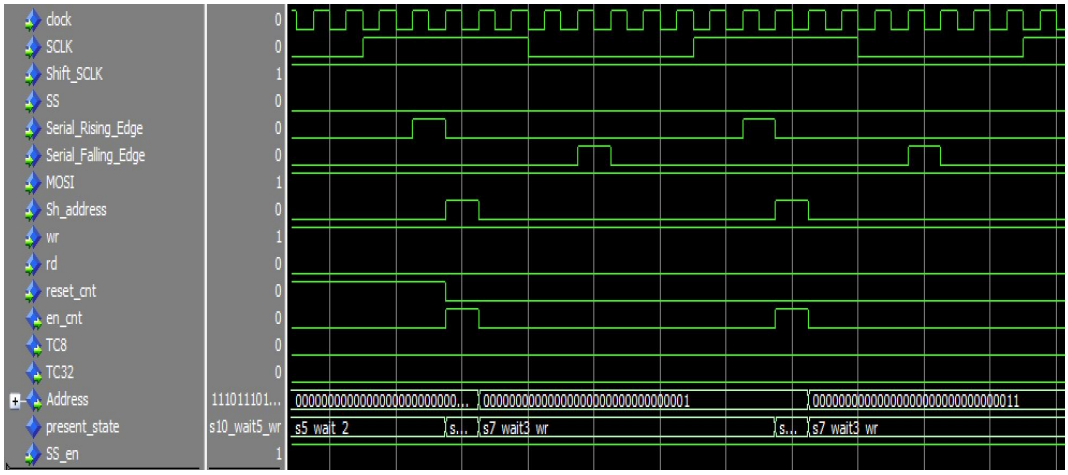


**Figura 2.19:** Simulazione del campionamento dei primi bit del segnale di indirizzo

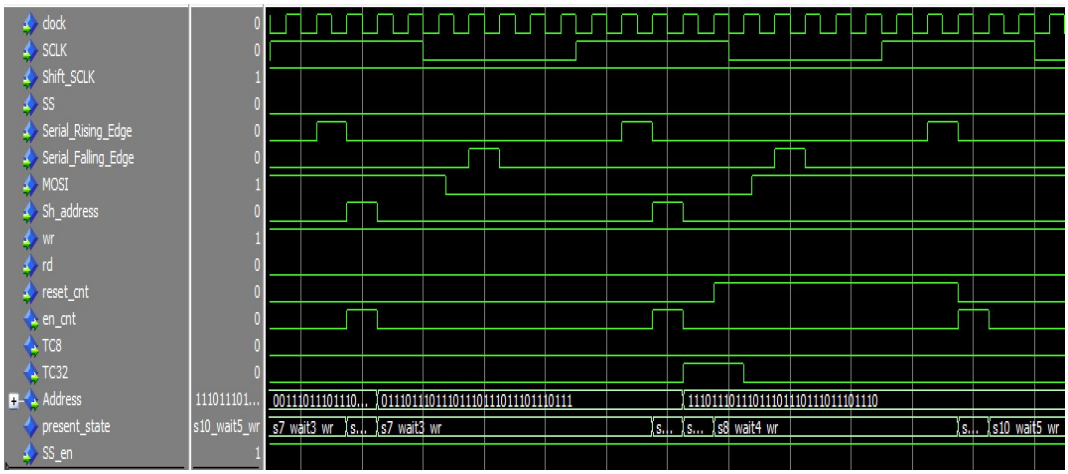


**Figura 2.20:** Simulazione del campionamento degli ultimi bit del segnale di indirizzo

In particolare si osservi come vegano campionati in sequenza il comando, l'indirizzo ed il dato. Una volta terminata la prima istruzione,se il comando è corretto, inizia una seconda istruzione, altrimenti si continua a campionare i segnali del $MOSI$ fin quando non si ottiene un comando corretto. Questo avviene perchè lo *Slave Select* resta attivo. In particolare, in figura 2.26 vengono fatte due scritture e due letture. Il dato letto in memoria verrà trasmesso sulla linea $MISO$ ogni volta che l'$en\_miso$ resta attivo.

## 2.8 Risultato ottenuto

Una volta simulato il sistema ed aver verificato che il comportamento sia lo stesso di quello progettato con la creazione dei timing diagram descritti nel paragrafo 2.6, si procede con il test sulla scheda VirtLAB. Per testare il sistema sono stati assegnati i vari pin con i segnali di ingresso ed uscita del sistema. Dopo di che, è stato caricato il file $.rbf$ sull'FPGA della parte USER. Per avere un terminale dal quale interfacciarsi al sistema, è stato programmato il µC con un file $.elf$, in modo tale da

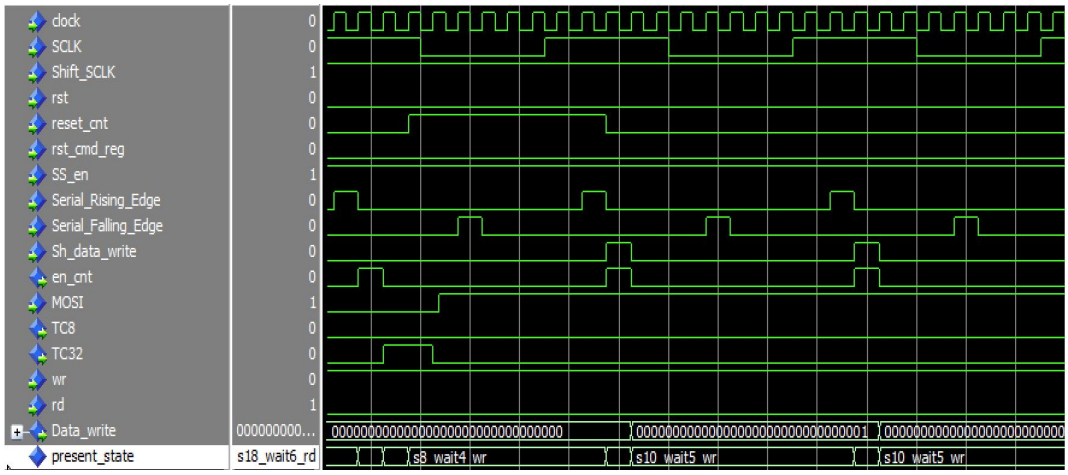


**Figura 2.21:** Simulazione del campionamento dei primi bit del segnale di dato

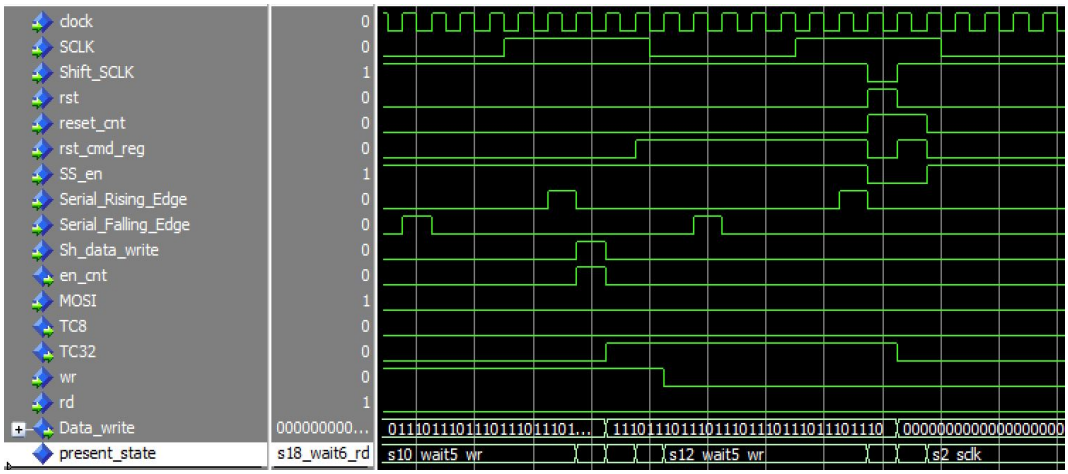


**Figura 2.22:** Simulazione del campionamento degli ultimi bit del segnale di dato

creare un'interfaccia testuale in cui inserire i comandi ed inviarli in modo corretto all'FPGA. Una volta collegata la VirtLAB in modo corretto ed aver programmato sia il µC che l'FPGA, apparirà la schermata presente in figura 2.27.

La figura 2.27 mostra la finestra con cui interfacciarsi all'FPGA. Si osservi come inserendo un "?" vengono mostrate le modalità di utilizzo del test. Per effettuare un'operazione di scrittura è sufficiente inserire come primo carattere la lettera "w", in quanto a tale lettera è stato assegnato il valore corretto del codice associato alla scrittura. Allo stesso modo, per effettuare una lettura è sufficiente inserire come primo carattere la lettera "r". Dopo aver stabilito l'operazione da effettuare, vengono inseriti i caratteri associati all'indirizzo. Essendo l'indirizzo di 32 bit, vengono utilizzati 8 caratteri esadecimali. Nel caso in cui si stia effettuando una lettura, non è necessario inserire nessun altro carattere, ed una volta inserito l'ultimo carattere dell'indirizzo, comparirà a schermo la lettura del dato in memoria. Al contrario, se si sta effettuando un'operazione di scrittura, una volta terminati i caratteri relativi all'indirizzo, dovranno essere inseriti i caratteri relativi al dato da scrivere in memoria. Anche in questo caso i bit da scrivere sono 32 e pertanto verranno utilizzati 8 caratteri.

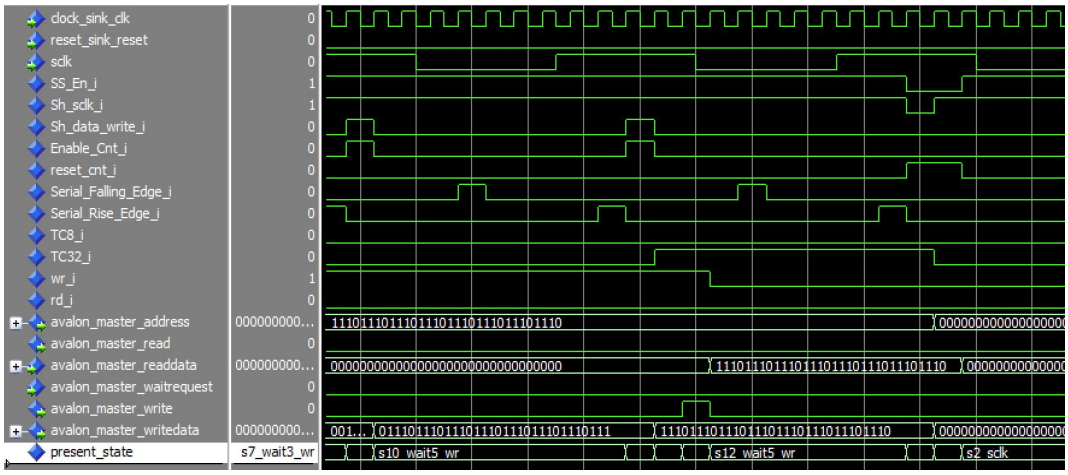


**Figura 2.23:** Simulazione della scrittura in memoria

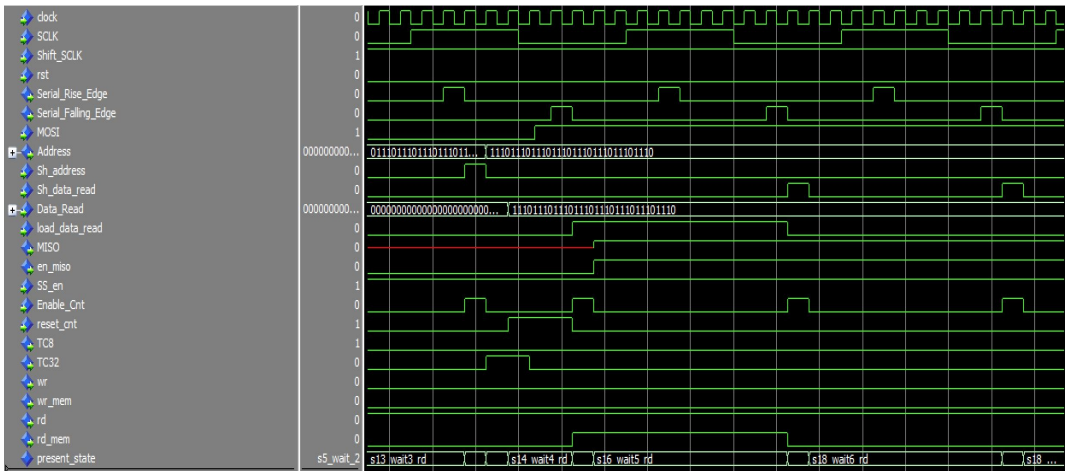


**Figura 2.24:** Simulazione della lettura in memoria ed inizio trasmissione nel $MISO$

In figura 2.28 viene mostrato il funzionamento del sistema.

Con la prima istuzione è stato scritto nel registro 0x00000001 il valore 0x1234abcd. con la seconda istruzione è stato scritto nell'indirizzo 0x00123a36 il dato 03a3cf17. Nella terza e nella quarta istruzione sono stati letti i precedenti valori. Nella quinta istruzione è stato sovrascritto il dato presente all'indirizzo 0x00000001 con il valore 11223344. Infine nell'ultima istruzione abbiamo letto l'indirizzo 0x00000001 per verificare che il dato fosse stato sovrascritto con quello nuovo.

Una volta terminata la fase di progettazione e simulazione, è stato analizzato il comportamento reale del sistema mediante l'uso di un oscilloscopio direttamente sulla scheda.

La figura 2.29 mostra un'operazione di scrittura. I segnali mostrati non sono altro che i 4 seggnali dell'SPI. In giallo osserviamo lo slave_select che si abbassa per tutta la durata dell'operazione; in azzurro il serial_clock, ovvero il segnale di temporizzazione, che oscilla per tutta la durata dell'istruzione; in rosa il MOSI, che trasmette prima il comando, poi l'indirizzo ed infine il dato da scrivere in memoria; in blu il MISO, il quale rimane in alta impedenza poichè non utilizzato in fase di

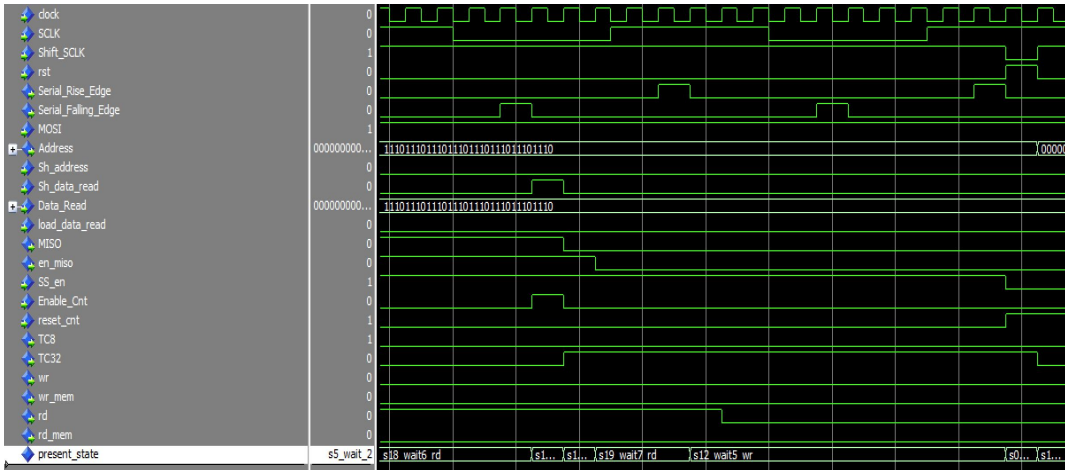


**Figura 2.25:** Simulazione di fine trasmissione del dato letto sul $MISO$

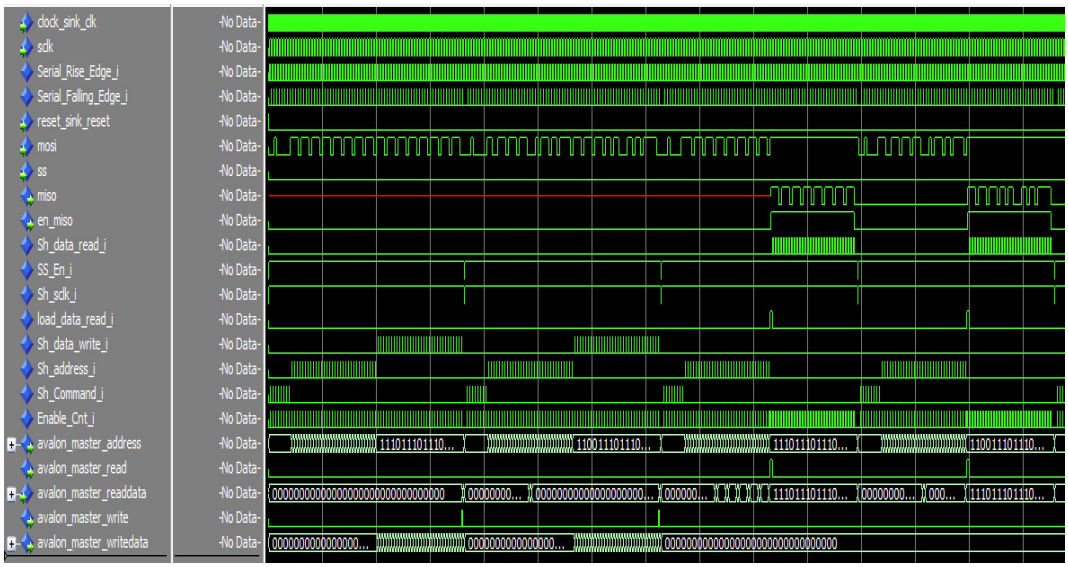


**Figura 2.26:** Simulazione di due scritture e due letture consecutive nel loro insieme

scrittura.

La figura 2.30 mostra invece un'operazione di lettura, in cui il MISO viene attivato, trasmettendo il dato appena letto al microcontrollore.

Questo dimostra come si sia effettivamente riusciti a mettere in contatto il µC della scheda con l'FPGA e come si sia riusciti a tradurre l'interfaccia SPI in interfaccia Avalon Memory-Mapped e viceversa. Questo progetto può essere considerato un lavoro preliminare per i progetti seguenti, in quanto rappresenta il fondamento su cui si basano.

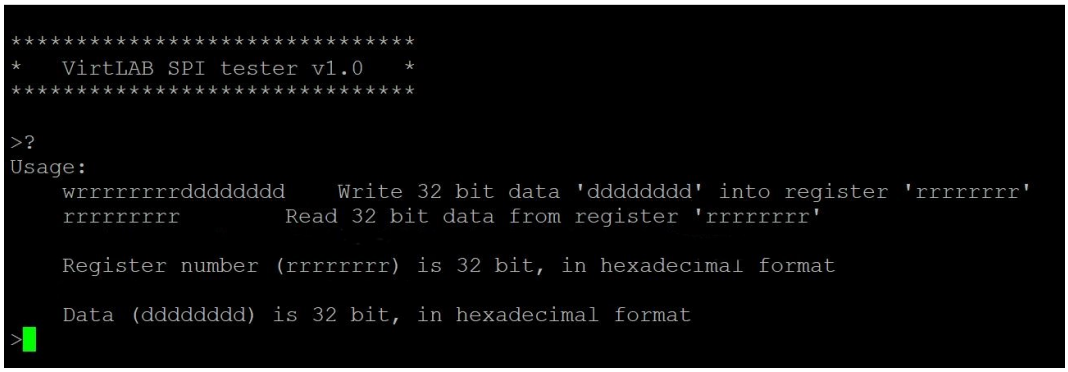


**Figura 2.27:** Interfaccia testuale del "Bridge SPI to Avalon"

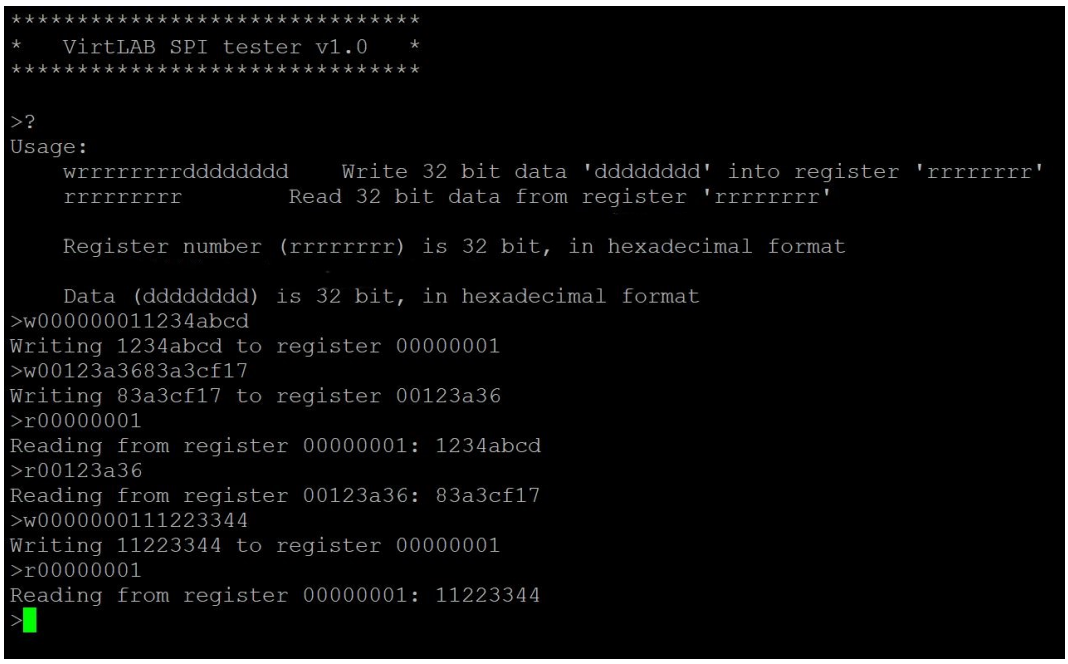


**Figura 2.28:** Test "Bridge SPI to Avalon"

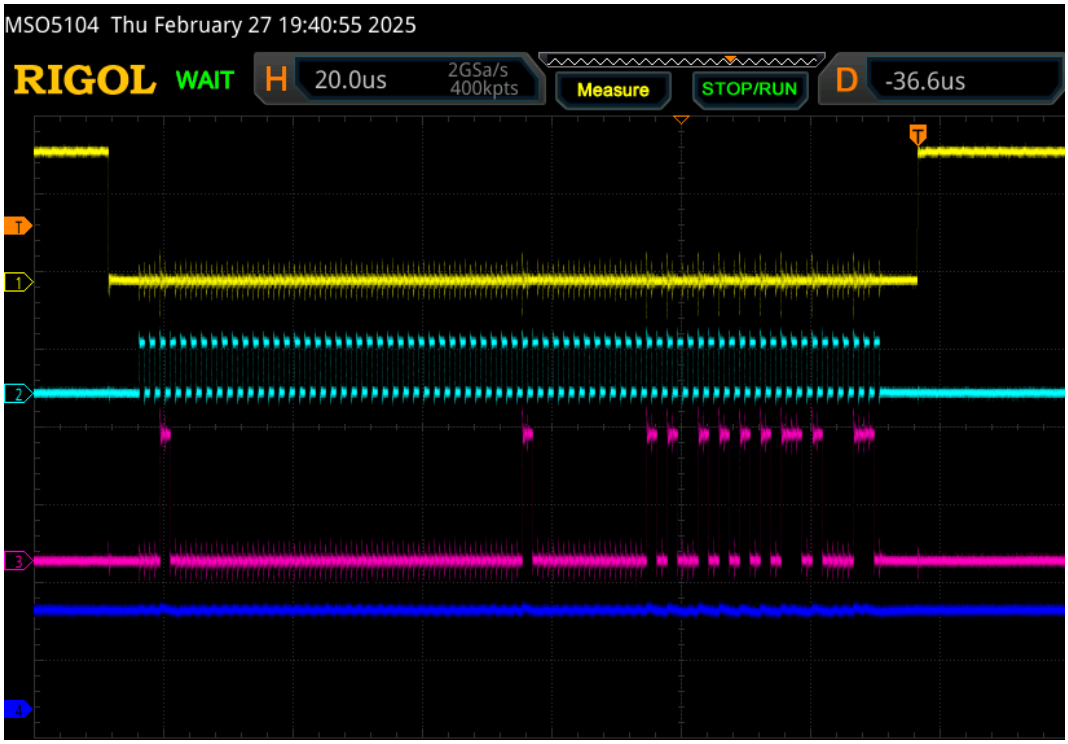


**Figura 2.29:** Operazione di scrittura osservata con un oscilloscopio a 4 canali

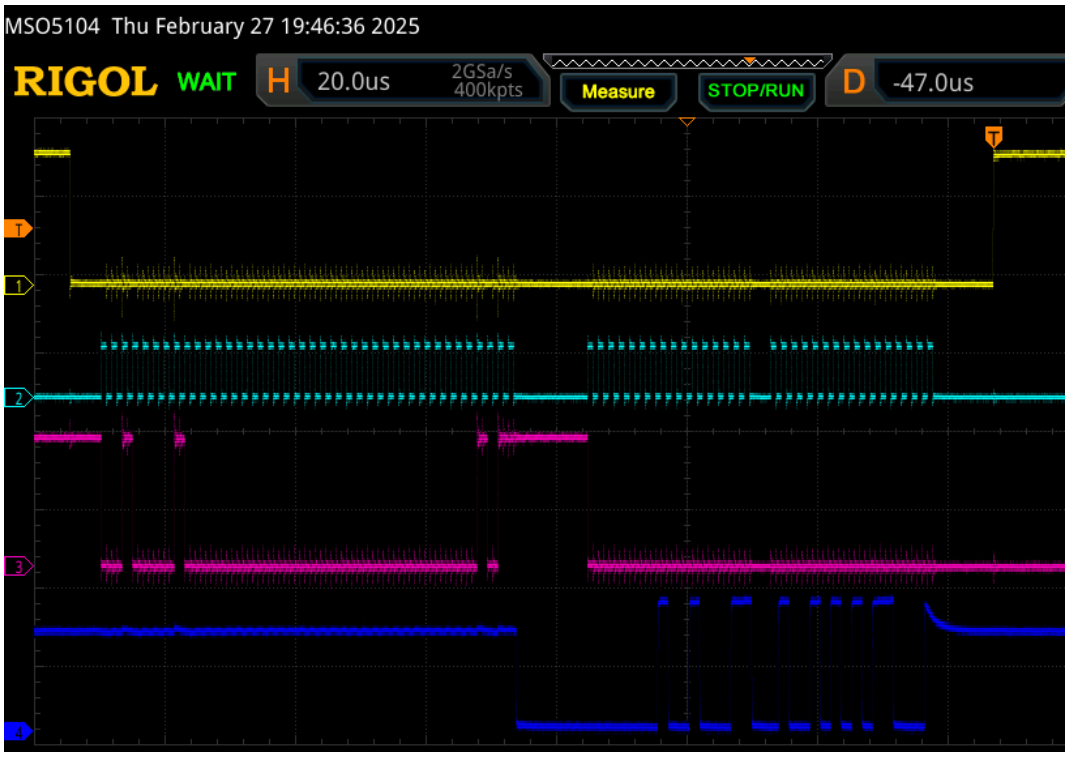


**Figura 2.30:** Operazione di lettura osservata con un oscilloscopio a 4 canali

# Capitolo 3

# Generatore di pattern digitali

## 3.1 Descrizione del progetto e risultati attesi

Nel mondo dell'elettronica digitale, avere la possibilità di generare pattern digitali in modo flessibile e preciso è fondamentale. Questo tipo di tecnologia trova applicazione in diversi ambiti, come per esempio nel test dei circuiti. In questa tesi, è stato sviluppato un generatore di pattern digitali che permette di generare sequenze programmabili dall'utente. Il cuore del generatore è un'architettura che permette di configurare dinamicamente alcune delle sue funzioni principali:

- Trigger: definisce le condizioni tali per cui è possibile avviare la generazione del pattern;
- Clock Generator: gestisce la temporizzazione della sequenza;
- Loop Manager: controlla il numero di ripetizioni dei pattern;
- Maschera di Uscita: consente di modificare e filtrare il segnale prodotto.

Per rendere il sistema facilmente gestibile, la configurazione dei registri di dato, stato e controllo avviene tramite il "Bridge SPI to Avalon" descritto nel capitolo precedente. Dopo aver definito la sequenza da generare ed aver verificato che rispetti i vari vincoli impostati dall'utente, questa viene salvata in una memoria, dalla quale potrà successivamente essere letta per la generazione del pattern sui pin di uscita.

Lo scopo di questa tesi è illustrare nel dettaglio l'architettura e il funzionamento del generatore di pattern digitali.

## 3.2 Architettura del "Pattern Generator"

Lo scopo del progetto è quello di creare un generatore di pattern, cioè una sorta di blocco che mandi sui pin di I/O un segnale preconfigurato. Per farlo viene sfruttato il "Bridge SPI to Avalon" per la configurazione dei registri, i cui dati verranno successivamente elaborati al fine di salvare la sequenza in memoria oppure eseguire la generazione. Per dare un'organizzazione visiva, la figura 3.1 mostra i macro blocchi di questo progetto e come sono collegati tra loro.

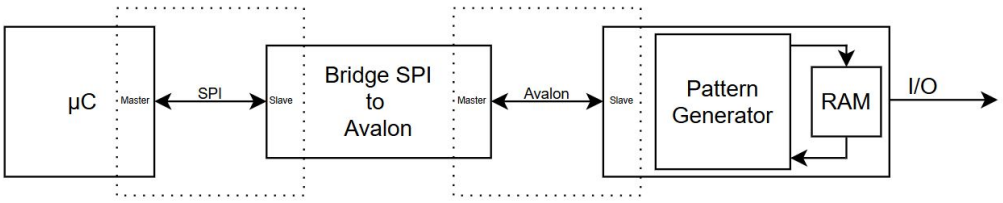


**Figura 3.1:** Blocchi che compongono il sistema "Pattern Generator"

### 3.2.1 Specifiche di progetto

Per realizzare il progetto del "Pattern Generator" è stato sfruttato il blocco "Bridge SPI to Avalon" progettato in precedenza. A differenza di quanto esposto nel capitolo precedente, il "Bridge SPI to Avalon" non scriverà i dati direttamente in memoria, ma nei registri presenti nell'architettura del "Pattern Generator". Questo concetto è molto importante in quanto permette all'utente di scrivere direttamente nei registri che andranno a modificare il comportamento del sistema. Questa infatti risultava essere una delle specifiche richieste, ovvero la flessibilità di utilizzo.

Le specifiche di questo progetto richiedevano lo sviluppo di alcune caratteristiche fondamentali per un generatore di pattern:

- La possibilità di modificare la frequenza di funzionamento del sistema mediante un divisore del clock base.
- La possibilità di selzionare un clock esterno, piuttosto che uno interno.
- La possibilità di inserire manualmente la sequenza da salvare in memoria e da generare in seguito.
- La possibilità di inserire manualmente l'indirizzo di memoria in cui salvare una determinata sequenza logica.
- La possibilità di inserire una maschera di trigger che indicasse i bit da analizzare e quelli da ignorare.
- La possibilità di inserire la condizione tale per cui il trigger si attivi e dia il via libera al salvataggio del pattern in memoria.
- La possibilità di selezionare la modalità di funzionamento del generatore tra sequenziale e one-shot.
- La possibilità di inserire un numero massimo di generazioni senza l'aggiornamento manuale dell'indirizzo.
- La possibilità di impostare una maschera di uscita, in modo da poter ignorare alcuni bit rispetto ad altri.
- La possibilità di generare un pattern in funzione delle specifiche impostate.

- La possibilità di leggere il contenuto dei registri di stato e controllo, oltre che della memoria.

Per ognuna di queste specifiche è stato progettato un blocco funzionale o un registro che verranno analizzati in seguito.

### 3.2.2 Datapath del componente "Pattern Generator"

L'architettura del sistema "Pattern Generator" è complessa in quanto oltre al blocco "Bridge SPI to Avalon" ed alla memoria, presenta tutta la serie di blocchi necessari per soddisfare le funzionalità introdotte precedentemente. Analizziamo nel dettaglio l'architettura del componente "Pattern Generator" in figura 3.2.

I blocchi funzionali che compongono il datapath sono:

- Registro PIPO (Parallel-In-Parallel-Out) = Registro di 32 bit che in cui il dato da memorizzare ($Data\_IN$) entra in modo parallelo, cosi come in uscita($Data\_OUT$). Questo vuol dire che in un colpo di clock verranno scritti 32 bit di dato ed in uscita verranno inviati 32 bit di dato insieme. Questo registro dispone del segnale di *reset*, necessario ad azzerare il contenuto presente al suo interno ed un segnale di *Enable* che abilita o disabilita il registro.

- Clock Generator = Questo blocco ha come ingresso un segnale di *clock*, un segnale di *reset* ed un valore di divisione ($DIVIDER$). In base a questo valore, genera un clock in uscita (*clock_out*) con una frequenza inferiore rispetto a quella del clock di ingresso.

- Address Manager = Questo blocco è un gestore di indirizzi il quale incrementa il valore dell'indirizzo (*address_in*) di 1 quando il segnale di abilitazione (*enable_adder*) è attivo. Il valore aggiornato viene poi salvato in *address_out*.

- Trigger Manager = Questo blocco implementa un sistema di rilevamento di condizioni basato su una maschera (*mask_in*). Confronta il risultato di un'operazione AND tra *data_in* e *mask_in* con un valore di riferimento (*value_trigger_condition*). Se la condizione è soddisfatta, attiva il segnale di uscita *trigger_out*.

- Loop Manager = Questo blocco gestisce un contatore che si ripete fino a un massimo ($Max\_Rep$). Quando il contatore raggiunge $Max\_Rep$, il segnale $End\_Loop$ viene attivato, segnalando la fine del ciclo.

- Multiplexer = I multiplexer sono utilizzati per selezionare uno tra due o più segnali in ingresso e indirizzarlo su un'unica uscita, in base al valore del segnale di selezione.

- Memoria RAM = Questo blocco è una memoria con 256 allocazioni da 32 bit. Anche se si hanno a disposizione 32 bit di indirizzo in ingresso al "Pattern Generator", vengono inviati solamente i bit address[7-0].

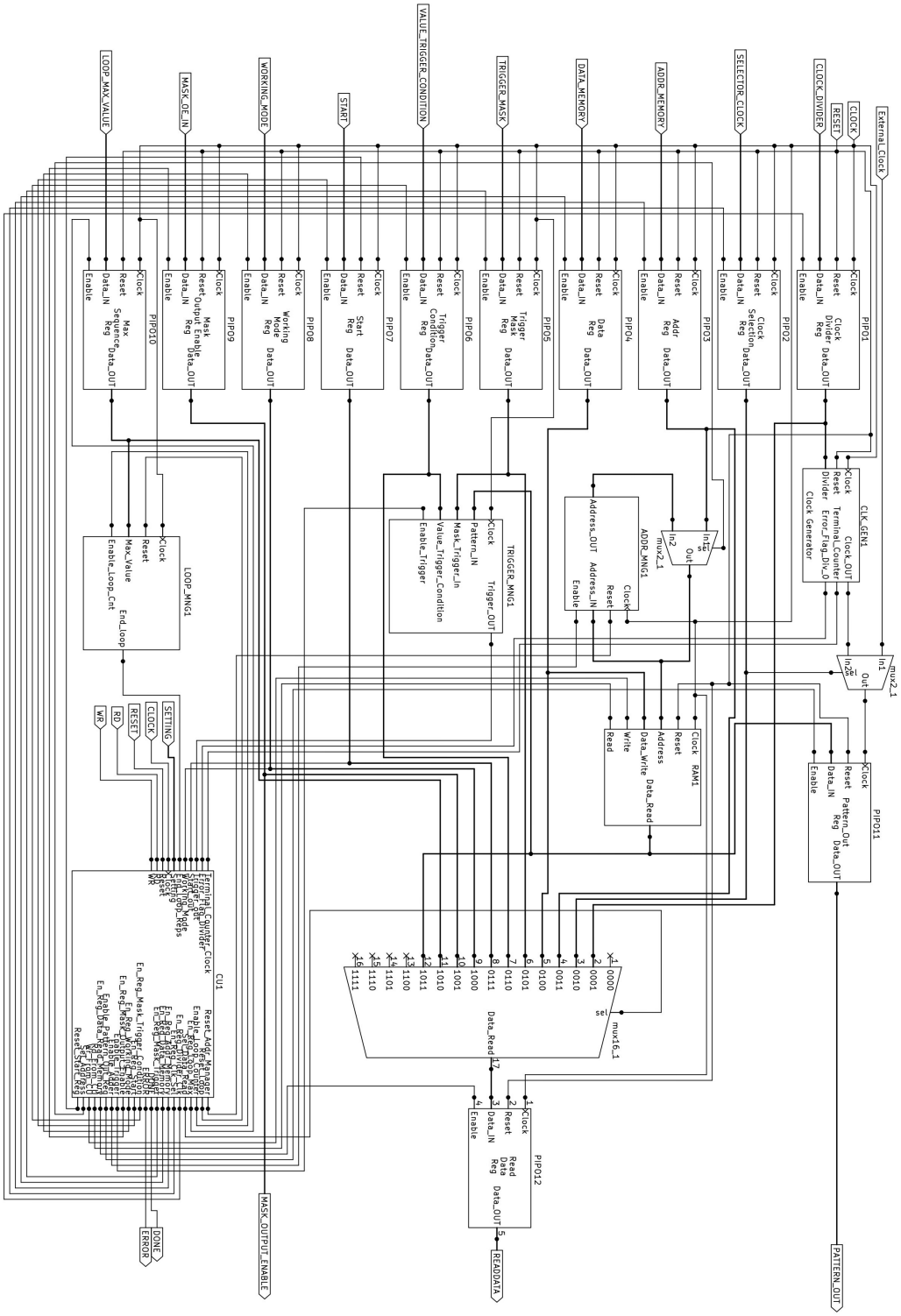


**Figura 3.2:** Datapath del "Pattern Generator" con unità di controllo

Una volta chiarito il compito dei vari blocchi implementati, è necessario avere chiaro il significato dei segnali in ingresso e in uscita all'architettura. Nelle tabelle 3.1 e 3.2 questi segnali vengono analizzati nel dettaglio.

| **Segnale** | **Bit** | **Provenienza** | **Descrizione** |
|---|---|---|---|
| *clock* | 1 | Esterno | Segnale di temporizzazione del sistema. |
| *reset* | 1 | Esterno | Segnale di ripristino del sistema. |
| *external_clock* | 1 | Esterno | Segnale di temporizzazione inviato dall'esterno ed utilizzabile al posto del *clock* interno. |
| *clock_divider* | 32 | Esterno | Segnale che determina il fattore di divisione del clock. |
| *selector_clock* | 32 | Esterno | Segnale che determina l'utilizzo del *clock* o dell'*external_clock*. |
| *addr_memory* | 32 | Esterno | Segnale indicante la locazione di memoria in cui effettuare un'operazione di lettura o scrittura. |
| *data_memory* | 32 | Esterno | Segnale di dato indicante il valore da scrivere in memoria. |
| *trigger_mask* | 32 | Esterno | Maschera usata per selezionare specifici bit di pattern_in. |
| *value_trigger _condition* | 32 | Esterno | Valore di riferimento per il confronto con pattern_in mascherato. |
| *start* | 32 | Esterno | Valore che indica l'inizio della generazione del pattern. |
| *working_mode* | 32 | Esterno | Valore che indica la modalità di funzionamento del generatore. Se l'ultimo bit è 0 allora il funzionamento è *One_Shot*, se 1 il funzionamento è *Sequenziale*. |
| *mask_oe_in* | 32 | Esterno | Maschera usata per selezionare specifici bit di uscita del pattern_out |
| *loop_max_value* | 32 | Esterno | Segnale indicante il numero di ripetizioni in modalita sequenziale, senza l'aggiornamento di *address* in modo manuale. |
| *reset_addr _manager* | 1 | CU | Segnale di ripristino di *addr_manager*. |
| *reset_loop* | 1 | CU | Segnale di ripristino di *loop_manager*. |

| | | | |
|---|---|---|---|
| *enable_loop _counter* | 1 | CU | Segnale di abilitazione del contatore presente nel *loop_manager*. |
| *enable_reg _loop_max* | 1 | CU | Segnale di abilitazione del registro *max_sequence_reg*. |
| *sel_data_read* | 4 | CU | Segnale di selezione del multiplexer, la cui uscita è collegata al registro *readdata_register* |
| *enable_reg _divider_clock* | 1 | CU | Segnale di abilitazione del registro *clock_divider_reg*. |
| *enable_reg _clock_sel* | 1 | CU | Segnale di abilitazione del registro *clock_selection_reg*. |
| *enable_reg _addr_memory* | 1 | CU | Segnale di abilitazione del registro *addr_register*. |
| *enable_reg _data_memory* | 1 | CU | Segnale di abilitazione del registro *data_register*. |
| *enable_reg _mask_trigger* | 1 | CU | Segnale di abilitazione del registro *trigger_mask_reg*. |
| *enable_reg_ _mask_trigger_ _condition* | 1 | CU | Segnale di abilitazione del registro *trigger_mask_condition_reg*. |
| *enable_reg_start* | 1 | CU | Segnale di abilitazione del registro *start_reg*. |
| *enable_reg _working_mode* | 1 | CU | Segnale di abilitazione del registro *working_mode_reg*. |
| *enable_reg_ _mask_output_ _enable* | 1 | CU | Segnale di abilitazione del registro *mask_OE_reg*. |
| *enable_trigger* | 1 | CU | Segnale di abilitazione di *trigger_manager*. |
| *enable_adder* | 1 | CU | Segnale di abilitazione del sommatore presente in *addr_manager*. |
| *enable_reg _pattern_out* | 1 | CU | Segnale di abilitazione del registro *pattern_out_reg*. |
| *enable_reg_data _read_memory* | 1 | CU | Segnale di abilitazione del registro *readdata_reg*. |
| *rd_from_CU* | 1 | CU | Segnale di abilitazione della lettura in memoria. |
| *wr_from_CU* | 1 | CU | Segnale di abilitazione della scrittura in memoria. |
| *sel_address* | 1 | CU | Segnale di selezione del multiplexer in uscita da *addr_register*. |

| | | | |
|---|---|---|---|
| *reset_start_reg* | 1 | CU | Segnale di ripristino del registro *start_register*. |

**Tabella 3.1:** Significato dei segnali di input presenti nel datapath del Pattern Generator

| **Segnale** | **Bit** | **Direzione** | **Descrizione** |
|---|---|---|---|
| *pattern_out* | 32 | Esterno | Segnale di uscita principale, indicante il pattern generato dal sistema |
| *readdata* | 32 | Esterno | Segnale indicante il dato letto in uno dei registri di dato o stato. |
| *mask_oe_out* | 32 | Esterno | Maschera usata per selezionare specifici bit di uscita del pattern$_o ut$ |
| *terminal_ _counter_clock* | 1 | CU | Segnale che indica il raggiungimento del conteggio impostato nel *clock_generator*. |
| *error_flag_ _divider* | 1 | CU | Segnale di errore che si attiva quando $DIVIDER = 0$. |
| *trigger_out* | 1 | CU | segnale di trigger che si attiva se la condizione nel *trigger_manager* è soddisfatta. |
| *start_out* | 1 | CU | Segnale che se attivo abilita la generazione del pattern |
| *working_mode* | 1 | CU | Segnale che indica la modalità di funzionamento del sistema |
| *end_loop_reps* | 32 | CU | Segnale che indica la fine del loop |

**Tabella 3.2:** Significato dei segnali di output presenti nel datapath del Pattern Generator

A questo punto è possibile entrare più nel dettaglio dell'architettura e del suo funzionamento. I primi blocchi architetturali che vengono utilizzati sono sicuramente i registri di dato, stato e controllo. Infatti, per poter utilizzare il generatore di pattern digitali, è necessario sceglierne le impostazioni. Analizziamo il significato di questi registri e l'indirizzo associato ad ognuno di questi:

- CLOCK_DIVIDER_REGISTER = In questo registro a 32 bit viene memorizzato il divisore della frequenza del clock. In funzione del valore memorizzato al suo interno, il periodo del clock utilizzato dal generatore di pattern verrà moltiplicato di questa quantità, facendo funzionare quindi il sistema con una velocità inferiore. L'indirizzo associato a questo registro è 0x00000001 in esadecimale.
- CLOCK_SELECTION_REGISTER = In questo registro a 32 bit viene memorizzato il selettore del clock. In funzione del valore memorizzato al suo

interno, il generatore di pattern utilizzerà un clock interno, oppure un clock inviato dall'esterno. Se l'ultimo bit è 0, allora si utilizzerà il clock interno, se è 1 utilizzerà il clock esterno. L'indirizzo associato a questo registro è 0x00000002 in esadecimale.

- ADDR_REGISTER = In questo registro a 32 bit viene memorizzato l'indirizzo di memoria su cui andare a scrivere o leggere. Essendo la memoria di sole 256 locazioni, sono necessari solo gli ultimi 8 bit meno significativi. L'utilizzo di un registro così grande è giustificato dal fatto di implementare un solo blocco "PIPO" per tutti i vari registri ed eventualmente facilitare la possibilità di estensione delle specifiche. L'indirizzo associato a questo registro è 0x00000003 in esadecimale.
- DATA_REGISTER = In questo registro a 32 bit viene memorizzato il dato da scrivere in memoria, e cioè il futuro pattern digitale da generare. Ogni volta che viene scritto un valore in questo registro, in automatico verrà salvato anche nella memoria RAM presente nell'architettura. L'indirizzo associato a questo registro è 0x00000004 in esadecimale.
- TRIGGER_MASK_REGISTER = In questo registro a 32 bit viene memorizzata la maschera del trigger. La funzione di questa maschera è quella di permettere all'utente di ignorare alcuni bit del pattern piuttosto che altri, al fine di verificare che una certa condizione sia soddisfatta. L'indirizzo associato a questo registro è 0x00000005 in esadecimale.
- TRIGGER_CONDITION_REGISTER = In questo registro a 32 bit viene memorizzata la condizione del trigger, ovvero la condizione che deve rispettare il pattern una volta mascherato. L'indirizzo associato a questo registro è 0x00000006 in esadecimale.
- START_REGISTER = In questo registro a 32 bit viene memorizzato il segnale di avvio della generazione. Se l'ultimo bit è un 1, allora avvia la generazione del pattern in uscita. L'indirizzo associato a questo registro è 0x00000007 in esadecimale.
- WORKING_MODE_REGISTER = In questo registro a 32 bit viene memorizzata la modalità di funzionamento del generatore. Se l'ultimo bit è 0, allora il generatore funzionerà in modalità one-shot, ovvero effettuerà una sola operazione. Se l'ultimo bit sarà 1, allora il generatore funzionerà in modalità sequenziale, ovvero potrà effettuare più operazioni in sequenza. L'indirizzo associato a questo registro è 0x00000008 in esadecimale.
- MASK_OUTPUT_ENABLE_REGISTER = In questo registro a 32 bit viene memorizzata la maschera di uscita del pattern. Poichè i piedini di uscita del sistema sono in modalità tri-state, per uscire dall'alta impedenza hanno la necessità di un enable. Ognuno di questi bit abilita uno dei piedini di uscita. L'indirizzo associato a questo registro è 0x00000009 in esadecimale.

- MAX_SEQUENCE_REGISTER = In questo registro a 32 bit viene memorizzato il numero massimo di operazioni che possono essere eseguite in sequenza. L'indirizzo associato a questo registro è 0x0000000A in esadecimale.
- MEMORY = L'indirizzo associato alla memoria è 0x0000000B in esadecimale. A differenza degli altri casi, questo non è un registro in su cui è possibile effettuare letture e scritture, bensì una memoria RAM. L'unica operazione che è possibile effettuare manualmente è la lettura, in quanto la scrittura avviene in automatico ogni volta che viene scritto il registro "DATA_REG". Se si vuole conoscere il valore salvato in memoria all'indirizzo presente in "ADDR_REG", è sufficiente scrivere il comando di lettura e l'indirizzo associato alla memoria per leggere il valore scritto in memoria.
- READDATA_REGISTER = In questo registro a 32 bit viene memorizzato il dato da leggere mediante il "Bridge SPI to Avalon". L'ingresso di questo registro è collegato ad un multiplexer su cui sono collegati tutti i registri sopra elencati. Una volta che il selettore del multiplexer viene assegnato, il dato viene salvato sul registro e verrà inviato al bridge. Non è stato assegnato un indirizzo a questo registro in quando viene selezionato automaticamente ogni volta che l'operazione di lettura viene asserita, e perché l'indirizzo scritto serve per settare il selettore del multiplexer.

Gli indirizzi associati a questi registri servono al componente "BRIDGE SPI TO AVALON" per potervi accedere ed effettuare le scritture e le letture. Infatti, a differenza di quando accadeva nel progetto del precedete, il bridge non è collegato direttamente ad una memoria RAM, ma a questi registri. Quindi le scritture e le letture avverranno sui registri e non direttamente alla memoria. In funzione dell'operazione da eseguire e dall'indirizzo inserito, la CU del generatore di pattern abiliterà il registro mediante il segnale di abilitazione corrispondente.

Una volta settati i vari registri, vediamo quale è il funzionamento dei vari moduli presenti nel datapath.

#### 3.2.2.1 Generatore di clock

Il "Clock Generator" è il primo blocco che viene attivato ed è il responsabile della generazione del segnale di clock utilizzato dal sistema. Il suo funzionamento si basa su un divisore di frequenza, che riduce la velocità del *clock* in ingresso secondo un valore configurabile.

Il "CLOCK_DIVIDER_REG" memorizza il divisore impostato dall'utente, ovvero il numero di fronti del clock in ingresso necessari per produrre una variazione del clock in uscita. Qualora il valore del divisore fosse impostato a zero, il "clock Generator" attiverà un flag di errore ($error\_flag\_divider\_zero$), indicando una configurazione non valida alla CU.

L'architettura del "Clock Generator" è basata quindi su un contatore che viene incrementato a ogni fronte del clock in ingresso. Quando il contatore raggiunge

il valore indicato nel divisore, il clock in uscita viene negato e il contatore viene azzerato, ricominciando il conteggio dall'inizio. In questo modo si ottiene un segnale di clock con una frequenza ridotta rispetto a quella iniziale.

Oltre al segnale *error_flag_divider_zero*, il blocco produce un segnale chiamato *Terminal_Count_Clock* che indica il raggiungimento del valore indicato dal divisore e quindi una variazione del clock di uscita. Questo segnale è utile alla CU per sincronizzare alcuni passaggi di stato.

Dopo aver generato il clock in uscita dal blocco, questo passa per un multiplexer. Una delle specifiche del progetto è quella di poter scegliere se utilizzare il clock appena generato, oppure un clock proveniente dall'esterno. Infatti, grazie al valore salvato nel registro "CLOCK_SELECTION_REGISTER" viene selezionato uno dei due clock, il quale sarà il segnale di temporizzazione del registro di generazione dei pattern.

#### 3.2.2.2 Gestione degli indirizzi

L'"ADDR Manager" è il blocco che si occupa della gestione degli indirizzi della memoria, aggiornandoli in base alle impostazioni scelte dall'utente. Il suo compito è quello di determinare l'indirizzo successivo da utilizzare durante la lettura o la scrittura dei dati.

Questo blocco riceve in ingresso un indirizzo iniziale e, a ogni ciclo di *clock*, valuta se incrementarlo di 1 o mantenerlo invariato. Se il segnale di abilitazione è attivo, l'"ADDR Manager" legge l'indirizzo corrente e lo incrementa di 1, generando un nuovo indirizzo da utilizzare nel ciclo successivo. Se invece viene attivato il segnale di *reset*, l'indirizzo torna a un valore iniziale.

Il nuovo indirizzo generato sarà poi l'ingresso di un multiplexer insieme al valore scritto nel registro "ADDR_REGISTER". In funzione della modalità di funzionamento dell'architettura, per ogni ciclo verrà scelto un solo indirizzo che verrà poi utilizzato dalla memoria per effettuare la lettura o la scrittura.

#### 3.2.2.3 Gestione del Trigger

Il blocco chiamato "TRIGGER MANAGER" ha il compito di attivare o disattiva il il segnale di trigger in funzione dei dati che riceve in ingresso. Oltre al segnale di temporizzazione, il "TRIGGER_MANAGER" dispone di un segnale di abilitazione. Quando questo è attivo, allora il blocchetto in questione comincia l'elaborazione dei dati, quando è invece disattivato, li ignora. Tra i dati in ingresso troviamo il pattern da generare (*pattern_in*), la maschera del trigger proveniente dal registro "TRIGGER_MASK_REGISTER", e la condizione che questi dovrebbero rispettare, proveniente dal registro "TRIGGER_CONDITION_REGISTER".

Una volta ricevuto il segnale di abilitazione, il "TRIGGER_MANAGER" legge i bit del *pattern_in* e li combina con la maschera tramite un'operazione di AND bit a bit. La maschera è una sorta di filtro che permette di selezionare solo determinati bit, ignorando gli altri. Una volta che il dato è stato filtrato dalla maschera, il

segnale viene confrontato con la condizione di trigger, cioè un valore specifico che i bit devono assumere affinché il segnale di *trigger_out* venga attivato. Se il dato filtrato corrisponde esattamente alla condizione di trigger, il sistema attiva il segnale di uscita, mettendo *trigger_out* a 1. Se, invece, il dato filtrato non corrisponde alla condizione di trigger, il segnale di uscita rimane a 0.

#### 3.2.2.4 Gestione del Loop

Il blocco chiamato "Loop Management" ha il compito di contare un certo numero di iterazioni ed indicare quando il conteggio ha raggiunto un valore massimo. Questo vuol dire che il "Loop Management" non è altro che un contatore programmabile. I segnali in ingresso al blocco sono il segnale di *clock*, il segnale di *reset* necessario per azzerare il conteggio, il segnale di abilitazione al conteggio, ed infine il segnale di *max_value* che proviene dal registro "MAX_SEQUENCE_REGISTER". Quando il segnale di abilitazione è attivo, per ogni colpo di clock il contatore viene incrementato di 1. Non appena il conteggio raggiunge il valore indicato dal *max_value*, un flag di uscita chiamato *end_loop* viene attivato e viene inviato alla CU.

#### 3.2.2.5 Generazione del Pattern Digitale

La generazione del pattern è l'operazione principale di questo sistema, ma che deve essere fatta dopo aver settato in precedenza tutti gli altri componenti. Una volta terminato il procedimento di impostazione del sistema, è sufficiente scrivere il registro "START_REG". Una volta campionato il segnale di start, iniziano tutti i controlli del caso. Avviene il controllo del divider maggiore di 0, viene controllato il trigger e la condizione, viene controllato il numero della ripetizione del loop. Se tutte le condizioni vengono rispettate, allora l'*enable* del registro "PATTERN_OUT_REG" viene abilitato e nel colpo di clock successivo, il segnale verrà messo in uscita. In particolare, questo dato verrà posto sui pin di I/O che pertanto saranno in tri-state. In funzione della maschera salvata nel registro "MASK_OUTPUT_ENABLE", questi pin di uscita verranno abilitati o no.

#### 3.2.2.6 Gestione delle letture

Quando si vuole effettuare la lettura di uno dei registri o della memoria, mediante i comandi esposti nel capitolo 2, si invia il codice di lettura e l'indirizzo associato al registro che si vuole leggere. Tutti i dati scritti nei vari registri sono posti in ingresso ad un multiplexer. Una volta ricevuto il comando di lettura e l'indirizzo associato, questo indirizzo viene inviato alla CU la quale lo decodificherà ed invierà il corrispondente segnale di selezione del multiplexer. In questo modo, solo il dato del registro scelto verrà posto in uscita del multiplexer. Una volta selezionato il dato, questo verrà salvato in un registro PIPO chiamato "READ_DATA_REG" e successivamente mandato al "Bridge SPI to Avalon" che lo invierà sul segnale "MISO" precedentemente discusso.

## 3.3 Unità di controllo del "Pattern Generator"

Una volta chiarito il funzionamento dell'architettura, è possibile analizzare l'unità di controllo del sistema. Il pallogramma di questa unità di controllo verrà suddivisa in più parti per questioni grafiche, ma è da considerarsi un'unica entità. La prima parte del pallogramma è rappresentato in figura 3.3.


**Figura 3.3:** Sezione dell'unità di controllo del "Pattern Generator" relativa alla scrittura dei registri

Questa sezione dell'unità di controllo evidenzia la scrittura dei registri di stato e controllo del progetto. Come abbiamo già spiegato in precedenza, i dati necessari per l'operazione di scrittura vengono inviati alla CU dal blocco "Bridge SPI to Avalon". Questo invierà un segnale di *read* o *write* ed un indirizzo che nell'unità di controllo verrà indicato come *setting*. Dopo aver avviato il sistema ed essere usciti dallo stato di reset (*s0_reset*), lo stato corrente diventerà *s1_idle*. Da qui, se il segnale di *start* non sarà stato ancora abilitato, in funzione del segnale di *read* o *write* e del *setting*, si passerà ad uno degli stati mostrati in figura 3.3. Ognuno di questi stati rappresenta la scrittura di un registro. Infatti, quando lo stato corrente della CU passa in uno di questi, si ottiene l'abilitazione del registro associato a quello stato, con la conseguente scrittura del dato in ingresso. In particolare:

| **Setting** | **Operazione** | **Stato di destinazione** | **Registro** |
|---|---|---|---|
| 0000 | Write | *s*1_*idle* | Nessuno |
| 0001 | Write | *s*2_*wr_clock_* _*divider_reg* | clock _divider_reg |
| 0010 | Write | *s*4_*wr_clock_* _*selection_reg* | clock_selection_reg |
| 0011 | Write | *s*6_*wr_addr_reg* | addr_reg |
| 0100 | Write | *s*8_*wr_data_reg* | data_reg |
| 0101 | Write | *s*10_*wr_trigger_* _*mask_reg* | trigger_mask_reg |
| 0110 | Write | *s*12_*wr_trigger_* _*condition_reg* | trigger_condition_reg |
| 0111 | Write | *s*14_*wr_start_reg* | start_reg |
| 1000 | Write | *s*16_*wr_working_* _*mode_reg* | working_mode_reg |
| 1001 | Write | *s*18_*wr_mask_OE_reg* | mask_OE_reg |
| 1010 | Write | *s*20_*wr_max_loop_reg* | max_loop_reg |

**Tabella 3.3:** Setting associato ai vari stati e registri per operazioni di scrittura

Una volta terminata la scrittura nei registri, si ritornerà in automatico allo stato di idle in attesa di un nuovo comando. L'unica differenza rispetto agli altri casi sta nello stato *s*8_*wr_data_reg*. Un volta scritto il registro *data_reg*, si ha in automatico la scrittura nella memoria RAM. Infatti, dallo stato *s*8_*wr_data_reg* si entrerà nello stato *s*33_*wr_data_on_memory*, nel quale si avrà l'effettiva scrittura in memoria. In base alla modalità di funzionamento del generatore (one-shot o sequenziale), cambierà lo stato successivo. Se la modalità è one-shot, allora l'operazione terminerà e lo stato futuro sarà lo stato *s*30_*done*. Se invece la modalità di funzionamento è sequenziale, allora lo stato futuro sarà *s*32_*update_addr_memory*, il quale aggiornerà l'indirizzo per la scrittura successiva. Lo stato successivo sarà lo stato *s*32_1_*update_addr_memory* che rappresenta uno stato di attesa in cui verrà campionato il nuovo indirizzo e verrà aggiornato il contatore dei loop. Se il flag *end_loop_reps* risulterà attivo, il limite massimo di ripetizioni sarà stato raggiunto e lo stato futuro sarà *s*30_*done*. Altrimenti, se il flag *end_loop_reps* risulta 0, allora non sarà stato raggiunto il limite massimo di ripetizioni e lo stato futuro sarà *s*1_*idle*.

Una volta compreso come avvengono le scritture dei registri, è opportuno analizzare anche le letture di essi. Nella figura 3.4 viene mostrata la sezione del pallogramma relativa alle letture dei registri.

| **Setting** | **Operazione** | **Stato di destinazione** | **Registro** |
|---|---|---|---|
| 0000 | Read | *s*1_*idle* | Nessuno |
| 0001 | Read | *s*3_*rd_clock_divider_reg* | clock _divider_reg |

| | | | |
|---|---|---|---|
| 0010 | Read | *s*5_*rd*_*clock*_ _*selection*_*reg* | clock_selection_reg |
| 0011 | Read | *s*7_*rd*_*addr*_*reg* | addr_reg |
| 0100 | Read | *s*9_*rd*_*data*_*reg* | data_reg |
| 0101 | Read | *s*11_*rd*_*trigger*_ _*mask*_*reg* | trigger_mask_reg |
| 0110 | Read | *s*13_*rd*_*trigger*_ _*condition*_*reg* | trigger_condition_reg |
| 0111 | Read | *s*15_*rd*_*start*_*reg* | start_reg |
| 1000 | Read | *s*17_*rd*_*working*___ *mode*_*reg* | working_mode_reg |
| 1001 | Read | *s*19_*rd*_*mask*_*OE*_*reg* | mask_OE_reg |
| 1010 | Read | *s*21_*rd*_*max*_*loop*_*reg* | max_loop_reg |
| 1011 | Read | *s*34_*rd*_*data*_*memory* | ram_pattern |

**Tabella 3.4:** Setting associato ai vari stati e registri per operazioni di lettura

Per la lettura, il comportamento della macchina a stati è simile a quello visto nel caso della scrittura. Dallo stato di idle, in funzione del segnale di *read*, dello *start* e del *setting*,verrà abilitata la lettura di uno dei registri. Gli indirizzi sono ovviamente gli stessi di quelli esposti nella sezione relativa alla scrittura. A differenza della scrittura però, nei vari stati mostrati in figura 3.4 non si accederà ai registri, bensì al segnale di selezione del multiplexer a 16 ingressi. Ciò accade perché le uscite dei vari registri saranno già disponibili ad essere lette e pertanto risulteranno già in ingresso al multiplexer. Con il segnale di *setting* verrà impostato il selettore del multiplexer, il quale renderà il segnale scelto disponibile al registro *Read_data_register*. Grazie al segnale di *read*, verrà attivato il segnale di abilitazione di questo registro che invierà successivamente il dato letto al "Bridge SPI to Avalon" per l'invio sul *miso*. Terminata l'operazione di lettura, si passerà allo stato *s*30_*done* che attiverà un flag che indichi la conclusione dell'operazione. Dopo di ciò si ritornerà allo stato di idle in attesa di una prossima istruzione.

Una volta scritto il segnale di *start* nel registro *start_reg* ed essere ritornati in idle, si avvierà in automatico la generazione del pattern. Osserviamo nel dettaglio la struttura della macchina a stati in figura 3.5.

Una volta campionato il segnale $start = 1$, si entrerà nello stato $s22\_divider1\_check$ in cui verrà controllato il valore del $DIVIDER$ scritto nel $CLOCK\_GENERATOR$. Se il valore campionato è uno 0, verrà campionato il flag $ERROR\_FLAG\_DIVIDER = 1$ e si entrerà nello stato di errore, terminando la generazione del pattern. Se invece il $DIVIDER$ è diverso da 0, allora il flag sarà disattivato e si entrerà nello stato $s23\_loop\_active$ in cui si abiliterà il $LOOP\_MANAGER$. Anche in questo caso si effettuerà un controllo, in particolare sul flag $END\_LOOP\_REPS$. Se risulta essere uguale ad 1, allora si andrà nello stato di errore poiché sarà già stato raggiunto


**Figura 3.4:** Sezione dell'unità di controllo del "Pattern Generator" relativa alla lettura dei registri

il numero massimo di ripetizioni in sequenza, se invece il flag risulta essere 0, si proseguirà nello stato $s24\_memory\_active$. Qui avverrà la lettura del dato nell'indirizzo di memoria corretto, il quale verrà inviato al $TRIGGER\_MANAGER$.

Infatti lo stato successivo risulta essere $S25\_trigger\_active$, in cui verrà dato il segnale di abilitazione al $TRIGGER\_MANAGER$. In questo colpo di clock il $TRIGGER\_MANAGER$ campionerà l'*enable* e comincerà ed effettuare le verifiche necessarie tra pattern da generare, maschera e condizione. Al colpo di clock successivo si andrà nello stato $s26\_trigger\_value$ in cui verrà campionato il segnale di $TRIGGER\_OUT$. Se $trigger\_out = 0$ allora la condizione non sarà stata rispettata e si andrà nello stato di errore, se invece risulta essere uguale ad 1 si andrà nello stato $s27\_wait\_clock\_sample$, ovvero lo stato di attesa in cui avverrà la sincronizzazione con il clock generato mediante il "clock generator".

Campionato il fronte del clock, si passerà allo stato $s28_pattern_generator$ in cui si avrà l'abilitazione del registro $PATTERN\_OUT\_REG$ con la conseguente scrittura del dato in ingresso e l'invio del dato verso il "Bridge SPI to Avalon". A questo punto, se la modalità di funzionamento sarà one-shot, l'algoritmo terminerà andando


**Figura 3.5:** Sezione dell'unità di controllo del "Pattern Generator" relativa alla generazione dei pattern

nello stato $s30\_done$. Se invece la modalità selezionata sarà sequenziale, sarebbe necessario effettuare alcuni controlli ed eventualmente aggiornare l'indirizzo. Se al termine della generazione il segnale $end\_loop\_reps = 1$ e $working\_mode = 1$, allora anche il processo sequenziale terminerà correttamente e si andrà nello stato $s30\_done$. Se invece $end\_loop\_reps = 0$ e $working\_mode = 1$, sarà necessario aggiornare l'indirizzo di memoria su cui leggere il pattern per effettuare la generazione successiva. Si andrà quindi negli stati $s29_update\_addr\_loop$, che attiva il segnale di abilitazione dell'$ADDR\_MANAGER$, e nello stato $s29\_1\_update\_addr_loop$ in cui si selezionerà il selettore del multiplexer che gestisce gli indirizzi.

Terminato l'aggiornamento degli indirizzi, si andrà nello stato di idle in attesa di un nuovo segnale di *start*. La differenza tra la modalità one-shot e sequenziale è che se fossimo in modalità one-shot, ogni volta che campioneremmo il segnale di start, verrebbe generato sempre lo stesso pattern perché l'indirizzo resterebbe sempre lo stesso. In modalità sequenziale invece, per ogni start campionato avremo la generazione di pattern differenti, scritti in locazioni di memoria consecutive.

## 3.4 Timing del "Pattern Generator"

Come fatto nel capitolo precedente, verrà analizzato il timing diagram del generatore di pattern. Per quanto riguarda la scrittura dei registri, il comportamento dal punto di vista del "Bridge SPI to Avalon" è quello discusso nel capitolo precedente. Andiamo quindi ad analizzare il comportamento dei segnali dal punto di vista del generatore di pattern. Il primo timing che andremo ad analizzare sarà quello relativo alla scrittura dei registri di controllo e di dato ed alla generazione del pattern (figura 3.6).

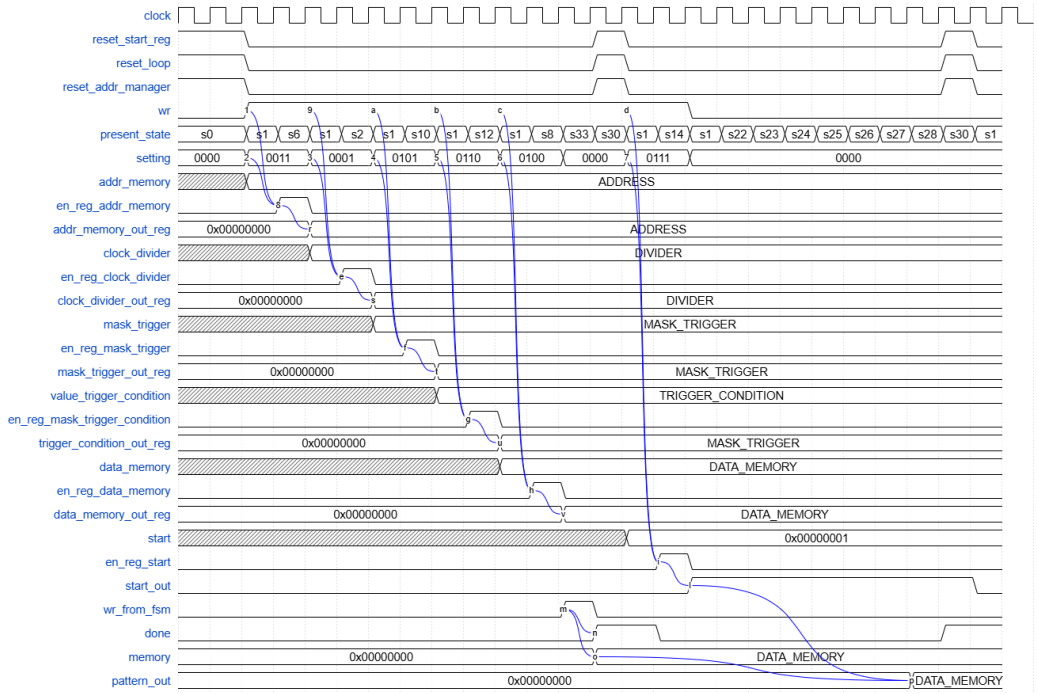


**Figura 3.6:** Timing diagram relativo alla scrittura dei registri e alla generazione del pattern

Come possiamo vedere dall'immagine 3.6, la prima cosa da fare per utilizzare correttamente il generatore di pattern è quello di scrivere i registri di dato e controllo. Una volta usciti dallo stato di *reset*, ci si troverà nello stato di idle, in attesa di istruzioni. Una volta ricevuti i segnali di *wr* e di *setting*, inizierà la scrittura nei registri. Nel timing mostrato vengono scritti i registri relativi all'indirizzo, al divisore della frequenza di clock, alla maschera del trigger, alla condizione del trigger ed al pattern da salvare in memoria. Infatti, una volta scritto il dato nel registro, avverrà in automatico il salvataggio del dato in memoria ed una volta concluso, il segnale di *done* si attiverà. Questi registri sono quelli essenziali per avere un corretto funzionamento del generatore. Non essendo stato scritto, il registro *working_mode* resta nel suo valore di default e quindi la modalità di funzionamento è one-shot. In questo modo non è necessario settare il registro *max_loop*. Anche quello relativo al selettore di clock resta al valore di default, e quindi si utilizzerà il clock interno.

Una volta terminata la procedura di impostazione, per avviare la generazione sarà necessario scrivere il registro di start. Campionato il segnale di $start\_out = 1$,

inizierà la fase di controllo del divisore, del loop e del trigger. Se tutti i requisiti saranno rispettati, allora inizierà la generazione del pattern scrivendolo sul registro *pattern_out* e successivamente inviandolo sui pin di I/O.

Osserviamo nel dettaglio il funzionamento del trigger, che rimarrà invariato sia nel caso di modalità sequenziale, sia nel caso di modalità one-shot.

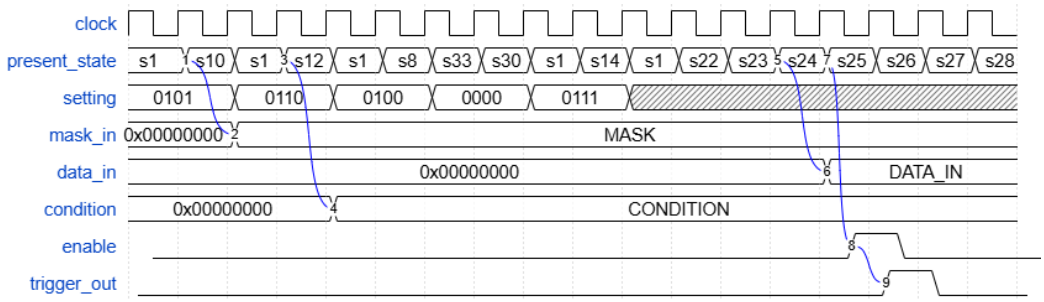


**Figura 3.7:** Timing diagram relativo al Trigger Manager

Il comportamento del trigger manager è molto semplice come si può vedere dalla figura 3.7. Ogni volta che viene aggiornato uno tra i registri di *mask_trigger_reg* e *mask_trigger_condition_reg*, in automatico si aggiorneranno gli ingressi del trigger manager. Una volta avviata la generazione del pattern, quando questo verrà letto dalla memoria, entrerà nel trigger manager. Quando si entrerà nello stato *s25_trigger_active*, verrà abilitato il segnale di *enable* del manager. Qui il segnale *data_in* verrà prima mascherato con il valore indicato da *mask_in* e successivamente si verificherà se il risultato ottenuto e la condizione indicata dal segnale *condition* sono uguali. Se la risposta è affermativa, il segnale di *trigger_out* si attiverà, altrimenti resterà a 0.

Osserviamo adesso come si comporta il generatore di pattern quando il divider è diverso da 1 e maggiore di 0. In figura 3.8 in particolare, viene rappresentato il caso di divisore pari a 9.

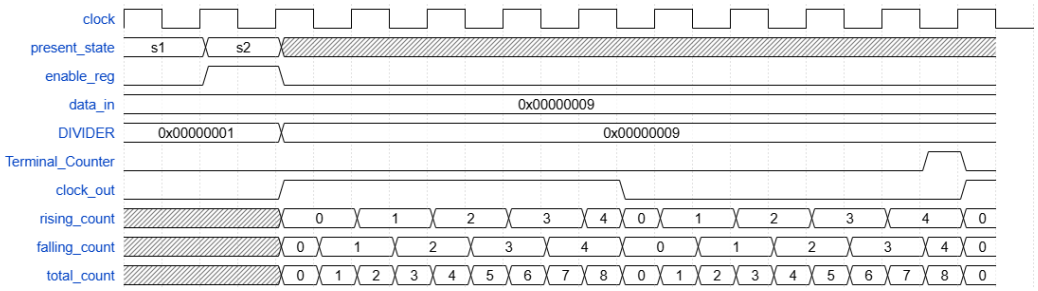


**Figura 3.8:** Timing diagram relativo al clock Generator

Una volta configurato il registro *clock_divider_reg*, il nuovo valore verrà immediatamente campionato dal generatore di clock. Il generatore, essendo sempre attivo non necessita di un segnale di abilitazione e quindi inizierà immediatamente la generazione con il nuovo clock. Per generare il segnale di *clock_out* verranno sfruttati due contatori e un sommatore. Infatti il primo contatore, terrà il conto dei fronti di salita del clock mentre il secondo dei fronti di discesa. Il sommatore avrà

invece il compito di sommare i due valori. In questo modo, quando la somma sarà uguale al valore del divisore, il segnale *clock_out* verrà negato, simulando così un fronte del clock.

Infine osserviamo come funzionano il loop manager e l'address manager nel caso in cui si lavori in modalità sequenziale. Il timing viene mostrato in figura 3.9.

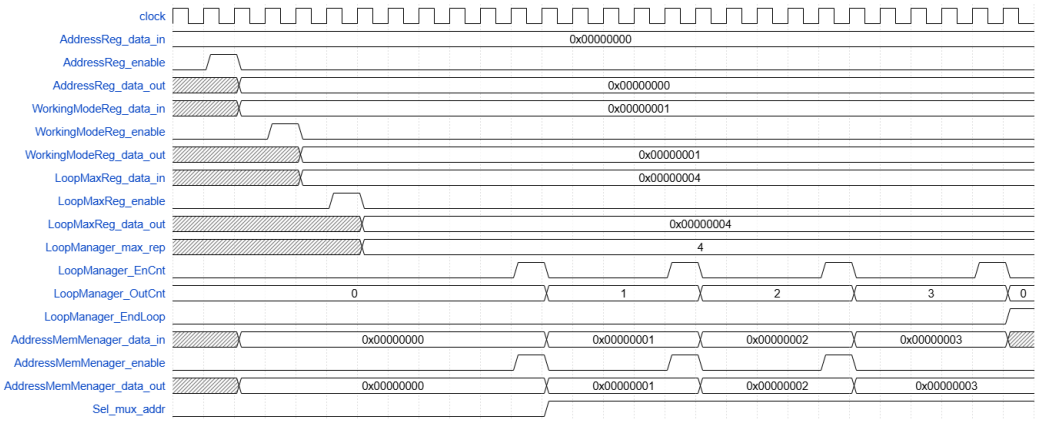


**Figura 3.9:** Timing diagram relativo al Loop Manager e all'Address Manager in caso di modalità sequenziale

Il funzionamento di questi due blocchi è strettamente legato l'uno all'altro. Infatti se la modalità di funzionamento fosse one-shot, l'address manager ed il loop manager sarebbero disattivati, in quanto la gestione degli indirizzi successivi ed il conteggio delle ripetizioni sarebbero inutili. Nel momento in cui si imposta il generatore di pattern in modalità sequenziale mediante la scrittura dei dati nei registri corrispondenti, il segnale di abilitazione dell'address manager e del contatore del loop manager verranno attivati nei momenti corretti. In questo modo verranno aggiornati sia gli indirizzi che il conteggio delle sequenze. Infine, si noti come, una volta che il contatore è diverso da 0, il selettore del multiplexer posto in uscita all'address manager assume valore 1, in modo da utilizzare l'indirizzo aggiornato e non quello scritto nel registro *addr_register*.

Osserviamo infine come avviene la lettura dei registri di dato stato e controllo, in modo da aver approfondito tutti gli aspetti del generatore. Il timing viene mostrato in figura 3.10.

Quando dal "Bridge SPI to Avalon" vengono inviati i segnali di *read* e di *setting*, la CU gestisce questi segnali ed invia il segnale di selezione del multiplexer a 16 ingressi ed il segnale di abilitazione del registro di lettura. In questo modo, in funzione del segnale di selezione, verrà scelto il dato da leggere e portare all'esterno del multiplexer, ed in funzione del segnale di abilitazione, questo dato verrà scritto nel registro *readdata_reg*. Una volta scritto nel registro, questo verrà inviato al "Bridge SPI to Avalon" che lo gestirà e lo invierà sulla linea del *miso*.

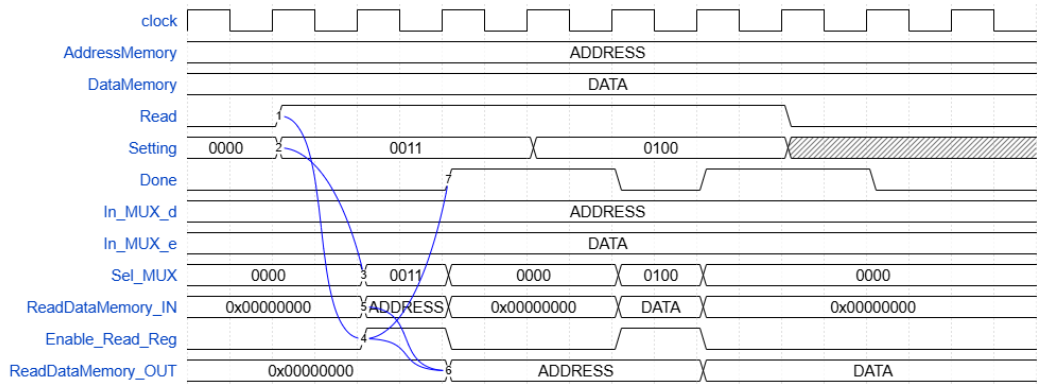


**Figura 3.10:** Timing diagram relativo alla lettura dei registri di dato, stato o controllo

## 3.5 Simulazione del comportamento del "Pattern Generator"

Una volta conclusa la spiegazione dei timing, si può procedere con la simulazione delle varie parti appena descritte.

La figura 3.11 mostra la simulazione della scrittura dei registi di stato, dato e controllo, compreso il registro di start, il quale avvia la generazione del pattern.

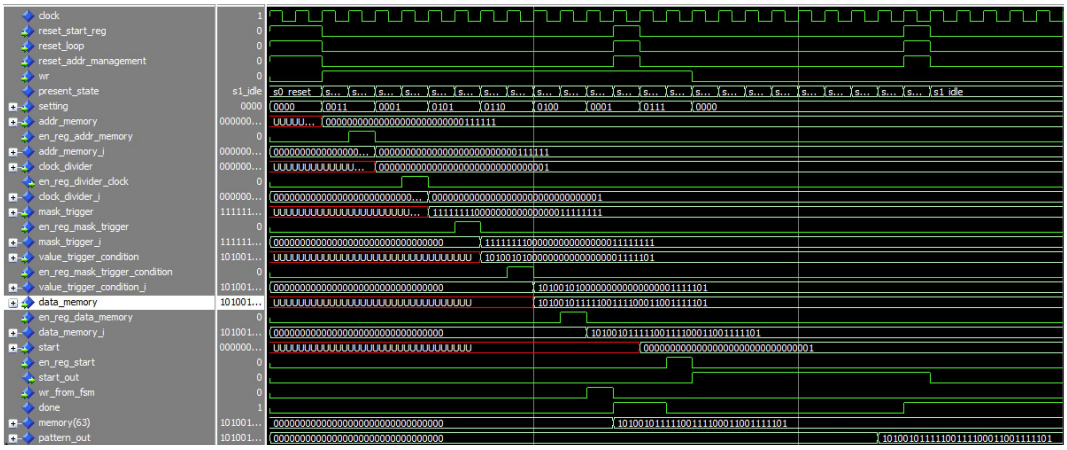


**Figura 3.11:** Simulazione relativa alla scrittura dei registri e alla generazione del pattern

La riuscita della generazione del pattern in questa immagine è evidenziata dal corretto valore sul segnale *pattern_out* e dal segnale di *done*. Inoltre, i diagrammi temporali mostrati corrispondono a quelli mostrati in figura 3.6. Anche in questo caso sono stati scritti prima i registri dell'indirizzo, del divisore di clock, della maschera del trigger, della condizione del trigger ed infine del pattern da salvare in memoria. Non appena viene scritto il dato sul registro *data_memory_reg*, il pattern viene anche salvato in memoria all'indirizzo corrispondente al valore presente nel registro di indirizzo. Una volta salvato il dato in memoria, il segnale di *done* si attiva.

Dopo aver scritto il valore 0x00000001 sul registro di *start*, il valore appena scritto in memoria viene copiato sul registro *pattern_out_reg* ed avviene la generazione.

La presenza di questi dati sui pin di uscita verrà mostrata nel paragrafo successivo.

Appurato il corretto funzionamento del generatore, analizziamo il comportamento del "Trigger Manager" con la simulazione rappresentata nell'immagine 3.12.


**Figura 3.12:** Simulazione relativa al Trigger Manager

Una volta aggiornati i registri di *mask_trigger_reg* e *mask_trigger_condition_reg*, gli ingressi del trigger manager sono pronti ed in attesa del pattern generato. Non appena viene avviata la generazione, il pattern viene letto dalla memoria e diventa disponibile al Trigger Manager.

Una volta attivato il segnale di abilitazione, il segnale *data_in* viene mascherato con *mask_in* e successivamente avviene il confronto con la condizione indicata dal segnale *value_trigger_condition*. Se il confronto, effettuato mediante una porta *AND* bit a bit, ha avuto esito positivo, il segnale di *trigger_out* si attiva, altrimenti resta a 0. Anche in questo caso, la simulazione rappresenta perfettamente il comportamento pensato durante la fase di progettazione del modulo.

Si proceda adesso con la simulazione del generatore di clock mostrato in figura 3.13.

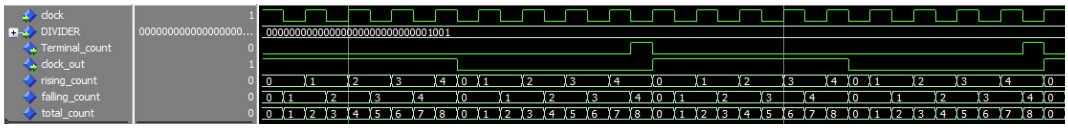


**Figura 3.13:** Simulazione relativa al Generatore di Clock

Si osservi come il divider sia impostato a 9 e di conseguenza, quando la somma dei due contatori arriva a contare 9 fronti, si attiva il terminal count e il segnale di *clock_out* viene negato. Dopo di che il conteggio riparte da 0 automaticamente.

Per quanto riguarda il funzionamento del loop manager e dell'address manager, si osservi la figura 3.14.

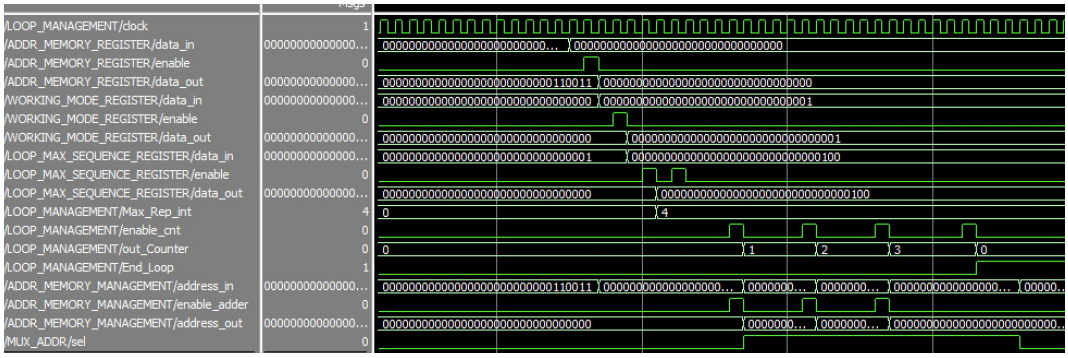


**Figura 3.14:** Simulazione relativa al Loop Manager e all'Address Manager

Impostati i registri associati all'indirizzo, alla modalità di funzionamento ed al numero massimo di ripetizioni, si attiva il *loop_manager* e l'*address_manager*. Essendo l'indirizzo di partenza 0x00000000 ed essendo il numero massimo di ripetizioni impostato a 4, si osservi come ad ogni operazione, si attiva l'enable del contatore del *loop_manager* e del sommatore dell'*address_manager*. Il conteggio del loop passa da 0 ad 1 e l'indirizzo passa da 0x00000000 a 0x00000001. Ad ogni attivazione dei segnali di enable, questi vengono aggiornati fino a quando non si raggiunge il numero massimo di ripetizioni e si attiva il segnale *end_loop_reps* e il conteggio del loop manager ricomincia da 0. In questo caso l'indirizzo finale sarà 0x00000003. Il segnale di selezione del multiplexer è quello che gestisce l'indirizzo di ingresso sia in memoria che nel *address_manager*. Infatti si osservi come questo segnale passi da 0 ad uno quando gli enable vengono attivati per la prima volta.

Osserviamo infine come avviene la lettura dei registri di stato e controllo in figura 3.15.

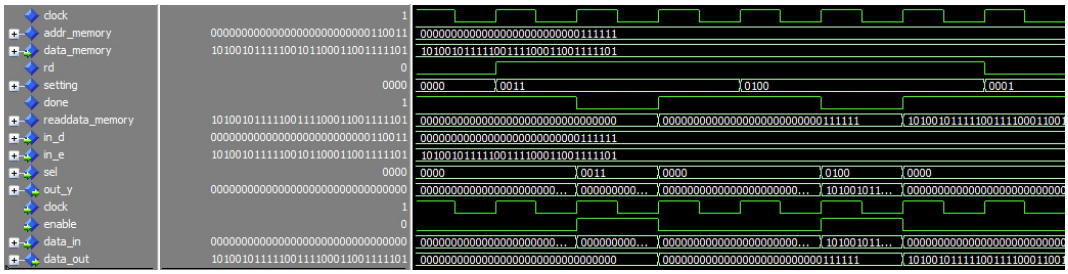


**Figura 3.15:** Simulazione relativa alla lettura dei registri

Quando il segnale di *read* viene attivato, in funzione del segnale di *setting* viene impostato il segnale di selezione del multiplexer a 16 ingressi ed il segnale di abilitazione del registro di lettura. In questo modo, il dato scelto passerà attraverso il multiplexer e verrà salvato sul registro *readdata_reg*, che successivamente lo renderà disponibile al "Bridge SPI to Avalon".

## 3.6 Simulazione del sistema "Bridge SPI to Avalon" - "Pattern Generator"

A questo punto possiamo considerare terminata l'analisi del generatore di pattern inteso come componente, ma possiamo osservare come si comporta a livello di sistema, cioè con il componente "Bridge SPI to Avalon" collegato. Inserire tutti i segnali utilizzati nel progetto creerebbe solamente confusione. Per questo motivo, nelle immagini inserite in questo documento vedrete solamente i segnali principali, ovvero quelli che permettono di osservare il proseguimento delle operazioni nel tempo. Questa scelta è anche giustificata dal fatto che il comportamento dei vari blocchi è già stato analizzato nel dettaglio.

Nella figura 3.16 viene mostrato il comportamento del sistema durante la scrittura di uno dei registri del generatore di pattern. Come si può osservare, l'istruzione che identifica la scrittura di uno specifico dato in uno specifico registro, viene mandata

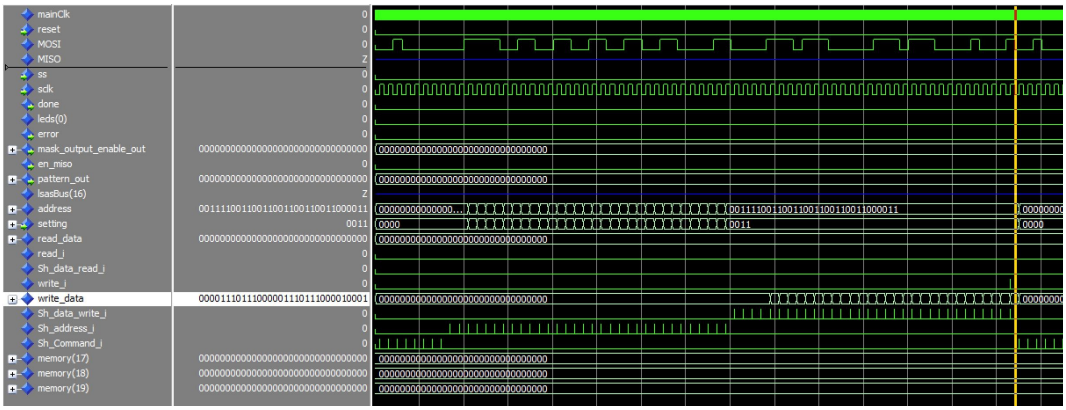


**Figura 3.16:** Simulazione relativa alla scrittura di un registro dal punto di vista sistemistico

dal microprocessore mediante l'interfaccia SPI ed in particolare nella linea *MOSI*. Il dato che proviene dalla linea *MOSI* viene campionato e decodificato, identificando prima l'operazione da fare (scrittura in questo caso), poi l'indirizzo su cui effettuare l'operazione (in questo caso *address_reg* del generatore di pattern) ed infine il dato da scrivere. Questi dati verranno decodificati dal blocco "Bridge SPI to Avalon" e verranno inviati al componente "Pattern Generator" che li gestirà sfruttando la CU interna ed abilitando i segnali utili per la scrittura del registro.

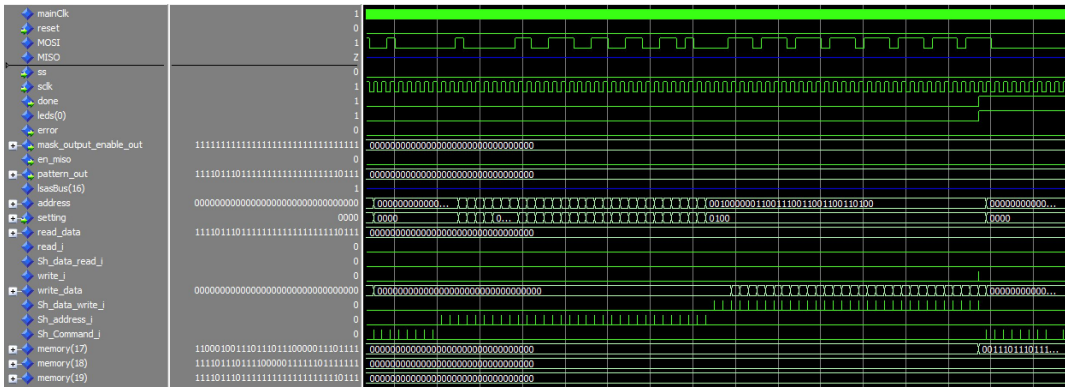


**Figura 3.17:** Simulazione relativa alla scrittura del pattern in memoria in modalità one-shot dal punto di vista sistemistico

Nella figura 3.17 invece viene mostrata la scrittura del dato nel registro *data_reg* del generatore. Come abbiamo già spiegato in precedenza, quando il dato viene scritto in questo registro, viene automaticamente scritto anche in memoria nell'indirizzo presente nel registro *addr_reg*. In particolare, nella simulazione viene scritto il dato appena campionato nell'indirizzo 17 della memoria. In concomitanza con la scrittura in memoria, si attiva il segnale di *done*, collegato direttamente con il led[0], che si accende una volta terminata l'operazione.

Nella figura 3.18 invece viene mostrata la lettura del dato contenuto in memoria. Come già spiegato nel capitolo relativo al "Bridge SPI to Avalon", una volta campionato il comando di lettura e l'indirizzo relativo al registro in cui leggere, il dato viene salvato nel registro *readdata_reg*. Dopo di che, questo viene fatto traslare un

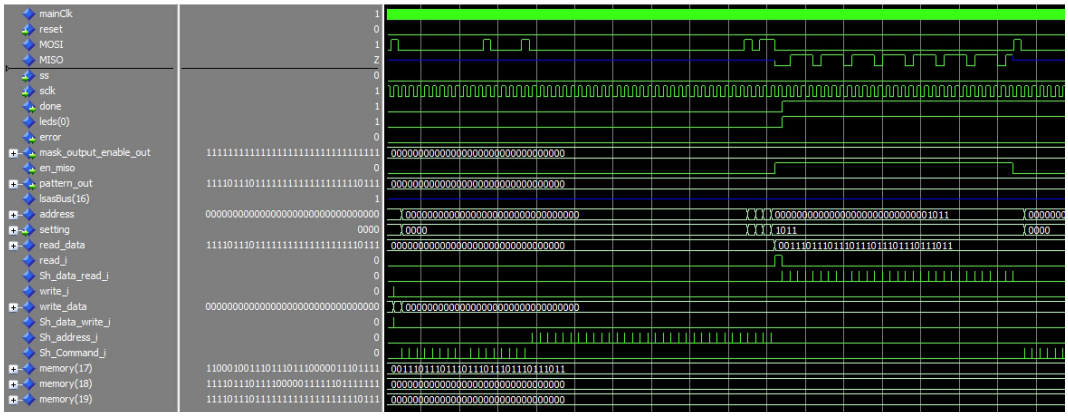


**Figura 3.18:** Simulazione relativa alla lettura del pattern scritto in memoria in modalità one-shot dal punto di vista sistemistico

bit per colpo di clock e viene posto sul segnale $MISO$ dell'interfaccia SPI. Essendo il $MISO$ posto in uno dei pin di I/O, ed essendo questi dei pin tri-state, è necessario attivare il segnale di enable del pin corrispondente. Nell'immagine si vede che il segnale *en_miso* resta attivo durante tutta la durata della trasmissione del dato. Una volta terminata, il segnale di enable si abbassa ed il $MISO$ torna nello stato di alta impedenza.

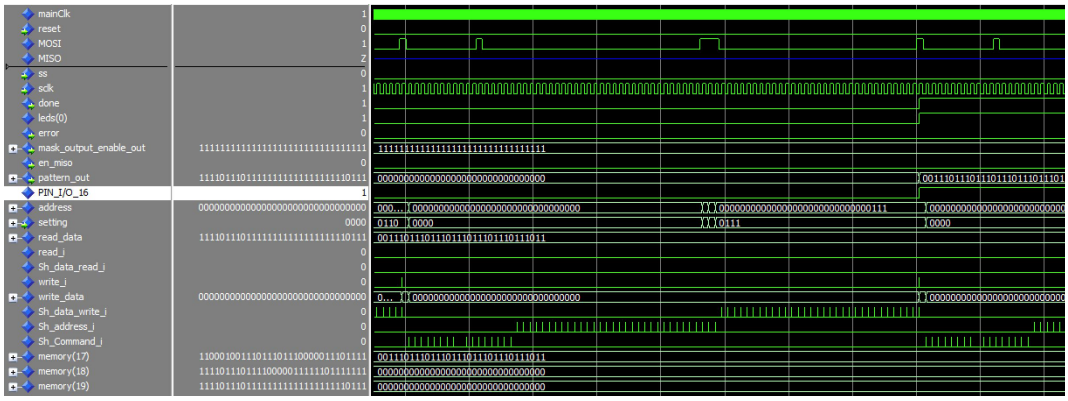


**Figura 3.19:** Simulazione relativa alla generazione del pattern presente in memoria in modalità one-shot dal punto di vista sistemistico

La generazione del pattern, mostrata in figura 3.19, avviene scrivendo il valore 1 nel registro di start. Una volta effettuata la scrittura, il generatore di pattern si avvierà e farà i vari controlli discussi in precedenza. Se questi vengono tutti rispettati, il dato verrà inviato sui pin di I/O della scheda. Nella figura, per evitare di intasare inutilmente l'immagine, viene mostrato esclusivamente il comportamento del pin[16], essendo gli altri uguali a livello comportamentale. Si osservi inoltre come la la maschera per gli *output_enable* sia composta di tutti 1. Ciò vuol dire che tutti i bit verranno inviati sui pin corrispondenti. Se per ipotesi, il bit numero 16 fosse stato impostato a 0, il segnale dell'immagine sarebbe rimasto in alta impedenza.

Nella figura 3.20 viene mostrata la scrittura in memoria in modalità sequenziale. Dopo aver impostato il numero massimo di ripetizioni a 3 nel registro *max_loop_reg*

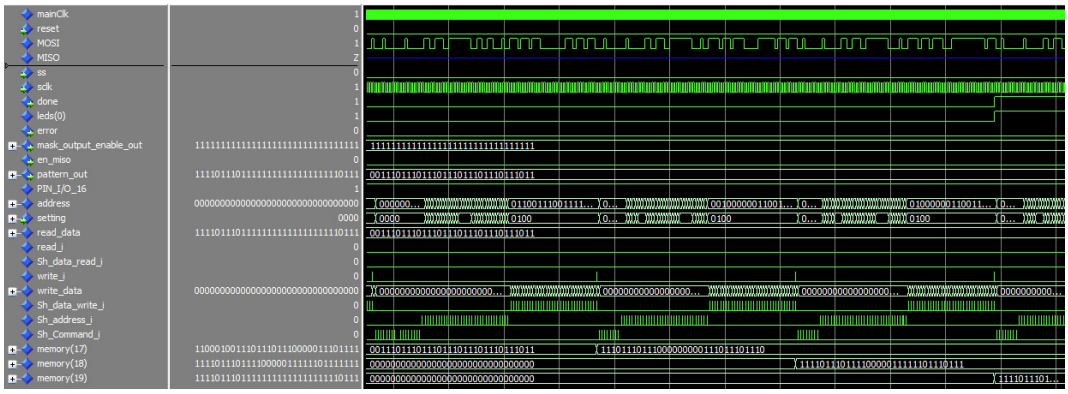


**Figura 3.20:** Simulazione relativa alla scrittura dei pattern in memoria in modalità sequenziale dal punto di vista sistemistico

ed aver impostato il *working_mode_reg* ad 1, la modalità sequenziale può funzionare senza problemi. In questo particolare esempio vediamo come viene scritto per 3 volte consecutive il registro *data_reg*. Se la modalità fosse stata one-shot, avremmo sovrascritto il dato sempre nello stesso indirizzo. In questo caso invece, i dati vengono scritti su 3 allocazioni di memoria consecutive. Infatti avremo che la scrittura avverrà nell'indirizzo presente nel registro *addr_reg* e quindi l'allocazione 17. La seconda scrittura invece avverrà nell'allocazione 18 e la terza nella 19. A differenza del caso one-shot, il segnale di *done* non si alza al termine di ogni scrittura, ma esclusivamente al termine della terza. In questo caso sono state fatte 3 operazioni di scrittura, ma è possibile anche effettuare 3 generazioni consecutive (figura 3.21), 3 letture consecutive (figura 3.22) o addirittura delle operazioni miste, il tutto considerando sempre l'incremento dell'indirizzo di riferimento ad ogni istruzione.

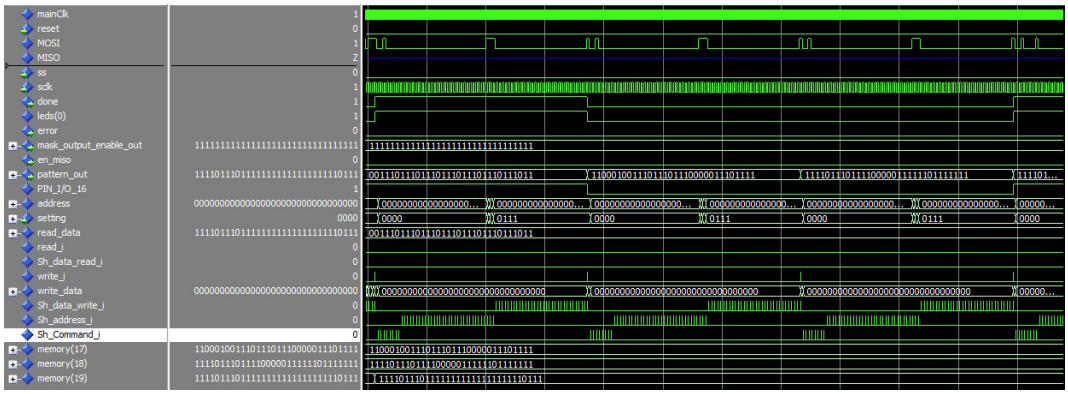


**Figura 3.21:** Simulazione relativa alla generazione dei pattern scritti in memoria in modalità sequenziale dal punto di vista sistemistico

Per completezza, in figura 3.23 viene mostrata la simulazione di un tipico utilizzo del generatore, ovvero la scrittura dei registri di dato e controllo, il salvataggio dei pattern in memoria, la lettura dei pattern e la generazione dei pattern sui pin di uscita, sia in modalità one-shot che sequenziale.

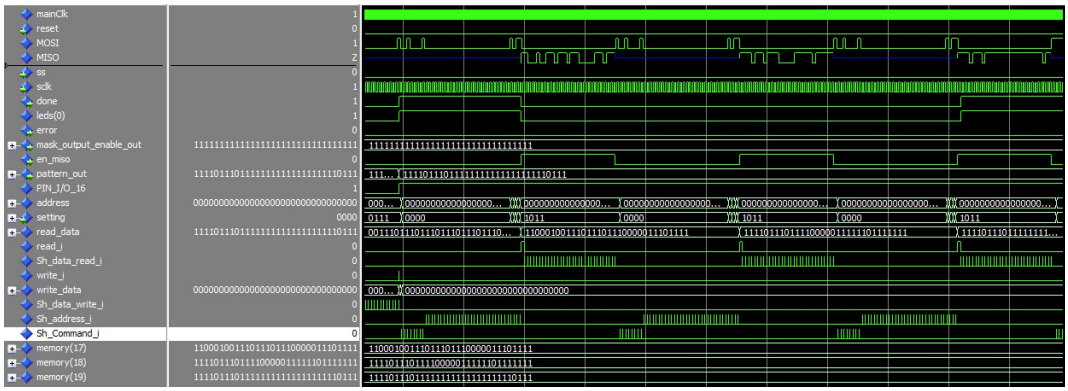


**Figura 3.22:** Simulazione relativa alla lettura dei pattern scritti in memoria in modalità sequenziale dal punto di vista sistemistico

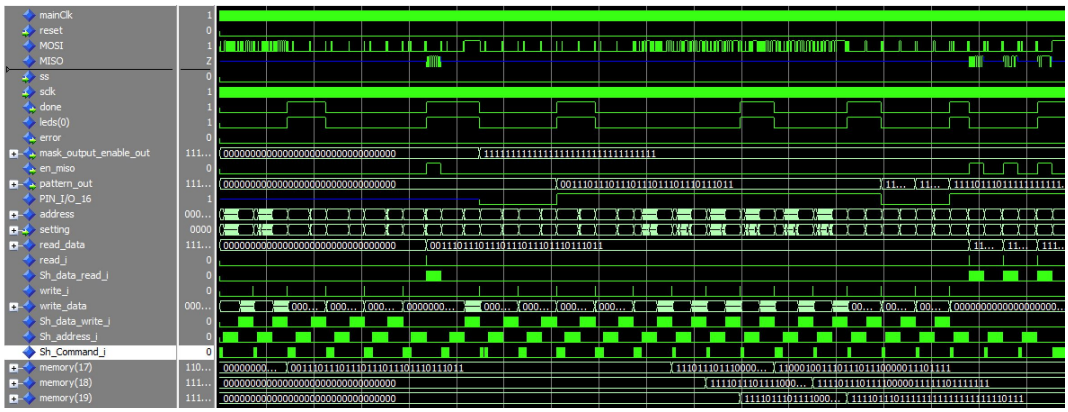


**Figura 3.23:** Simulazione completa dal punto di vista sistemistico

## 3.7 Risultato ottenuto

Una volta simulato il sistema ed aver verificato che il comportamento sia coerente con le specifiche richieste e progettate, si procede con il test sulla scheda VirtLAB. Anche in questo caso ci si è avvalsi del terminale dal quale interfacciarsi al sistema. E' stato programmato il microcontrollore con un file .elf, in modo tale da creare l'interfaccia testuale in cui inserire i comandi ed inviarli in modo corretto all'FPGA. Una volta collegata la VirtLAB in modo corretto ed aver programmato sia il microcontrollore che l'FPGA, apparirà la schermata presente in figura 3.24.

Questa interfaccia testuale è molto simile a quella del per il "Bridge SPI to Avalon" in quanto effettivamente anche questa interfaccia serve per comunicare con il bridge. La prima sezione infatti indica come scrivere le istruzioni, che è lo stesso di quanto visto in precedenza. La seconda sezione mostra invece gli indirizzi associati ai vari registri di dato e controllo. In particolare, essendo la rappresentazione dei bit in esadecimale, è sufficiente selezionare in modo corretto solamente l'ultimo bit, mentre gli altri verranno ignorati. La terza sezione invece indica alcune informazioni importanti per il corretto utilizzo del generatore di pattern, ovvero:

- Indica che il divisore di clock deve essere settato con un valore diverso da zero, altrimenti si entrerà in uno stato di errore.

```
****************************************
**   VirtLAB Pattern Generator (32bit)   **
****************************************

>?
Usage:
    HOW WRITE AND READ ON REGISTER:
    wrrrrrrrrddddddddd ---> Write 32 bit data 'dddddddd' into register 'rrrrrrrr'
    rrrrrrrrr          ---> Read 32 bit data from register 'rrrrrrrr'
    Register number (rrrrrrrr) is 32 bit, in hexadecimal format
    Data (dddddddd) is 32 bit, in hexadecimal format



    ADDRESS REGISTER:

    CLOCK_DIVIDER_REGISTER      ==> 00000001
    CLOCK_SELECTION_REGISTER    ==> 00000002
    ADDR_REGISTER               ==> 00000003
    DATA_REGISTER               ==> 00000004
    TRIGGER_MASK_REGISTER       ==> 00000005
    TRIGGER_CONDITION_REGISTER  ==> 00000006
    START_REGISTER              ==> 00000007
    WORKING_MODE_REGISTER       ==> 00000008
    MASK_OUTPUT_ENABLE_REGISTER ==> 00000009
    MAX_SEQUENCE_REGISTER       ==> 0000000a
    READ_DATA_ON_MEMORY         ==> 0000000b

    All others are UNUSED!


    OTHER USEFULL INFORMATION FOR DATA TO WRITE INSIDE REGISTERS!!
    1) CLOCK_DIVIDER_REGISTER:   Divider       != 00000000
    2) CLOCK_SELECTION_REGISTER: Internal_Clock = 00000000
                                 External_Clock = 00000001
    3) ADDR_REGISTER:            Max value     = 000000FF
    4) START_REGISTER:           Start         = 00000001
    5) WORKING_MODE_REGISTER:    One_Shot      = 00000000
                                 Sequence      = 00000001
```

**Figura 3.24:** Interfaccia testuale del "Pattern Generator"

- Indica che se si vuole utilizzare il clock interno della macchina, è necessario settare il registro corrispondente con il valore 0. Se invece si vuole utilizzare un clock esterno, deve essere settato con il valore 1.
- Indica che l'indirizzo più grande in memoria è 0x000000FF, che corrisponde alla locazione numero 255. Questo perché la memoria progettata è solo di 256 locazioni e quindi sono sufficienti 8 bit di indirizzo, rispetto a 32 a disposizione. Se i primi bit venissero settati con un valore diverso da 0, questi non avrebbero effetto.
- Indica che per iniziare la generazione del pattern, è necessario scrivere il valore 1 nel registro di start.
- Indica che per selezionare la modalità di funzionamento one-shot, è sufficiente settare il valore 0 nell'ultimo bit del *working_mode_reg*. Se si vuole utilizzare la modalità sequenziale è necessario settare questo bit con il valore 1.

Osserviamo il tipico utilizzo del generatore di pattern in modalità one-shot (figura 3.25).

```
****************************************
**   VirtLAB Pattern Generator (32bit)   **
****************************************

>w0000000100000001
Writing 00000001 to register 00000001
>w0000000300000000
Writing 00000000 to register 00000003
>w0000000500000001
Writing 00000001 to register 00000005
>w0000000600000001
Writing 00000001 to register 00000006
>w00000009ffffffff
Writing ffffffff to register 00000009
>w0000000401234ab1
Writing 01234ab1 to register 00000004
>r0000000b
Reading from register 0000000b: 01234ab1
>w0000000700000001
Writing 00000001 to register 00000007
```

**Figura 3.25:** Test del "Pattern Generator" in modalità one-shot

Per utilizzare in modo corretto il generatore di pattern in modalità one-shot, è necessario scrivere nei vari registri i dati da utilizzare per le impostazioni e i dati da generare. Il primo registro che è necessario scrivere è il *clock_divider_reg*, nel quale viene salvato un valore diverso da 0. In questo caso specifico è stato impostato il valore 0x00000001. Il secondo registro che è stato scritto è quello relativo alla locazione di memoria. In questo caso specifico è stato scritto il valore 0x00000000, cioè la prima locazione disponibile. Essendo appena usciti dallo stato di *reset*, è possibile non effettuare questa operazione di scrittura in quanto 0x00000000 è il valore di default dei registri. Il terzo registro che è stato scritto è quello relativo alla maschera del trigger. Per rendere l'operazione più semplice e comprensibile è stato scelto il valore 0x00000001, in modo tale da mascherare tutti i bit del pattern ad eccezione del primo. Il quarto registro che è stato scritto è quello relativo alla condizione del trigger. Per lo stesso motivo, la condizione scelta è stata 0x00000001. Questo vuol dire che, se il bit[0] del pattern è 1, allora il trigger verrà attivato e si potrà procedere con la generazione. Il quinto registro scritto è quello relativo alla maschera degli *output_enable*. Poichè al momento non abbiamo dei vincoli, abbiamo deciso di rendere disponibili in uscita tutti i bit del pattern. Il sesto registro scritto è quello relativo al pattern, cioè al dato da salvare in memoria. In questo caso specifico è stato salvato il valore 0x01234ab1, valore che rispetta la condizione di trigger. Questo dato verrà quindi scritto nel registro ed anche in memoria. Per osservare l'effettiva scrittura in memoria, l'istruzione successiva rappresenta la lettura del dato presente in memoria all'indirizzo scritto nel registro di indirizzo. Come si può osservare, il dato che viene letto è lo stesso scritto nel registro precedente. Infine, l'ultima istruzione scritta è quella relativa al registro di start, che è il registro responsabile della generazione. Non appena il valore 0x00000001 viene campionato, la generazione ha inizio e sui pin di I/O viene visualizzato il pattern generato.

Osserviamo adesso il tipico utilizzo del generatore di pattern in modalità sequen-

ziale in figura 3.26.

```
*****************************************
**   VirtLAB Pattern Generator (32bit)   **
*****************************************

>w0000000100000001
Writing 00000001 to register 00000001
>w0000000300000000
Writing 00000000 to register 00000003
>w0000000500000001
Writing 00000001 to register 00000005
>w0000000600000001
Writing 00000001 to register 00000006
>w00000009ffffffff
Writing ffffffff to register 00000009
>w0000000800000001
Writing 00000001 to register 00000008
>w0000000a00000003
Writing 00000003 to register 0000000a
>w0000000400000001
Writing 00000001 to register 00000004
>w0000000400000011
Writing 00000011 to register 00000004
>w0000000400000111
Writing 00000111 to register 00000004
>r0000000b
Reading from register 0000000b: 00000001
>r0000000b
Reading from register 0000000b: 00000011
>r0000000b
Reading from register 0000000b: 00000111
>w0000000700000001
Writing 00000001 to register 00000007
>w0000000700000001
Writing 00000001 to register 00000007
>w0000000700000001
Writing 00000001 to register 00000007
>
```

**Figura 3.26:** Test del "Pattern Generator" in modalità sequenziale

Il funzionamento del generatore di pattern in modalità sequenziale è molto simile alla modalità one-shot. I registri impostati per la modalità one-shot devono essere impostati anche in questa modalità. Oltre a questi verrà scritto il registro *working_mode_reg* con il valore 0x00000001 per indicare la modalità sequenziale, il registro *max_loop_rep* con il valore 0x00000003 (in questo caso, ma è sufficiente un qualsiasi numero maggiore di 0x00000000). Semplicemente impostando questi due registri, siamo passati dalla modalità one-shot alla quella sequenziale. A questo punto è possibile scrivere i pattern in memoria in modo sequenziale. Per farlo è stato scritto il registro *data_reg* tre volte consecutive con i valori 0x00000001, 0x00000011 e 0x00000111. Questi tre valori sono stati salvati in modo automatico nelle locazioni di memoria 0x00000000, 0x00000001 e 0x00000002. Per osservare la corretta scrittura di queste locazioni, sono state fatte 3 letture consecutive in memoria ed anche in questo caso abbiamo ottenuto i valori scritti in precedenza. Per concludere l'esposizione sull'utilizzo del generatore di pattern, è stata effettuata la tripla generazione dei pattern salvati ed appena letti. Quindi mediante terminale è stato scritto il valore 0x00000001 nel registro di start e per ogni generazione, abbiamo osservato sui pin di

uscita il pattern corrispondente.

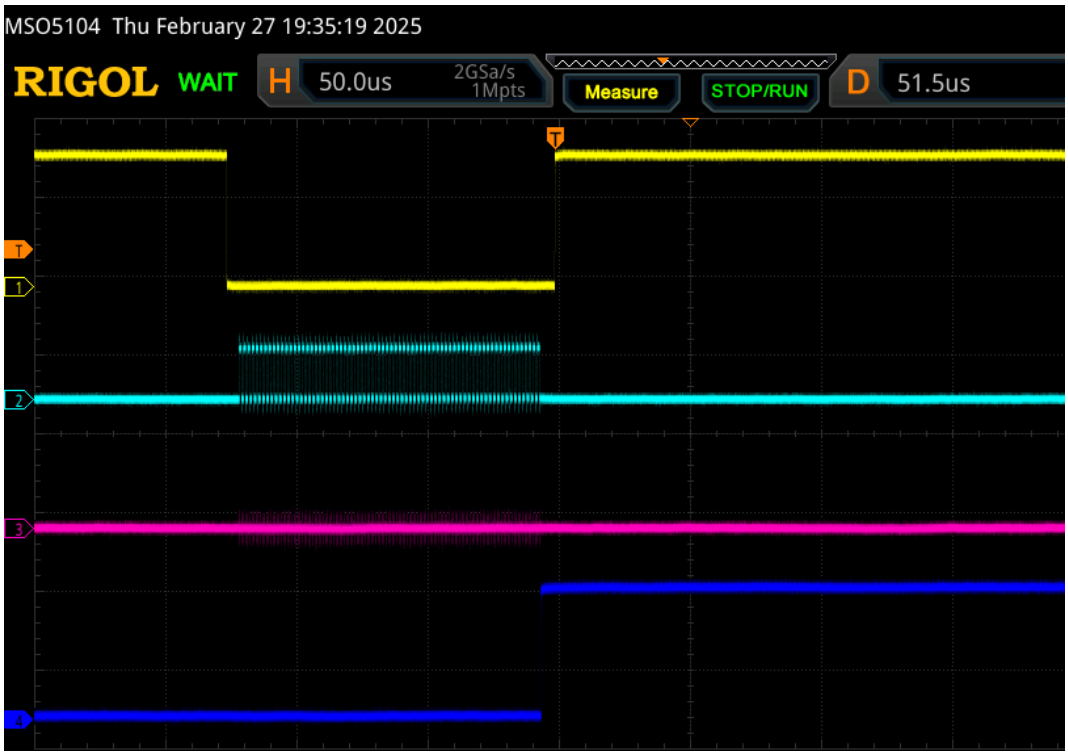


**Figura 3.27:** Comportamento reale di GPIO[0] e GPIO[1] aquisito mediante l'uso di un oscilloscopio

In figura 3.27 viene mostrato l'andamento di alcuni segnali quando avviene la generazione. Il segnale mostrato in giallo, è il segnale di *slave_select*, che assume un valore basso quando lo slave è selezionato. Il segnale in azzurro, invece, è il *serial_clock*, appunto un segnale di temporizzazione, attivo per tutta la durata dell'istruzione. Infine il segnale rosa mostra l'andamento del pin I/O[0] mentre quello blu il pin I/O[1]. Considerando il valore precedentemente scritto in memoria, ovvero 0x00000001, osserviamo che una volta decodificata l'istruzione, troviamo sui pin di uscita il corretto valore da generare.

### 3.7.1 Report Post-Sintesi

La sintesi del "Pattern Generator" ha evidenziato alcune caratteristiche che è opportune riportare anche in questo documento. In particolare, è opportuno mostrare il numero di elementi logici utilizzati per la sintesi, la quantità di memoria occupata e la frequenza massima di funzionamento senza l'utilizzo di tecniche di ottimizzazione. I valori sono riportati nella tabella (4.5).

| Memoria occupata | Elementi Logici | Frequenza Massima |
|---|---|---|
| 8192/608256 (1%) | 1162/24624(5%) | 49,99 MHz |

**Tabella 3.5:** Compilation Report relativo al "Pattern Generator"

Si osservi come sia la memoria utilizzata, sia gli elementi logici sono ben al di sotto del limite strutturale imposto dall'FPGA. Utilizzare solamente l'1% della memoria disponibile indica che il progetto è poco esigente in termini di memoria e lascia molto margine per future espansioni o aggiunta di nuove funzionalità. Inoltre anche il numero di LE è soddisfacente(5%) in quanto dimostra un utilizzo efficiente delle risorse, senza eccessivo spreco. Per quanto riguarda la massima frequenza (49,99Mhz) è anche questa soddisfacente per il target a cui è destinato il progetto. In ogni caso, se fosse necessario aumentare la frequenza, si potrebbero applicare tecniche di pipeline o timing optimization per aumentarne la frequenza.

# Capitolo 4

# Analizzatore di stati logici

## 4.1 Descrizione del progetto e risultati attesi

Nel mondo dell'elettronica digitale, la possibilità di analizzare e monitorare i segnali digitali è essenziale tanto quanto la loro generazione. Per questa ragione, è stato sviluppato anche un analizzatore di stati logici che consente di acquisire e interpretare i pattern inviati dall'utente.

Il cuore dell'analizzatore di stati logici è simile a quanto visto nel generatore di pattern. Infatti è un'architettura configurabile dinamicamente, che presenta le seguenti funzioni principali:

- Clock Generator: gestisce la temporizzazione della cattura dei dati;
- Loop Manager: controlla il numero di acquisizioni consecutive;
- Trigger Manager: definisce le condizioni di acquisizione dei segnali;
- Glitch Detector: segnala la presenza di glitch durante l'acquisizione del segnale.

Anche in questo caso, la configurazione dei registri di stato, controllo e dato avviene tramite il "Bridge SPI to Avalon". Dopo aver impostato i parametri di acquisizione e verificato i vincoli definiti dall'utente, i dati catturati vengono salvati in memoria, da cui possono essere successivamente letti e analizzati.

Lo scopo di questo capitolo è illustrare nel dettaglio l'architettura e il funzionamento dell'analizzatore di stati logici.

## 4.2 Architettura del "Logic Analyzer

Lo scopo del progetto è quello di creare un analizzatore di stati logici, ovvero un blocco che riceva in ingresso una sequenza di stati logici, li campioni e ne salvi il valore in memoria. Per la configurazione dei registri viene usato il "Bridge SPI to Avalon" come già fatto per il generatore di pattern. Da un punto di vista ad alto livello, possiamo considerare il sistema dell'analizzatore come composto dai seguenti blocchi:

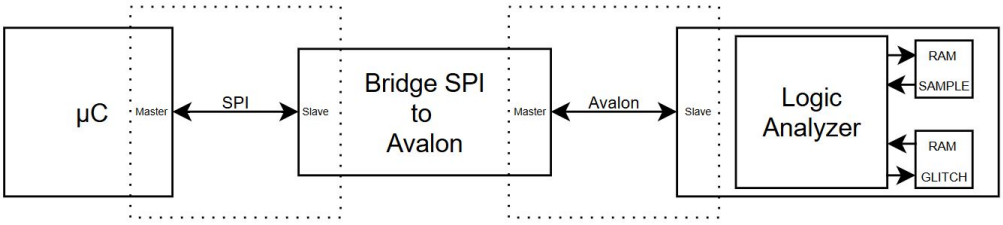


**Figura 4.1:** Blocchi che compongono il sistema "Logic_Analyzer"

### 4.2.1 Specifiche di progetto

Nella realizzazione di questo progetto è stato sfruttato il "Bridge SPI to Avalon" per la scrittura dei registri di controllo, affinché l'utente possa utilizzare anche l'analizzatore in modo flessibile. Le specifiche di questo progetto richiedevano lo sviluppo di alcune caratteristiche fondamentali per un analizzatore di stati logici, che sono compatibili con quelle del generatore. In particolare si richiede di:

- Avere la possibilità di modificare la frequenza di funzionamento del sistema mediante un divisore del clock base.
- Avere la possibilità di selezionare un clock esterno, piuttosto che uno interno.
- Avere la possibilità di individuare dei glitch durante il campionamento del segnale di ingresso.
- Avere la possibilità di inserire manualmente l'indirizzo di memoria in cui salvare i campioni acquisiti ed i glitch associati.
- Avere la possibilità di inserire una maschera di trigger che indichi i bit da analizzare e quelli da ignorare.
- Avere la possibilità di inserire la condizione tale per cui il trigger si attivi e dia il via libera al salvataggio dei campioni in memoria.
- Avere la possibilità di selezionare la modalità di funzionamento dell'analizzatore tra sequenziale e one-shot.
- Avere la possibilità di inserire un numero massimo di analisi senza l'aggiornamento manuale dell'indirizzo di riferimento.
- Avere la possibilità di leggere il contenuto dei registri di stato e controllo, oltre che delle memorie.

Per ognuna di queste specifiche è stato progettato un blocco funzionale o un registro.

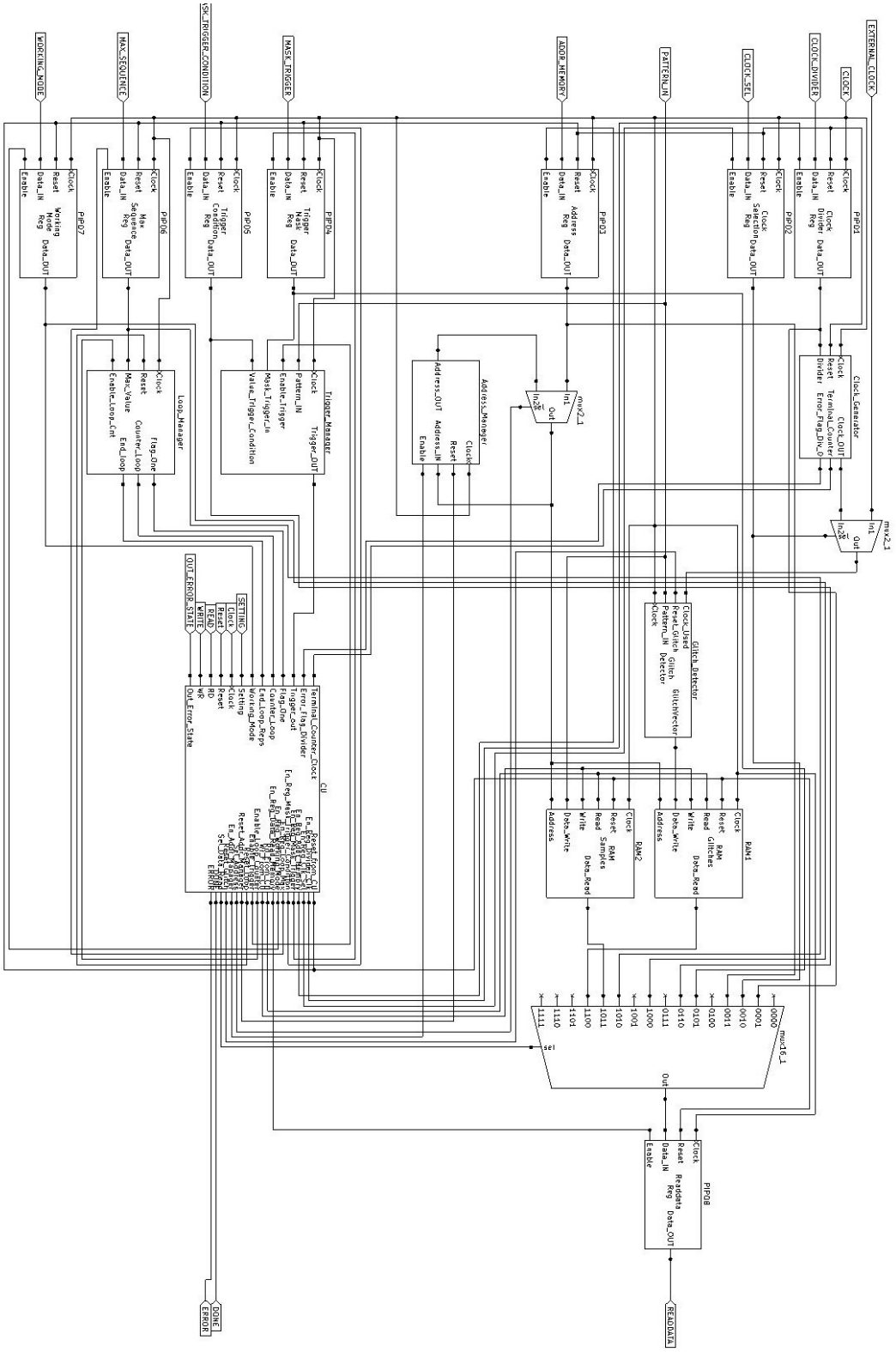


**Figura 4.2:** Datapath del "Logic Analyzer" con unità di controllo

### 4.2.2 Datapath del componente

Analizziamo nel dettaglio la struttura del datapath dell'analizzatore di stati logici (figura 4.2).

I blocchi funzionali che compongono il datapath sono fondamentalmente quelli presenti nel generatore di pattern, con l'aggiunta di alcuni nuovi moduli. Osserviamo infatti la presenza di:

- Registri PIPO (Parallel-In-Parallel-Out) a 32 bit
- Clock Generator
- Address Manager
- Trigger Manager
- Loop Manager
- Glitch Detector
- Multiplexer
- Memorie RAM

La tipologia di moduli utilizzati in questo progetto sono pressoché identici a quelli utilizzati per il generatore di pattern. La differenza più sostanziale sta nella riduzione del numero di registri PIPO presenti e soprattutto della presenza di due differenti memorie RAM, una per i campioni del pattern, ed una per gli eventuali glitch.

Una volta mostrate le differenze e le similitudini con il generatore di pattern, è necessario avere chiaro il significato dei segnali di ingresso e di uscita dell'architettura.

| **Segnale** | **Bit** | **Provenienza** | **Descrizione** |
|---|---|---|---|
| *clock* | 1 | Esterno | Segnale di temporizzazione del sistema. |
| *external_clock* | 1 | Esterno | Segnale di temporizzazione inviato dall'esterno ed utilizzabile al posto del *clock* interno. |
| *clock_divider* | 32 | Esterno | Segnale che determina il fattore di divisione del clock. |
| *selector_clock* | 32 | Esterno | Segnale che determina l'utilizzo del *clock* o dell'*external_clock*. |
| *addr_memory* | 32 | Esterno | Segnale indicante la locazione di memoria in cui effettuare un'operazione di lettura o scrittura. |
| *Pattern_IN* | 32 | Esterno | Segnale di dato indicante il valore da campionare ed analizzare. |

| | | | |
|---|---|---|---|
| *trigger_mask* | 32 | Esterno | Maschera usata per selezionare specifici bit di pattern_in. |
| *value_trigger _condition* | 32 | Esterno | Valore di riferimento per il confronto con pattern_in mascherato. |
| *working_mode* | 32 | Esterno | Valore che indica la modalità di funzionamento del generatore. Se l'ultimo bit è 0 allora il funzionamento è *One_Shot*, se 1 il funzionamento è *Sequenziale*. |
| *loop_max_value* | 32 | Esterno | Segnale indicante il numero di ripetizioni in modalita sequenziale, senza l'aggiornamento di *address* in modo manuale. |
| *reset_from_CU* | 1 | CU | Segnale di ripristino generale. |
| *reset_addr _manager* | 1 | CU | Segnale di ripristino di *addr_manager*. |
| *reset_loop* | 1 | CU | Segnale di ripristino di *loop_manager*. |
| *enable_loop _counter* | 1 | CU | Segnale di abilitazione del contatore presente nel *loop_manager*. |
| *enable_reg _loop_max* | 1 | CU | Segnale di abilitazione del registro *max_sequence_reg*. |
| *sel_data_read* | 4 | CU | Segnale di selezione del multiplexer, la cui uscita è collegata al registro *readdata_register* |
| *enable_reg _divider_clock* | 1 | CU | Segnale di abilitazione del registro *clock_divider_reg*. |
| *enable_reg _clock_sel* | 1 | CU | Segnale di abilitazione del registro *clock_selection_reg*. |
| *enable_reg _addr_memory* | 1 | CU | Segnale di abilitazione del registro *addr_register*. |
| *enable_reg _mask_trigger* | 1 | CU | Segnale di abilitazione del registro *trigger_mask_reg*. |
| *enable_reg_mask _trigger_condition* | 1 | CU | Segnale di abilitazione del registro *trigger_mask_condition_reg*. |
| *enable_reg _working_mode* | 1 | CU | Segnale di abilitazione del registro *working_mode_reg*. |
| *enable_trigger* | 1 | CU | Segnale di abilitazione di *trigger_manager*. |
| *enable_addr _manager* | 1 | CU | Segnale di abilitazione del sommatore presente in *addr_manager*. |

| | | | |
|---|---|---|---|
| *enable_reg_data _read_memory* | 1 | CU | Segnale di abilitazione del registro *readdata_reg*. |
| *rd_from_CU* | 1 | CU | Segnale di abilitazione della lettura in memoria. |
| *wr_from_CU* | 1 | CU | Segnale di abilitazione della scrittura in memoria. |
| *sel_address* | 1 | CU | Segnale di selezione del multiplexer in uscita da *addr_register*. |
| *reset_glitch* | 1 | CU | Segnale di ripristino del *clock_generator*. |

**Tabella 4.1:** Significato dei segnali di input presenti nel datapath del "Logic Analyzer"

| **Segnale** | **Bit** | **Direzione** | **Descrizione** |
|---|---|---|---|
| *readdata* | 32 | Esterno | Segnale indicante il dato letto in uno dei registri o in una delle due memorie. |
| *terminal_ _counter_clock* | 1 | CU | Segnale che indica il raggiungimento del conteggio impostato nel *clock_generator*. |
| *error_flag_ _divider* | 1 | CU | Segnale di errore che si attiva quando $DIVIDER = 0$. |
| *trigger_out* | 1 | CU | segnale di trigger che si attiva se la condizione nel *trigger_manager* è soddisfatta. |
| *flag_one* | 1 | CU | Segnale che si attiva se l'uscita del contatore del loop manager è 1 |
| *counter_loop* | 1 | CU | Segnale che si attiva se l'uscita del contatore del loop manager è diverso da 0 |
| *working_mode* | 1 | CU | Segnale che indica la modalità di funzionamento del sistema |
| *end_loop_reps* | 32 | CU | Segnale che indica la fine del loop |

**Tabella 4.2:** Significato dei segnali di output presenti nel datapath del "Logic Analyzer"

Analizziamo a questo punto l'architettura ed il funzionamento nel dettaglio. Così come per il generatore di pattern, i primi blocchi che vengono utilizzati sono i registri di dato, stato e controllo. Quelli utilizzati nell'architettura sono i seguenti:

- Clock Divider Register

- Clock Selection Register
- Addr Register
- Trigger Mask Register
- Trigger Condition Register
- Max Sequence Register
- Working Mode Register
- Readdata Register

Si può notare che i registri utilizzati nell'analizzatore sono gli stessi utilizzati nel generatore. La differenza maggiore sta infatti nell'omissione di alcuni registri utilizzati in precedenza, e non nel loro significato.

Un'altra sostanziale differenza sta nel fatto che in questo progetto, l'accesso alla memoria dei campioni è diretto, cioè non si passa per un registro intermedio. Questo perché il segnale da analizzare viene campionato e scritto direttamente in memoria. Ciò non succede per la memoria dei glitch, in quanto il segnale viene prima analizzato e, una volta terminata la ricerca dei glitch, avviene la scrittura di questi nella memoria dedicata. Ovviamente dovrà essere possibile leggere i segnali campionati e i relativi glitch. Per questo motivo è stato associato alle due memorie un indirizzo di riferimento. Per la memoria dei campioni, l'indirizzo è 0x0000000B, mentre per la memoria dei glitch l'indirizzo è 0x0000000C.

Una volta settati i vari registri di dato e controllo, entrano in funzione i vari moduli. Il loro funzionamento è lo stesso descritto nel capitolo precedente, e pertanto non verranno esposti. Analizziamo più nel dettaglio l'unico modulo introdotto nell'analizzatore di stati logici, ovvero il Glitch Detector.

#### 4.2.2.1 Glitch Detector

Il "Glitch Detector" è il modulo incaricato di individuare la presenza di glitch durante l'acquisizione del segnale, ovvero transizioni spurie e non volute. Il progetto del "Glitch Detector" è basato su un datapath, il cui compito è quello di individuare fisicamente una transizione, e un'unità di controllo, che decodifica se la transizione è un glitch o una transizione voluta del segnale. Queste due istanze sono collegate insieme e formano il "Glitch Detector" relativo ad un singolo bit del segnale da analizzare. Poiché il segnale in ingresso all'analizzatore di stati logici è programmato per essere di 32 bit, è stata creata una ulteriore struttura ad alto livello, il cui compito è quello di istanziare 32 volte il glitch detector per singolo bit.

Vediamo nel dettaglio l'architettura del "Glitch Detector" per singolo bit. Il datapath riceve come segnali di ingresso il *clock* ed il bit da analizzare, mentre fornisce in uscita un segnale chiamato *edge*. Ogni volta che il *clock* ha un fronte di salita, il segnale $pattern\_IN$ viene campionato. Il campione ottenuto viene memorizzato nella prima posizione di uno shift register a 2 bit, il cui valore precedente viene

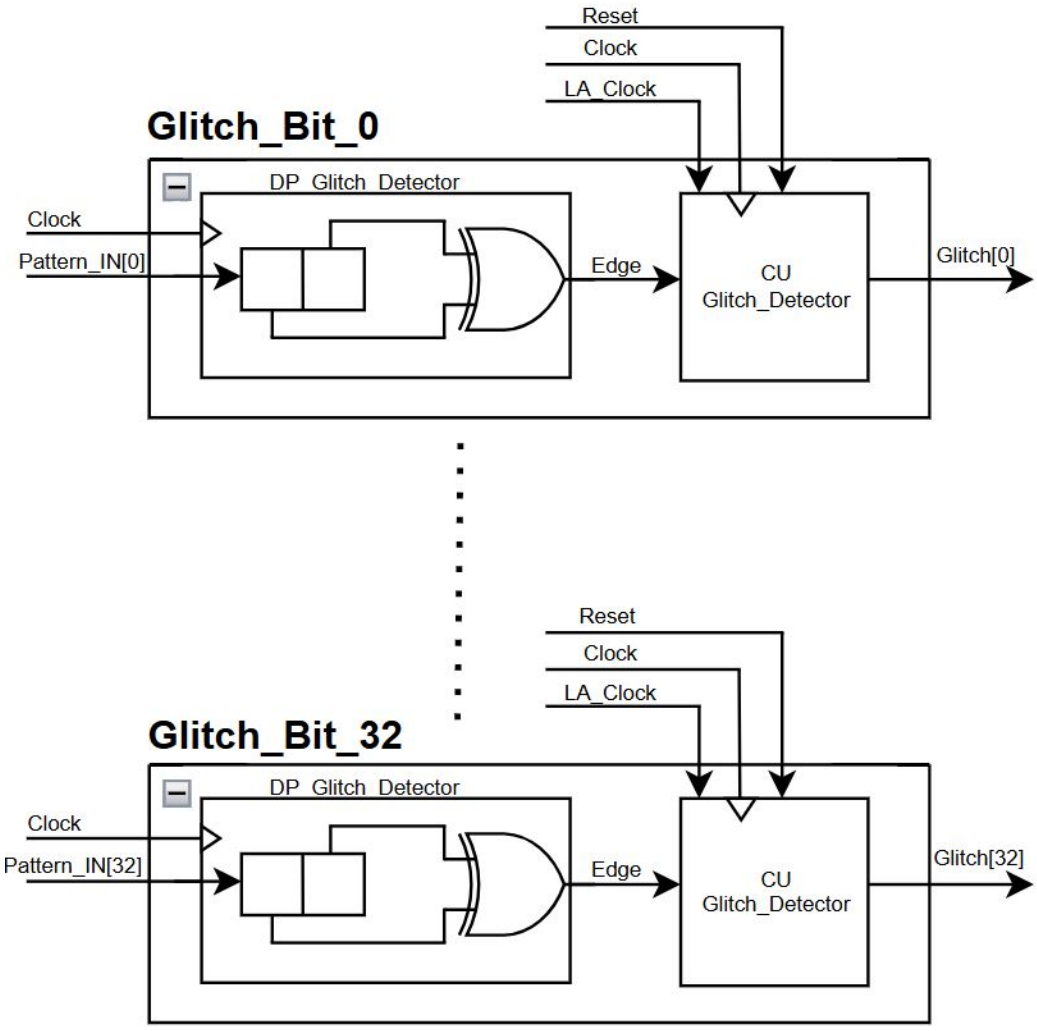


**Figura 4.3:** Struttura del "Glitch Detector"

spostato nella posizione successiva. Questo comporta che nel registro si avranno memorizzati i campioni dello stesso segnale ma in due istanti consecutivi. Questi due bit diventeranno gli ingressi di una porta XOR il cui compito è quello di verificare se questi sono uguali o meno. Nel caso in cui i due bit fossero uguali, l'uscita *edge* rimarrebbe a zero perché non è stata individuate nessuna transizione del segnale campionato. Se invece i due bit sono differenti, allora la transizione viene individuata ed il segnale di *edge* passerà ad un valore logico alto. Questo flag diventerà un ingresso dell'unità di controllo, insieme al segnale di *clock*, *reset* ed al segnale segnale *LA_Clock*. Quest'ultimo segnale è il clock generato dal "Clock Generator" ed è utilizzato per indicare per quanto tempo vengono analizzati i bit del pattern. Infatti, la temporizzazione dell'analizzatore di stati logici è dato proprio da questo segnale. La macchina a stati implementata è formata da 5 stati.

Il primo stato è *s0_idle* che rappresenta lo stato di attesa del segnale *LA_Clock*. Quando il segnale di temporizzazione ha una transazione verso l'alto, allora si passa allo stato $s1_zero$.

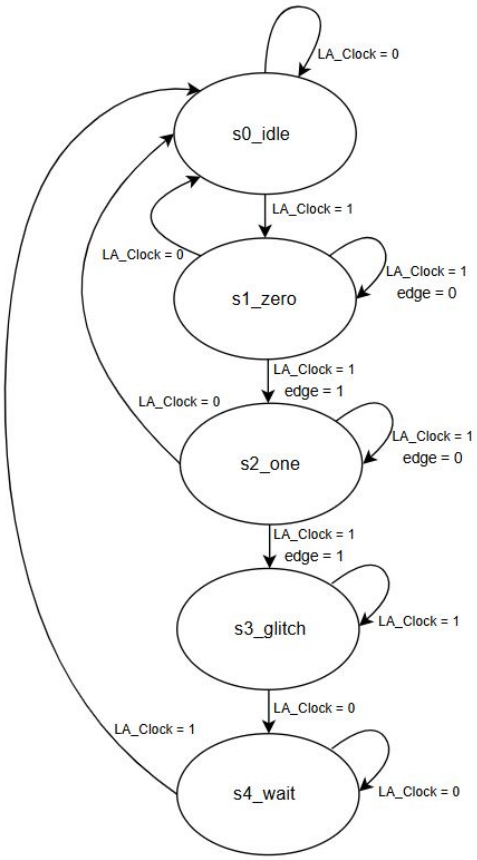


**Figura 4.4:** Struttura dell'unità di controllo del "Glitch Detector"

In questo stato, l'analisi dei glitch è iniziata ma ancora non è stata individuata nessuna transizione del bit monitorato. Nel momento in cui il segnale di *edge* proveniente dal datapath passa ad 1, vuol dire che è stata individuata una transizione e pertanto, se *LA_Clock* è ancora attivo, si passa allo stato successivo, ovvero *s2_One*.

Aver campionato una transizione, non implica necessariamente la presenza di un glitch, ma magari solamente un cambiamento dello stato del segnale. Tuttavia, se una volta dentro questo stato, il segnale *edge* rimane attivo anche nel colpo di clock successivo, vuol dire che è stata individuata una transizione subito successiva alla prima. In questo caso, si è in presenza di un glitch e si passa allo stato successivo, ovvero *s3_glitch*.

Una volta entrati in questo stato, il segnale di uscita *glitch* si attiva, segnalando quindi la presenza del glitch durante il campionamento. A questo punto ulteriori analisi sul bit in questione sarebbero inutili pertanto si resta in questo stato fin quando il segnale *LA_Clock* non diventa 0. In questo caso si passo allo stato *s4_wait*, in cui si attende un nuovo fronte attivo di *LA_Clock*.

E' da notare come le transizione da uno stato all'altro avvengono in funzione del segnale di *clock*, mentre l'attivazione dell'analisi dipende dal segnale *LA_Clock*, che svolge il ruolo di segnale di abilitazione.

L'unione del datapath e dell'unità di controllo appena descritti, forma il "Glitch Detector" di un singolo bit. Tuttavia, essendo il segnale $Pattern\_IN$ composto da 32 bit, è stato necessario creare un nuovo progetto con l'unico scopo di implementare 32 volte questo componente. In ognuna istanza viene inviato come segnale di ingresso, un differente bit di $Pattern\_IN$. I 32 segnali di glitch, pertanto formeranno un vettore che verrà salvato nella memoria $RAM\_GLITCH$. Se questo vettore è 0x00000000, allora durante l'analisi dei bit non sono emersi glitch. Invece se il valore è differente, indica la presenza di un glitch. Per individuare quale dei 32 bit presentava il glitch individuato, è sufficiente controllare la posizione dell'1 nel vettore di glitch.

Per esempio se il valore salvato in $RAM\_Glitch$ fosse 0x00000010, vorrebbe dire che è stato individuato un unico glitch in $Pattern\_IN[4]$.

## 4.3 Unità di controllo del "Logic Analyzer"

Compresa l'architettura del sistema, il compito ed il funzionamento dei singoli blocchi, si analizzi l'algoritmo alla base dell'analizzatore di stati logici. Anche in questo caso il "pallogramma" è stato suddiviso in più sezioni per facilitarne la rappresentazione. La prima parte è quella rappresentata in figura 4.5.

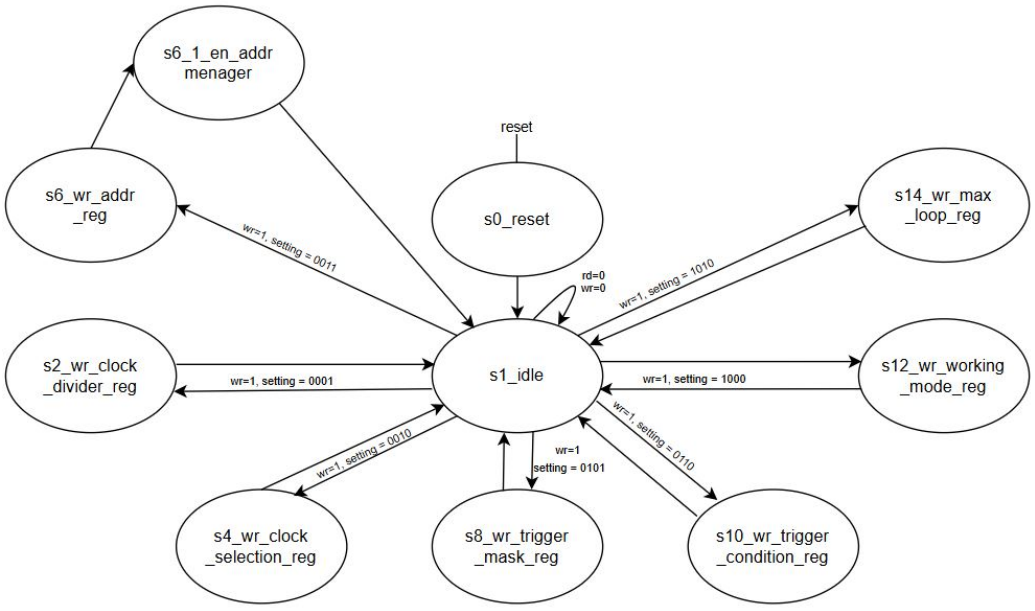


**Figura 4.5:** Sezione dell'unità di controllo del "Logic Analyzer" relativa alla scrittura dei registri

Questa sezione dell'unità di controllo evidenzia la scrittura dei registri di stato e controllo del progetto. Come in precedenza, i dati necessari per l'operazione di scrittura vengono inviati alla CU dal blocco "Bridge SPI to Avalon". Questo invierà un segnale di *read* o *write* ed un indirizzo che nell'unità di controllo viene indicato come *setting*. Una volta avviato il sistema ed essere usciti dallo stato di reset ($s0\_reset$), lo stato corrente diventa $s1\_idle$.

Da qui, in funzione del segnale di *read* o *write* e del *setting*, si passerà ad uno degli stati mostrati in figura 4.5. Ognuno di questi stati rappresenta la scrittura di un registro. Infatti, quando lo stato corrente della CU passa in uno di questi, si ottiene l'abilitazione del registro associato a quello stato, con la conseguente scrittura del dato in ingresso. In particolare, gli stati indirizzi disponibili per questi registri sono:

| **Setting** | **Operazione** | **Stato di destinazione** | **Registro** |
|---|---|---|---|
| 0000 | Write | *s*1_*idle* | Nessuno |
| 0001 | Write | *s*2_*wr_clock_* _*divider_reg* | clock _divider_reg |
| 0010 | Write | *s*4_*wr_clock_* _*selection_reg* | clock_selection_reg |
| 0011 | Write | *s*6_*wr_addr_reg* | addr_reg |
| 0101 | Write | *s*8_*wr_trigger_* _*mask_reg* | trigger_mask_reg |
| 0110 | Write | *s*10_*wr_trigger_* _*condition_reg* | trigger_condition_reg |
| 1000 | Write | *s*12_*wr_working_* _*mode_reg* | working_mode_reg |
| 1010 | Write | *s*14_*wr_max_loop_reg* | max_loop_reg |
| 1011 | Write | *s*18_*divider_check* | ram_sample ram_glitch |

**Tabella 4.3:** Setting associato ai vari stati e registri per operazioni di scrittura

Una volta terminata la scrittura nei registri, si ritorna in automatico allo stato di idle in attesa di un nuovo comando. L'unica differenza sta nello stato *s*6_*wr_addr_reg*. Una volta scritto l'indirizzo nel registro *addr_reg*, si passa nello stato *s*6_1_*enable_addr_manager* che abilita il gestore degli indirizzi.

Osserviamo adesso come avviene la lettura dei registri di stato e controllo che abbiamo scritto in precedenza (figura 4.6. Il comportamento descritto dalla macchina a stati è molto semplice, in quanto, una volta ricevuto il segnale di *read*, dallo stato di $s1_idle$ ci si sposta in uno degli stati relativi alla lettura. Gli indirizzi sono ovviamente gli stessi di quelli esposti nella sezione relativa alla scrittura. Come descritto anche per il generatore di pattern, durante la lettura non si accederà ai registri, bensì al segnale di selezione del multiplexer a 16 ingressi. Infatti le uscite dei vari registri sono già in ingresso al multiplexer. In funzione del segnale di *setting* si imposta il selettore del multiplexer che renderà il segnale scelto disponibile al registro *Read_data_register*. Grazie al segnale di *read*, verrà attivato il segnale di abilitazione di questo registro che invierà successivamente il dato letto al "Bridge SPI to Avalon" per l'invio sul *miso*. Terminata l'operazione di lettura, si passerà allo stato *s*30_*done* che attiverà un flag che indicherà la conclusione dell'operazione e successivamente in *s*1_*idle*.

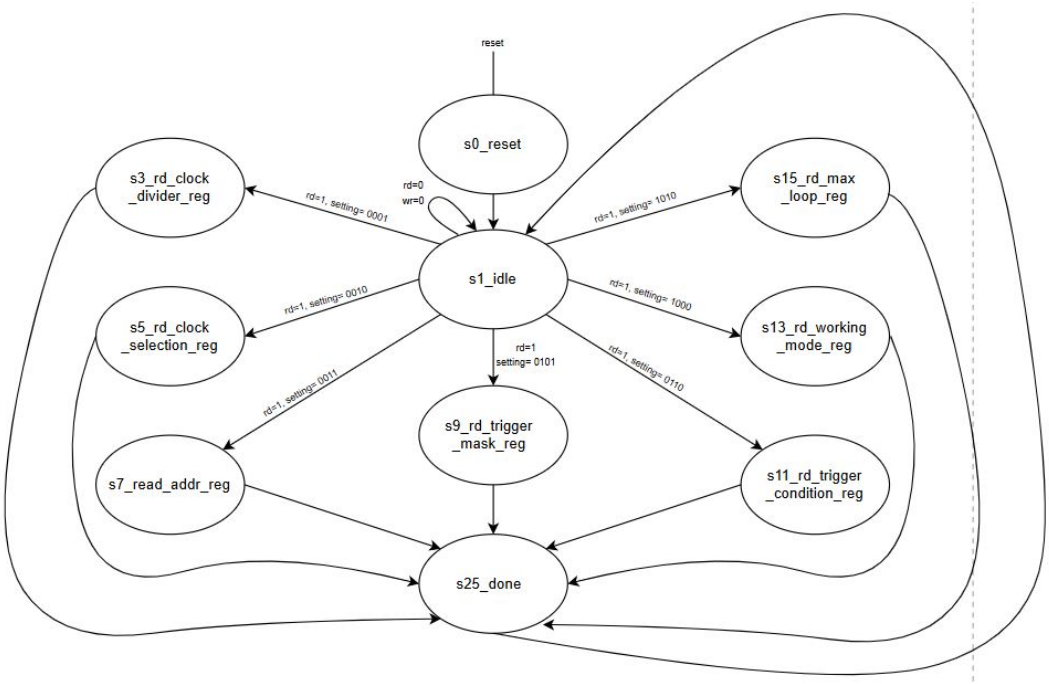


**Figura 4.6:** Sezione dell'unità di controllo del "Logic Analyzer" relativa alla lettura dei registri

| **Setting** | **Operazione** | **Stato di destinazione** | **Registro** |
|---|---|---|---|
| 0000 | Read | *s1_idle* | Nessuno |
| 0001 | Read | *s3_rd_clock_divider_reg* | clock _divider_reg |
| 0010 | Read | *s5_rd_clock_selection_reg* | clock_selection_reg |
| 0011 | Read | *s7_rd_addr_reg* | addr_reg |
| 0101 | Read | *s9_rd_trigger_mask_reg* | trigger_mask_reg |
| 0110 | Read | *s11_rd_trigger_ _condition_reg* | trigger_condition_reg |
| 1000 | Read | *s13_rd_working_ _mode_reg* | working_mode_reg |
| 1010 | Read | *s15_rd_max_loop_reg* | max_loop_reg |
| 1011 | Read | *s16_rd_sample* *s16_2_loop_read_first* *s16_5_update_address* | ram_sample |
| 1100 | Read | *s17_rd_glitch* *s17_2_loop_read_first* *s17_5_update_address* | ram_glitch |

**Tabella 4.4:** Setting associato ai vari stati e registri per operazioni di lettura

A questo punto osserviamo cosa accade quando arriva in ingresso il segnale da analizzare (figura 4.7). Una volta campionato il segnale $write = 1$ ed il $setting = 1011$ relativo alla memoria dei campioni, si entra nello stato *s18_divider_check* in cui si controlla il valore del $DIVIDER$ scritto nel $CLOCK\_GENERATOR$. Come

per il generatore di pattern, se il valore campionato è uno 0, viene campionato il flag $ERROR\_FLAG\_DIVIDER = 1$ e si entra nello stato di errore, terminando la l'acquisizione dei campioni. Se invece il $DIVIDER$ è diverso da 0, allora il flag sarà disattivato e si entrerà nello stato $s19\_loop\_active$ in cui si abilita il $LOOP\_MANAGER$.

Anche in questo caso si effettua un controllo, in particolare sul flag $END\_LOOP\_REPS$. Se è uguale ad 1, allora si andrà nello stato di errore poiché è già stato raggiunto il numero massimo di ripetizioni in sequenza, se invece il flag è 0 si prosegue nello stato $s20\_trigger\_active$, in cui viene dato il segnale di abilitazione al $TRIGGER\_MANAGER$.

In questo colpo di clock il trigger manager campiona l'enable e comincia ed effettuare le verifiche necessarie tra pattern da analizzare, maschera e condizione. Al colpo di clock successivo si andrà nello stato $s21\_trigger\_value$ in cui verrà campionato il segnale di $TRIGGER\_OUT$. Se $trigger\_out = 0$ allora la condizione non è stata rispettata e si andrà nello stato di errore, se invece è uguale ad 1 si andrà nello stato $s29\_wait\_clock\_sample$, ovvero lo stato di attesa in cui avviene la sincronizzazione con il clock generato mediante il clock generator.

Campionato il fronte del clock, si passerà allo stato $s22\_glitch\_manager$ in cui si andranno ad individuare eventuali glitch durante la ricezione del segale.

A questo punto, se la modalità di funzionamento è one-shot, lo stato successivo sarà $s24\_sampling\_writing$ in cui si andranno a scrivere sia i campioni acquisiti, sia i glitch individuati in memoria. Una volta terminata la scrittura, verranno resettati i vari moduli e si andrà nello stato $s25\_done$ terminando l'analisi.

Se invece la modalità selezionata è sequenziale è necessario effettuare alcuni controlli ed eventualmente aggiornare l'indirizzo. Se al termine dell'analisi dei glitch il segnale $flag\_one = 1$ e $working\_mode = 1$, allora anche il processo sequenziale condurrà allo stato $s24\_sampling\_writing$. Dopo la scrittura in memoria, si passerà allo stato $s27\_1\_addr\_loop\_first$ che aggiornerà il segnale $addr\_sel$ per le analisi successive. Dopo di che si andrà nello stato di idle.

Se invece $flag\_one = 0$ e $working\_mode = 1$, è necessario aggiornare l'indirizzo di memoria su cui scrivere i campioni. Si andrà quindi negli stati $s27\_addr\_loop\_upgrade$ che attiva il segnale di abilitazione dell'$ADDR\_MANAGER$, aggiorna il segnale $addr\_sel$ ed abilita la scrittura in memoria dei campioni e dei glitch. Terminata l'operazione di scrittura, si andrà nello stato di idle in attesa di un nuovo segnale da analizzare. Ciò accade solamente nel caso in cui $end\_loop\_reps = 0$, altrimenti si andrà nello stato di reset dei moduli e nello stato di done, per terminale l'analisi sequenziale dei segnali.

In modalità sequenziale per ogni dato campionato in ingresso avremo il salvataggio dei campioni in locazioni di memoria consecutive.

Osserviamo infine cosa accade nel momento in cui si vogliono effettuare delle letture dei campioni o dei glitch in modalità sequenziale (figura 4.8).

Se dallo stato di idle viene campionato il segnale di $rd = 1$ e $setting = 1011$, si procede con la lettura dei campioni salvati in memoria $ram\_sample$. In funzione

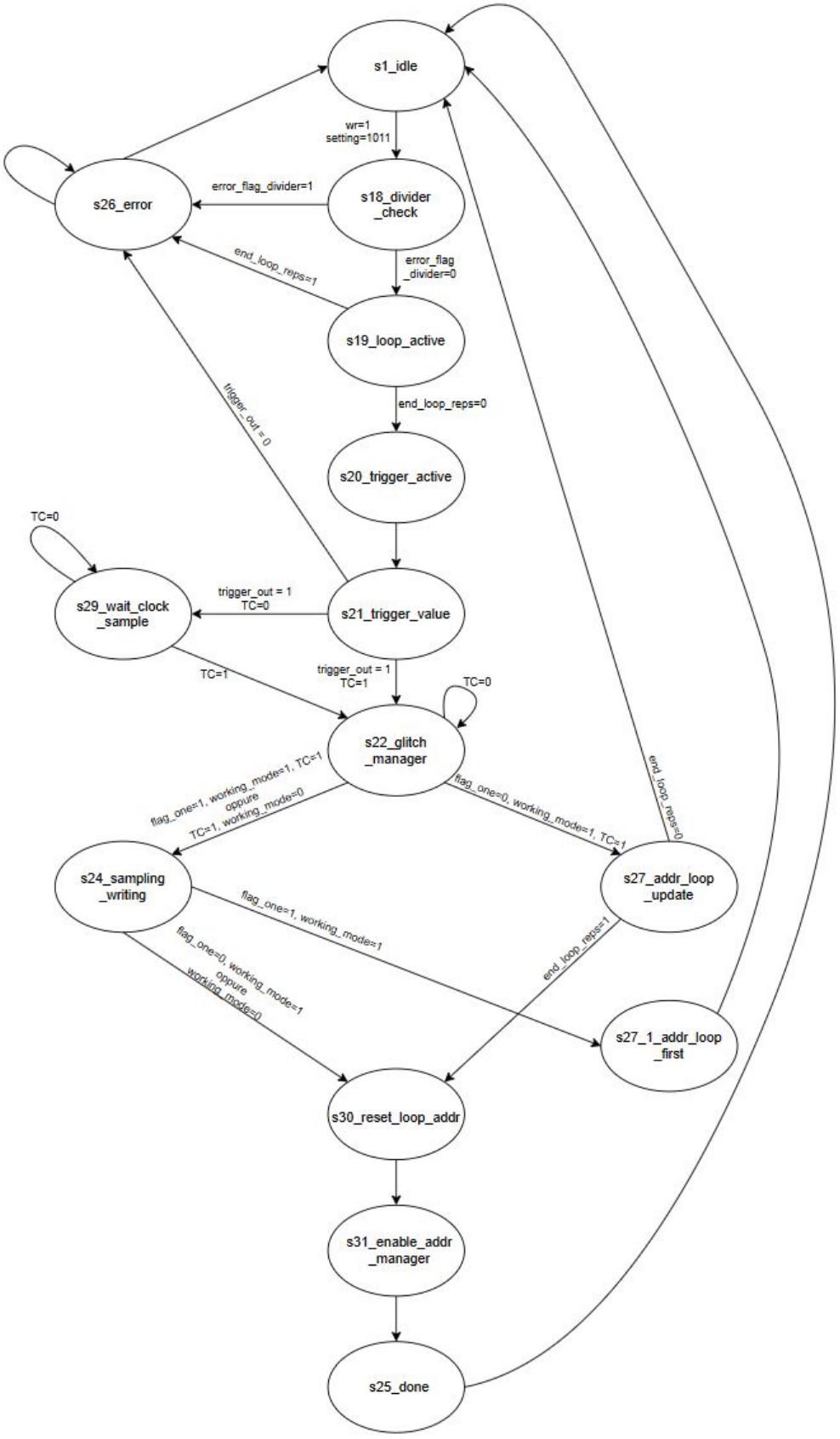


**Figura 4.7:** Sezione dell'unità di controllo del "Logic Analyzer" relativa al segnale da analizzare con conseguente scrittura dei campioni e glitch

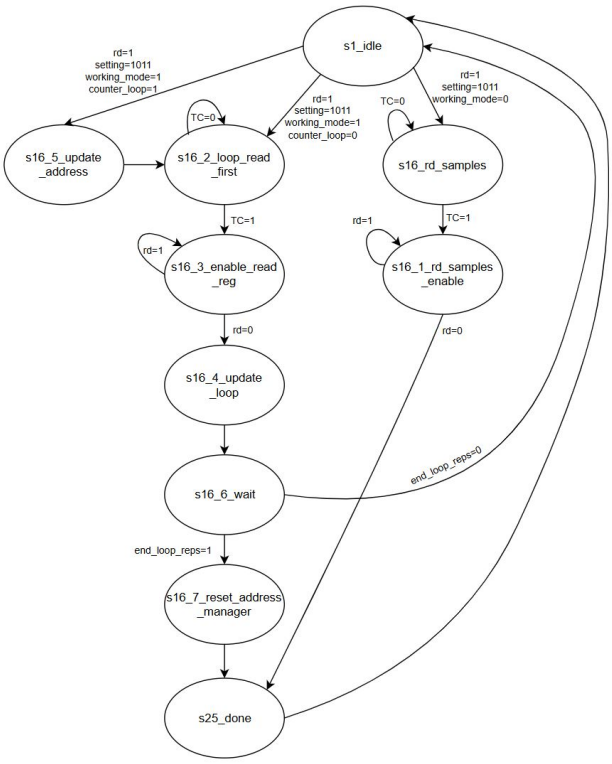


**Figura 4.8:** Sezione dell'unità di controllo del "Logic Analyzer" relativa alla lettura dei campioni in memoria

della modalità di funzionamento, si va in diversi stati. Se la modalità di lavoro scelto è one-shot, allora si proseguirà nello stato $s16\_rd\_sample$ in cui si invia il segnale di read alla memoria e si setta il segnale di selezione del multiplexer, e successivamente allo stato $s16\_1\_rd\_sample\_enable$ in cui si abilita il registro dedicato alla lettura. Dopo di che si andrà nello stato $s25\_done$ e ed infine nuovamente in $s1\_idle$.

Se invece la modalità di funzionamento scelta è quella sequenziale, si controllerà un altro segnale per scegliere lo stato futuro corretto. Se $counter\_loop = 0$, si andrà nello stato $s16\_2\_loop\_read\_first$, in cui si invierà il segnale di read alla memoria. Rilevato $TC = 1$, si andrà nello stato $s16\_3\_enable\_read\_reg$ in cui verrà abilitato $readdata\_reg$. Una volta conclusa la lettura, si andrà nello stato $s16\_4\_updtae\_loop$ in cui si abiliterà il loop manager, aggiornando il conteggio delle ripetizioni.Questo aggiornamento comporta che $counter\_loop$ passerà da 0 ad 1. Dopo di che si passerà in uno stato di attesa dal quale si prospettano due strade future. Se $end\_loop\_reps = 0$, allora si andrà in $s1_idle$ in attesa di una nuova istruzione. A questo punto, se si volesse effettuare una nuova lettura dei campioni, dallo stato $s1\_idle$ si andrà in $s16\_5\_update\_address$ e non $s16\_2\_loop\_read\_first$. Ciò perché dopo la prima lettura, $counter\_loop = 1$. In questo nuovo stato si seleziona il segnale di selezione del multiplexer degli indirizzi. In questo modo verrà utilizzato l'indirizzo aggiornato dall'address_manager e non l'indirizzo scritto nel registro addr_reg. Dopo di che si andrà nello stato $s16\_2_{l}oop\_read_{f}irst$, riprendendo il flusso spiegato precedentemente. Dallo stato $s16\_6\_wait$, non appena si rileva

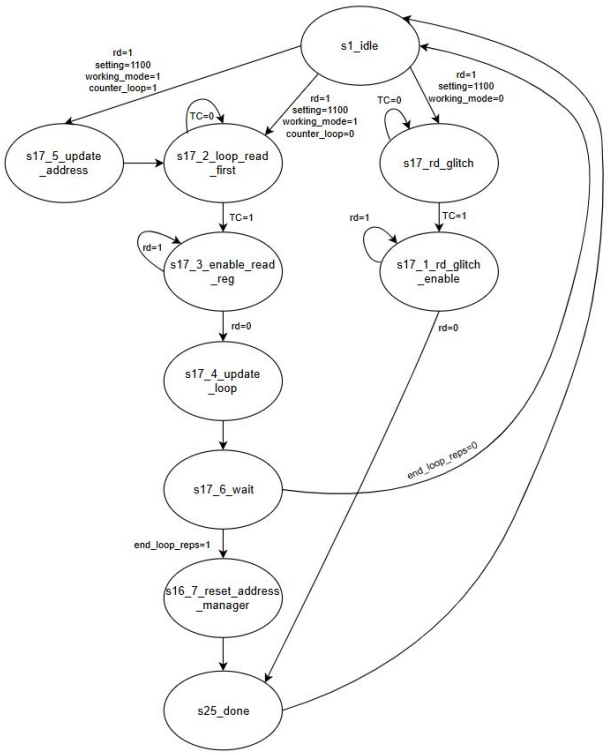


**Figura 4.9:** Sezione dell'unità di controllo del "Logic Analyzer" relativa alla lettura dei glitch in memoria

$end\_loop\_reps = 1$, si passerà allo stato $s16\_7\_reset\_address\_manager$ in cui si resetta l'address_manager e il selettore degli indirizzi torna a 0. Dopo di che, viene conclusa l'ultima operazione di lettura passando prima nello stato $s25\_done$ e successivamente in $s1\_idle$.

Nel caso in cui si volessero leggere gli eventuali glitch piuttosto che i campioni, si seguirà una struttura identica a quella appena esposta, con l'unica differenza dovuta a $setting = 1100$. La struttura dell'unità di controllo relativa alla lettura dei glitch è mostrata in figura 4.9

## 4.4 Timing dell'analizzatore

Concluso il discorso relativo all'unità di controllo dell'analizzatore di stati logici, è possibile analizzarne il timing. Come abbiamo già detto in precedenza, le operazioni di lettura e scrittura dei registri di dato e controllo sono affidate al "Bridge SPI to Avalon", così come discusso nel capitolo relativo al generatore di pattern(figura 3.6). Per questo motivo, verranno omessi i timing relativi a queste operazioni e si analizzerà direttamente il comportamento dell'analizzatore nel momento in cui si ha in ingresso il segnale da analizzare. Quindi si farà riferimento al comportamento descritto dall'unità di controllo in figura 4.7.

Il timing diagram relativo all'acquisizione di un segnale e la relativa analisi è mostrato in figura 4.10.

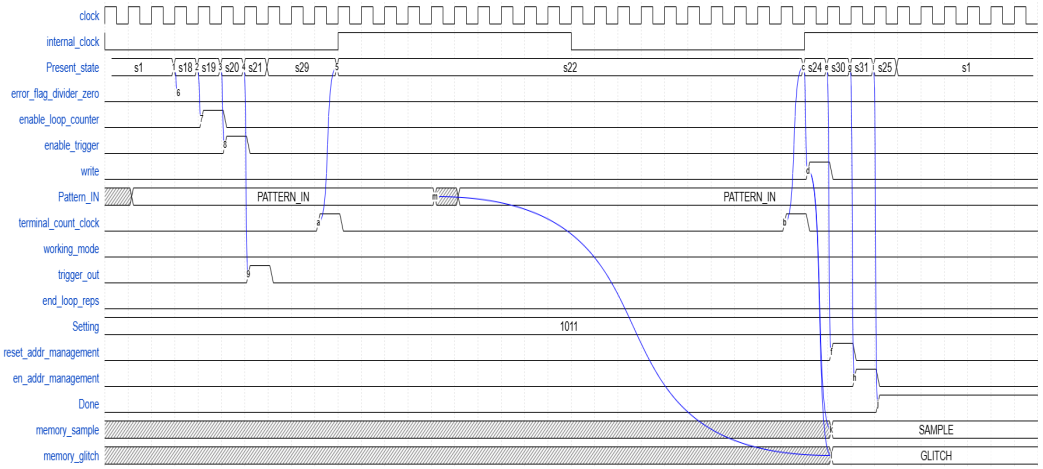


**Figura 4.10:** Timing diagram relativo all'analisi di un segnale in ingresso in modalità one-shot

Osserviamo il comportamento dell'analizzatore di stati logici una volta campionato il segnale di write del "Bridge SPI to Avalon" ed il *setting*=1011. Il segnale $Pattern\_IN$ è quindi disponibile all'analizzatore, il quale inizia l'analisi del dato e delle impostazioni.

Cosi come accadeva nel Pattern Generator, il primo controllo è quello relativo al divisore di clock. Se uguale a 0 si entra in uno stato di errore, altrimenti si procede con il l'abilitazione del loop manager. Anche in questo caso viene effettuato il controllo sul numero di ripetizioni e se il segnale $end_loop\_reps = 1$ si andrà nello stato di errore, altrimenti si procederà con l'attivazione del trigger manager.

Dopo l'attivazione si attende un colpo di clock affinché si facciano i dovuti controlli e si ottenga il valore di $trigger\_out$. Se il trigger non è stato raggiunto si andrà nello stato di errore, altrimenti si procederà con l'effettiva analisi del pattern.

Per prima cosa, si attende il primo fronte di salita del clock utilizzato, che in figura 4.10 è $internal\_clock$. Una volta campionato, si passa allo stato $s22\_glitch\_manager$ in cui avviene il campionamento del $pattern\_in$ alla frequenza del clock base. Questa analisi durerà per tutto il periodo dell'$internal\_clock$, il quale ricopre una sorta di ruolo da "enable". Se durante questo periodo si osserva una variazione dei bit di $pattern\_in$, si campionerebbero dei glitch. Terminato il periodo dell'$internal\_clock$, si procede con la scrittura in memoria dei campioni del pattern e dei glitch osservati. Infine verrà resettato l'address manager e il loop manager e si andrà nello stato di done.

Osserviamo adesso come si comporta il sistema se lavorasse in modalità sequenziale. Anche in questo caso omettiamo tutte le operazioni di scrittura necessarie per settare correttamente il sistema.

La parte preparatoria di ogni analisi è sempre uguale e corrisponde a quella analizzata per la modalità one-shot. Si osservano comportamenti leggermente diversi alla fine dell'analisi, ovvero dopo aver individuato eventuali glitch. Le possibilità sono 3 e sono mostrate in figura 4.11. Il primo caso corrisponde al primo dato ricevuto. Dopo aver analizzato eventuali glitch nello stato $s22\_glitch\_manager$, lo

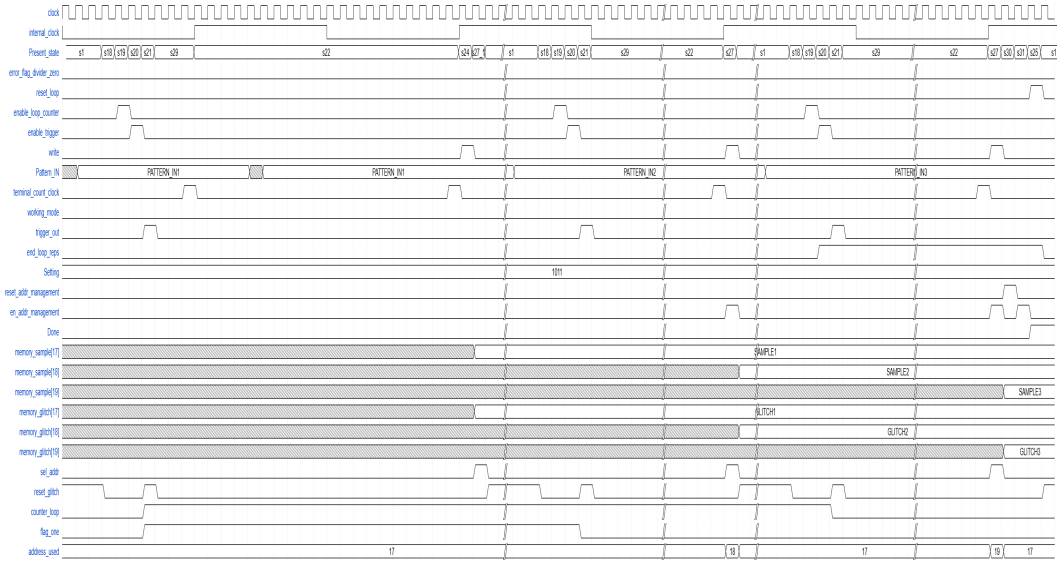


**Figura 4.11:** Timing diagram relativo all'analisi di tre segnali in modalità sequenziale

stato successivo sarà $s24\_sampling\_writing$, in cui avviene la scrittura dei dati nelle memorie, e successivamente $s27\_1\_addr_loop\_first$ in cui si aggiornerà il segnale $addr\_sel$ per le analisi successive. Dopo di che si andrà nello stato di idle in attesa del secondo dato.

Conclusa l'analisi dei glitch del secondo dato, lo stato successivo non sarà $s27\_1\_addr_loop\_first$, ma $s27\_addr\_loop\_upgrade$ nel quale viene attivato il segnale di abilitazione dell'$Address\_Manager$, viene aggiornato il segnale di selezione degli indirizzi $addr_sel$ e viene abilitata la scrittura in memoria dei campioni e dei glitch. Dopo di che si torna in idle in attesa del terzo ed, in questo caso specifico, ultimo dato.

Anche in questo caso la parte di controllo è la stessa, mentre avremo delle divergenze nella parte conclusiva dell'analisi. Dopo aver terminato l'acquisizione di eventuali glitch, l'unità di controllo si sposterà su $s27\_addr\_loop\_upgrade$. Dopo di che, verrà campionato il segnale $end\_loop\_reps = 1$, il quale indica il raggiungimento del numero massimo di ripetizioni. Quindi, vengono resettati i vari moduli e successivamente si entra nello stato $s25\_done$ per concludere l'operazione.

Osserviamo adesso come avviene la lettura one-shot dei campioni salvati in memoria e successivamente la lettura in modalità sequenziale. La lettura dei glitch verrà omessa in quanto il comportamento del sistema è lo stesso delle letture dei campioni. La lettura dei registri di controllo invece è la stessa commentata nel generatore di pattern e quindi verrà omessa anch'essa.

La lettura in modalità one shot è molto semplice e viene mostrata in figura 4.12.

Se nello stato di idle viene campionato il segnale di $rd = 1$ e $setting = 1011$, si procede con la lettura dei campioni salvati in memoria. Con la modalità di lavoro one-shot, si proseguirà nello stato $s16\_rd\_sample$ in cui verrà inviato il segnale di lettura alla memoria e verrà impostato il segnale di selezione del multiplexer. In questo modo, i campioni presenti in memoria verranno letti e passeranno attraverso

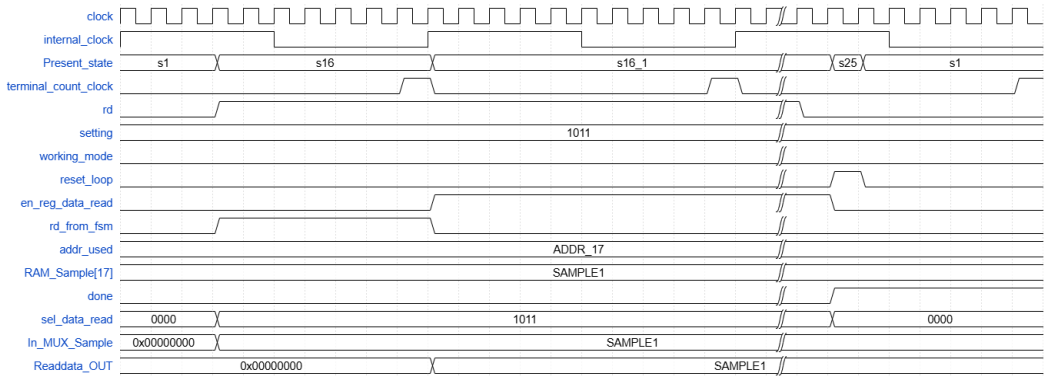


**Figura 4.12:** Timing diagram relativo alla lettura dei campioni presenti in memoria in modalità one shot

il multiplexer delle letture. Successivamente, nello stato $s16\_1\_rd\_sample\_enable$, verrà abilitato il segnale di enable del registro dedicato alla lettura ed i campioni saranno disponibili in uscita. Dopo di che si andrà nello stato $s25\_done$ ed infine nuovamente in $s1\_idle$, in attesa di una nuova istruzione.

La lettura in modalità sequenziale è invece leggermente più complicata, pur presentando come core lo stesso algoritmo della modalità one-shot. In modalità sequenziale, la lettura si divide in tre strade differenti. La prima riguarda la lettura del primo vettore di campioni, quindi quello salvato nell'indirizzo presente nel registro $addr\_reg$. La seconda strada viene percorsa per tutte le letture successive alla prima, in quanto è necessario aggiornare l'indirizzo in cui effettuare la lettura. Infine, la terza strada la si percorre durante l'ultima lettura, in quanto è necessario resettare i vari moduli interessati all'istruzione.

In figura 4.13 viene mostrato il timing diagram relativo alla lettura del primo e del secondo vettore di campioni.

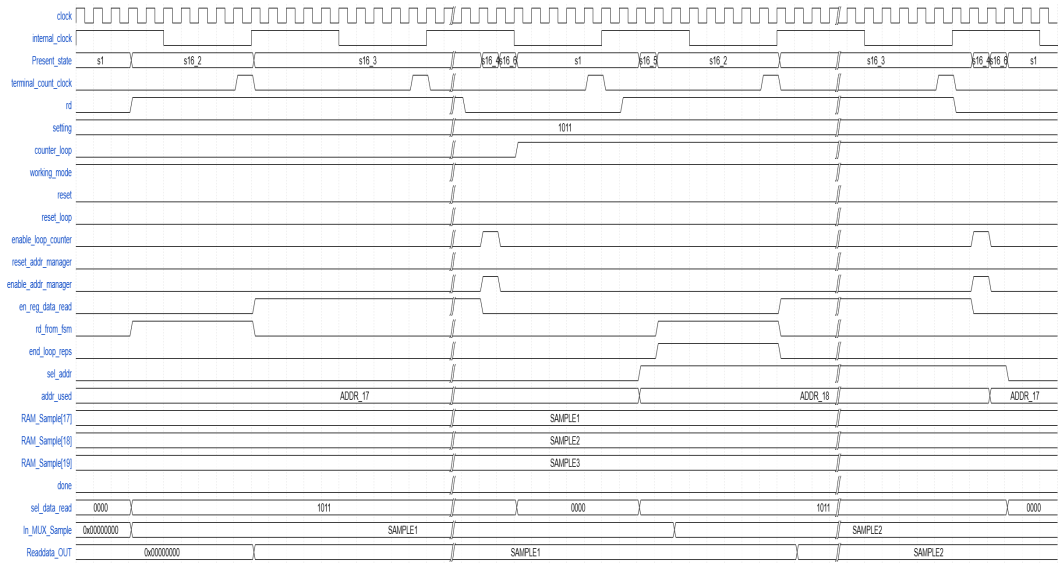


**Figura 4.13:** Timing diagram relativo alla lettura dei primi due vettori di campioni presenti in memoria in modalità sequenziale

Quando dallo stato di idle vengono campionati $rd = 1$, $setting = 1011$ e $working\_mode = 1$, ha avvio la lettura in modalità sequenziale. Il segnale che però determina l'ingresso nello stato $s16\_2\_loop\_read\_first$ è il segnale $counter\_loop = 0$. Questo segnale assume il valore 0 quando il contatore del $loop\_manager$ è a 0, mentre assume il valore 1 quando il contatore assume un valore diverso da 0. Nello stato $s16\_2\_loop\_read\_first$, si invierà il segnale di read alla memoria. Rilevato $TC = 1$, si andrà nello stato $s16\_3\_enable\_read\_reg$ in cui verrà abilitato il registro relativo alla lettura. A questo punto il vettore di campioni è già disponibile in uscita. Una volta conclusa la lettura, si andrà nello stato $s16\_4\_update\_loop$ in cui verrà abilitato il loop manager, aggiornando il conteggio delle ripetizioni. Questo aggiornamento comporta il passaggio da 0 ad 1 di $counter\_loop$. Essendo $end\_loop\_reps = 0$, allora si tornerà in idle in attesa di una nuova istruzione.

Con la seconda lettura, si passerà dallo stato $s1\_idle$ allo stato $s16\_5\_update\_address$ a causa di $counter\_loop = 1$, nel quale verrà impostato ad 1 il segnale di selezione del multiplexer degli indirizzi. In questo modo verrà utilizzato l'indirizzo aggiornato dall'address_manager e non l'indirizzo scritto nel registro addr_reg. Dopo di che si andrà nuovamente nello stato $s16\_2\_loop\_read\_first$, riprendendo il flusso spiegato precedentemente.

Il percorso rimarrà sempre questo fin quando non si arriverà all'ultima lettura possibile dal loop_manager. Infatti, dallo stato $s16\_6\_wait$, non appena viene rilevato il segnale $end_loop_reps = 1$, si passerà allo stato $s16\_7\_reset\_address\_manager$ in cui verrà resettato l'address_manager e il selettore degli indirizzi tornerà a 0. Dopo di che verrà attivato il segnale di done e la lettura in modalità sequenziale può essere considerata conclusa. Il timing diagram relativo all'ultima lettura è mostrato in figura 4.14.

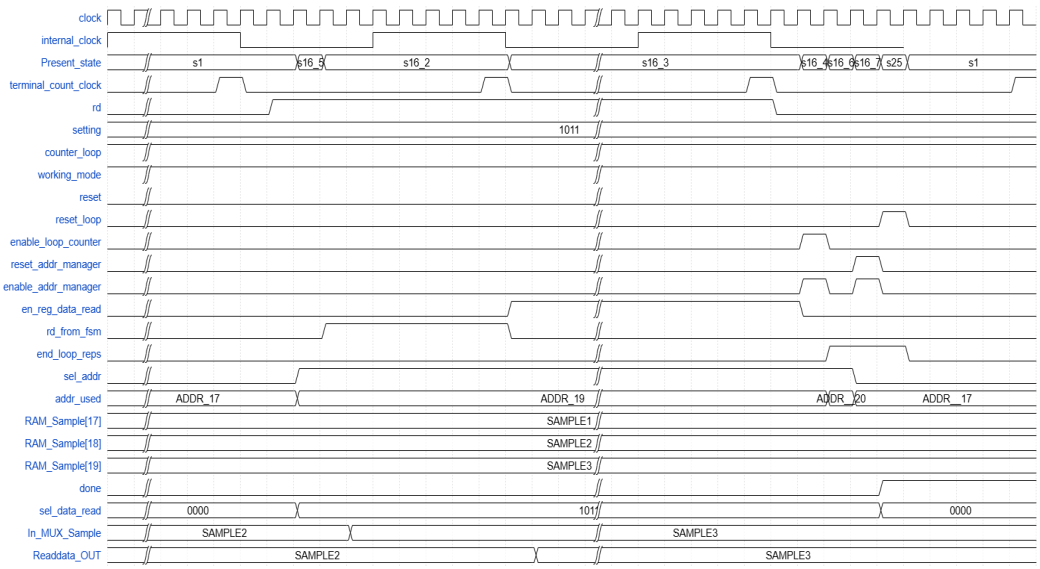


**Figura 4.14:** Timing diagram relativo alla lettura dell'ultimo vettore di campioni in modalità sequenziale

## 4.5 Simulazione del comportamento dell'analizzatore

Conclusa la sezione relativa ai timing utilizzati per il progetto, è giunto il momento di verificare il comportamento simulato del generatore di pattern. Cosi come fatto precedentemente, verranno omesse le simulazioni relative alla scrittura ed alla lettura dei vari registri, e si procederà con la simulazione di una analisi in modalità one-shot.

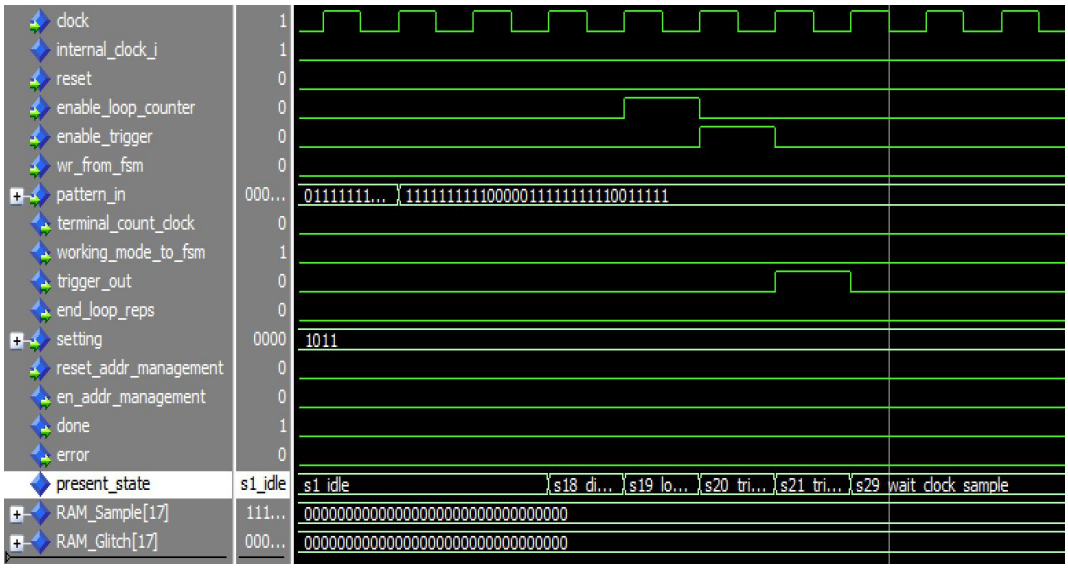


**Figura 4.15:** Simulazione del comportamento dell'analizzatore in modalità one-shot (Sezione relativa ai controlli).

In figura 4.15 viene mostrato il comportamento del sistema all'avvio dell'analisi. In questa sezione viene mostrata la fase relativa ai controlli sul pattern da analizzare. Acquisito il trigger, si entra in uno stato di attesa, dal quale si uscirà quando verrà campionato il segnale $terminal\_count\_clock = 1$.

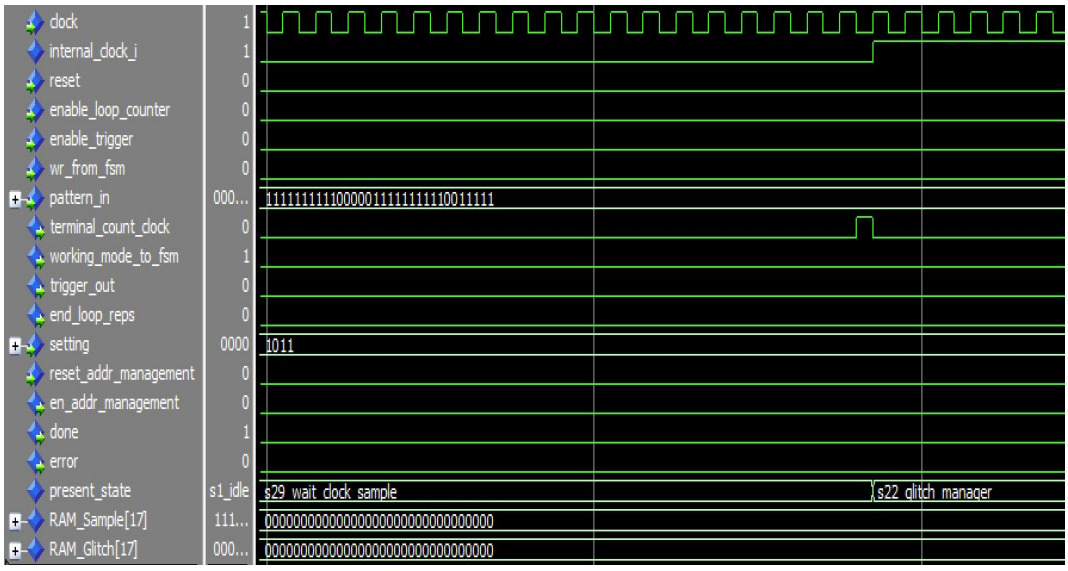


**Figura 4.16:** Simulazione del comportamento dell'analizzatore in modalità one-shot (Uscita dallo stato di attesa ed inizio analisi glitch).

Una volta individuato *terminal_count_clock* = 1, avrà inizio l'analisi sui glitch, ovvero ogni singolo bit del vettore da analizzare verrà campionato alla frequenza di *clock* per accertarne la stabilità (figura 4.16).

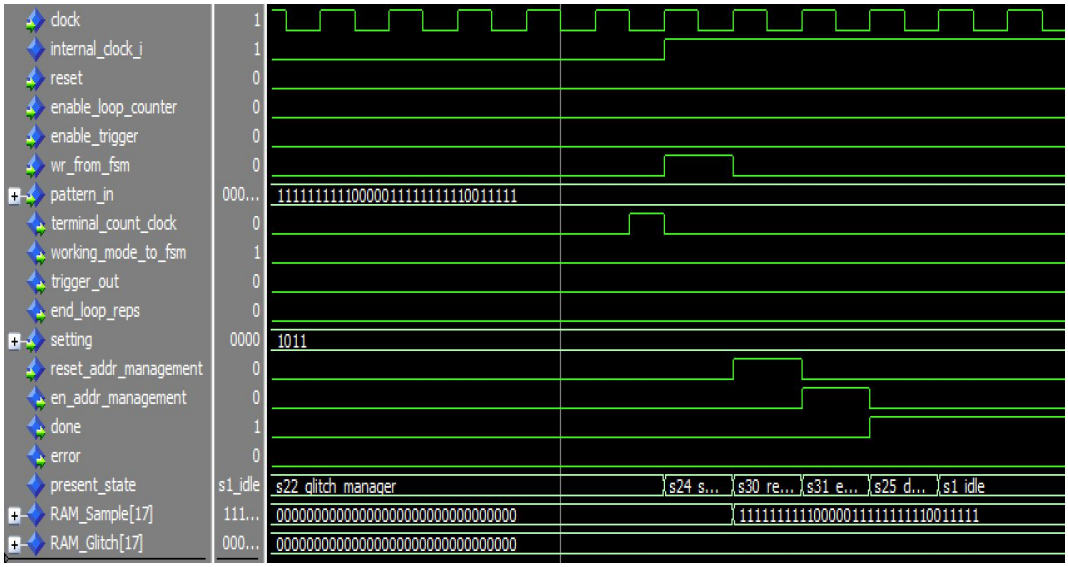


**Figura 4.17:** Simulazione del comportamento dell'analizzatore in modalità one-shot (Salvataggio in memoria dei campioni e dei glitch, con conclusione analisi).

In figura 4.17 viene mostrato il comportamento del sistema una volta terminata l'analisi dei glitch. Conclusa l'analisi, si proseguirà con il salvataggio dei dati in memoria e con il reset dei vari moduli utilizzati. Inoltre si andrà nello stato di done per segnalare la corretta riuscita dell'operazione.

Conclusa l'acquisizione dei campioni e dei glitch, è possibile leggere, sempre in modalità one-shot, i valori appena scritti in memoria.

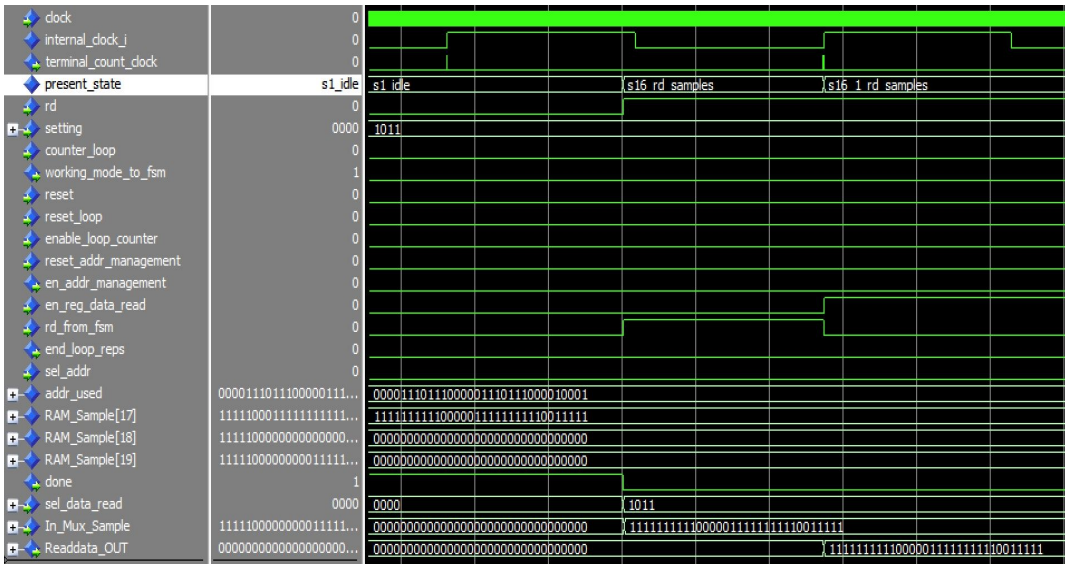


**Figura 4.18:** Simulazione del comportamento dell'analizzatore in modalità one-shot durante la lettura dei campioni.

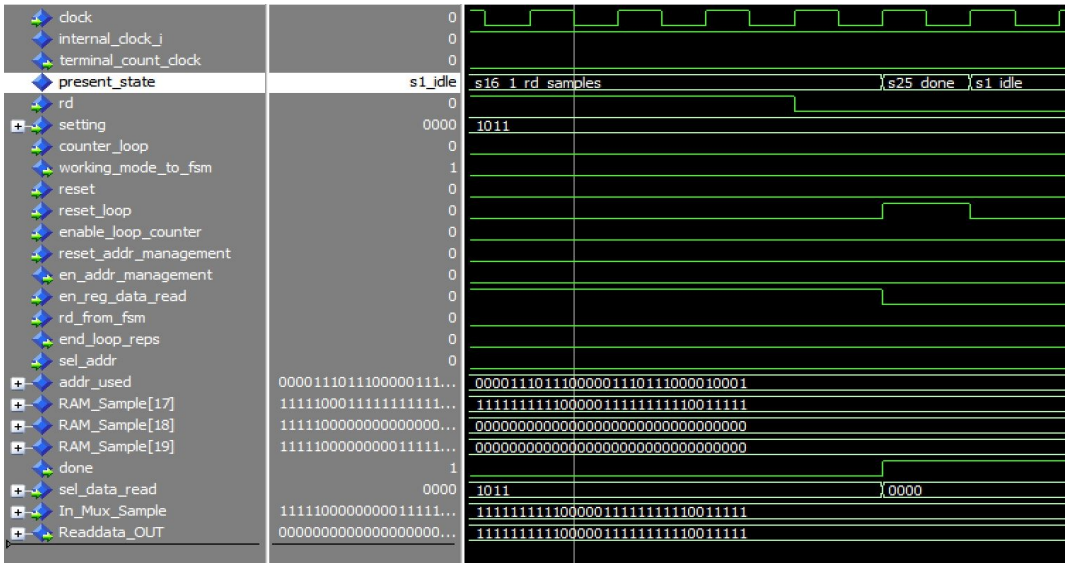


**Figura 4.19:** Simulazione del comportamento dell'analizzatore in modalità one-shot durante la lettura dei campioni.

In figura 4.18 viene mostrata l'operazione di lettura della memoria ed il passaggio del dato appena letto al multiplexer, il quale sarà collegato al registro *readdata_register*. Una volta campionato *terminal_count_clock* = 1 si passerà allo stato di done per segnalare la corretta riuscita dell'operazione(figura 4.19).

Definita la modalità di lavoro one-shot, è possibile visualizzare il comportamento dell'analizzatore in modalità sequenziale.

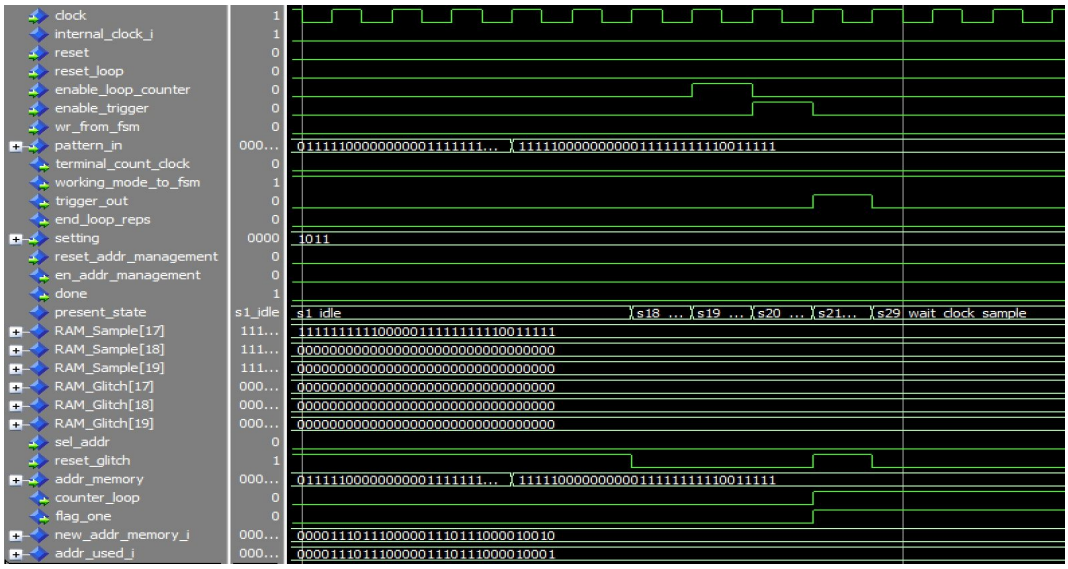


**Figura 4.20:** Simulazione del comportamento dell'analizzatore in modalità sequenziale - Prima Analisi (Sezione relativa al controllo del primo pattern).

Come si può osservare in figura 4.20, il comportamento relativo ai controlli è uguale a quanto visto in modalità one shot, e sarà così anche per i pattern successivi.

Anche la fase di attesa mostrata in figura 4.21 risulta essere in linea con quanto visto con la modalità one-shot.

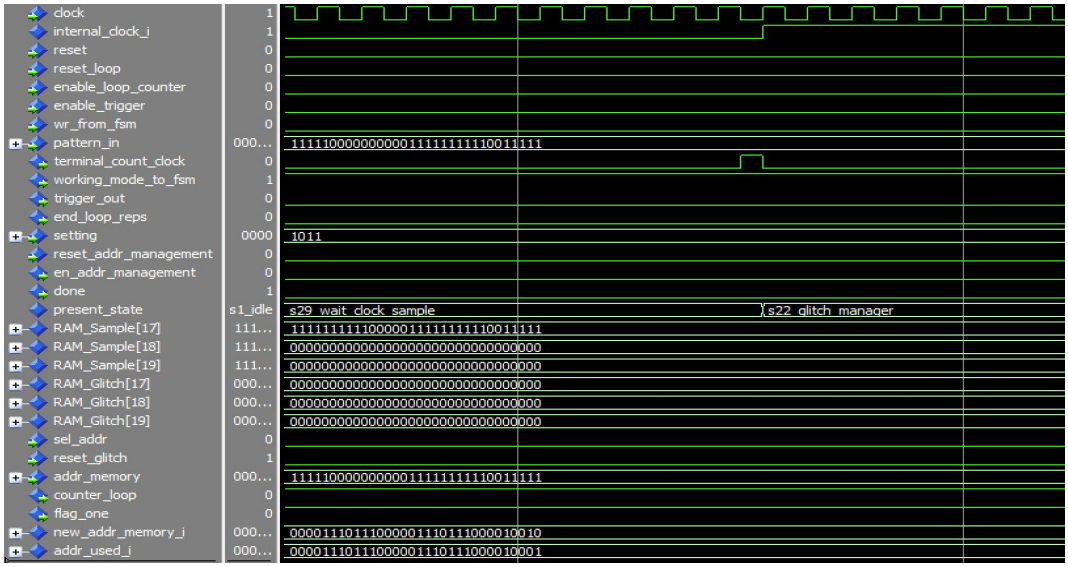


**Figura 4.21:** Simulazione del comportamento dell'analizzatore in modalità sequenziale - Prima Analisi (Uscita dallo stato di attesa ed inizio analisi glitch).

La prima differenza rispetto alla modalità one-shot vengono evidenziate in figura 4.22.

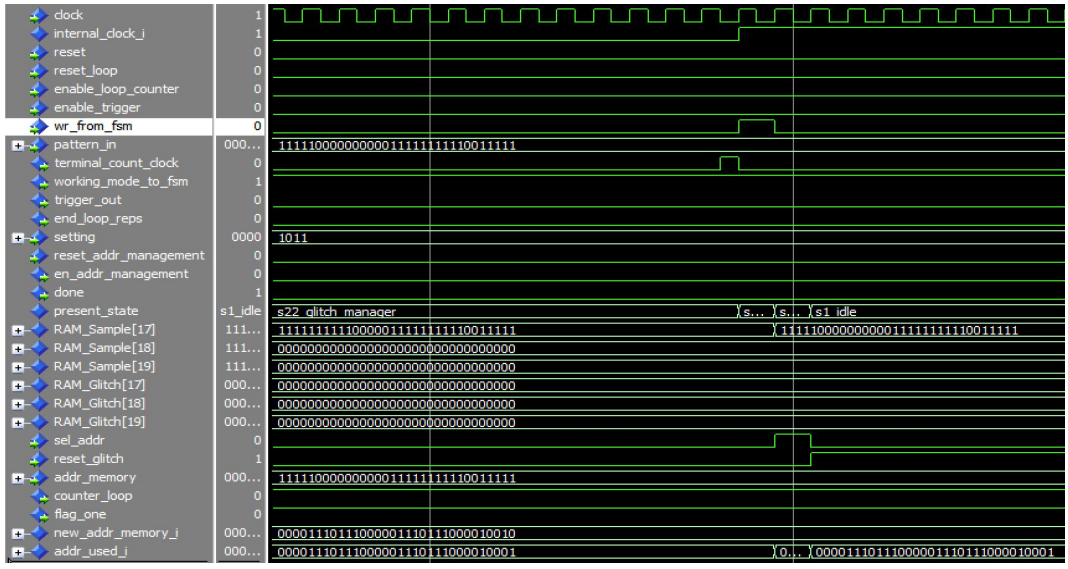


**Figura 4.22:** Simulazione del comportamento dell'analizzatore in modalità sequenziale -Prima Analisi (Salvataggio in memoria dei campioni e dei glitch e aggiornamento dell'indirizzo successivo).

Una volta terminata l'acquisizione dei glitch ed il loro salvataggio in memoria, non si procederà con lo stato di done, ma verrà aggiornato l'indirizzo per l'operazione successiva. Il done verrà bypassato in quanto l'operazione viene considerata conclusa al termine delle ripetizioni consentite dalle impostazioni. Per questo motivo si procederà allo stato di idle in attesa dell'istruzione successiva.

Come già detto, le operazioni di controllo, attesa e analisi dei glitch sono uguali per tutte le analisi, pertanto verranno trascurate. Osserviamo invece cosa accade

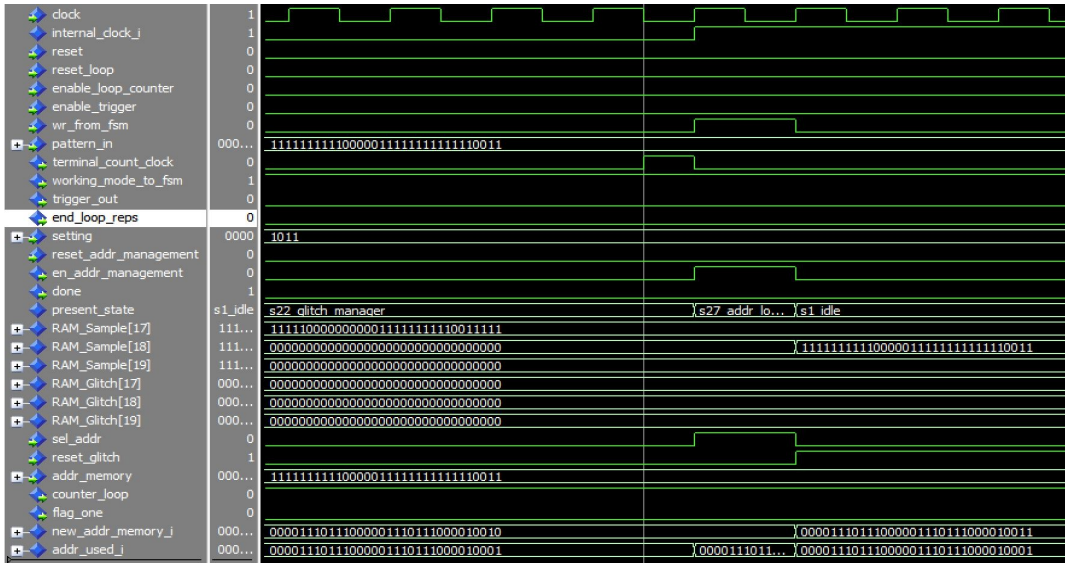


**Figura 4.23:** Simulazione del comportamento dell'analizzatore in modalità sequenziale - Seconda Analisi (Salvataggio in memoria dei campioni e dei glitch e aggiornamento dell'indirizzo successivo).

quando termina l'aacquisizione dei glitch per le analisi intermedie, cioè diverse dalla prima e dall'ultima analisi possibile.

Terminata l'acquisizione dei glitch, si andrà nello stato *s*27*_addr_loop_upgrade* in cui avviene la scrittura in memoria e contemporaneamente l'aggiornamento dell'indirizzo, affinché la scrittura avvenga nella locazione di memoria corretta. Dopo di ciò si passera nello stato di idle in attesa delle successive istruzioni.

Quando avviene l'ultima analisi, dettata dal valore presente nel registro *max_sequence_register*, il comportamento centrale dell'analisi è lo stesso. La differenza consiste negli stati che il sistema percorre una volta avvenuta l'operazione di scrittura dei dati in memoria. (Figura 4.24)

Infatti, dopo lo stato *s*27*_addr_loop_upgrade*, si passerà negli stati di reset del loop manager e dell'address manager, e successivamente nello stato di done per sancire la corretta riuscita dell'operazione sequenziale.

Osserviamo infine come avviene la lettura dei campioni in modalità sequenziale. In figura 4.25 viene mostrato l'inizio della prima operazione di lettura. Quindi dallo stato idle si passera allo stato di abilitazione della memoria per la lettura e successivamente allo stato di abilitazione del registro *readdata_reg*.

Una volta abilitato il registro, si resta nello stato in attesa che il segnale *rd* passi da 1 a 0. Una volta campionato ciò, verranno aggiornati il loop manager e l'address manager. Si controllerà il segnale $end\_loop\_reps = 0$ e si passerà allo stato di idle.

Una volta ricevuta una nuova istruzione di lettura, prima di ripercorrere la stessa strada mostrata in precedenza, passerà dallo stato *s*16_5*_update_address*, in cui verrà aggiornato il segnale di selezione del multiplexer dell'indirizzo, in modo da utilizzare l'indirizzo corretto(4.27). Dopo di che, il comportamento assunto dal sistema sarà quello mostrato con la prima lettura.

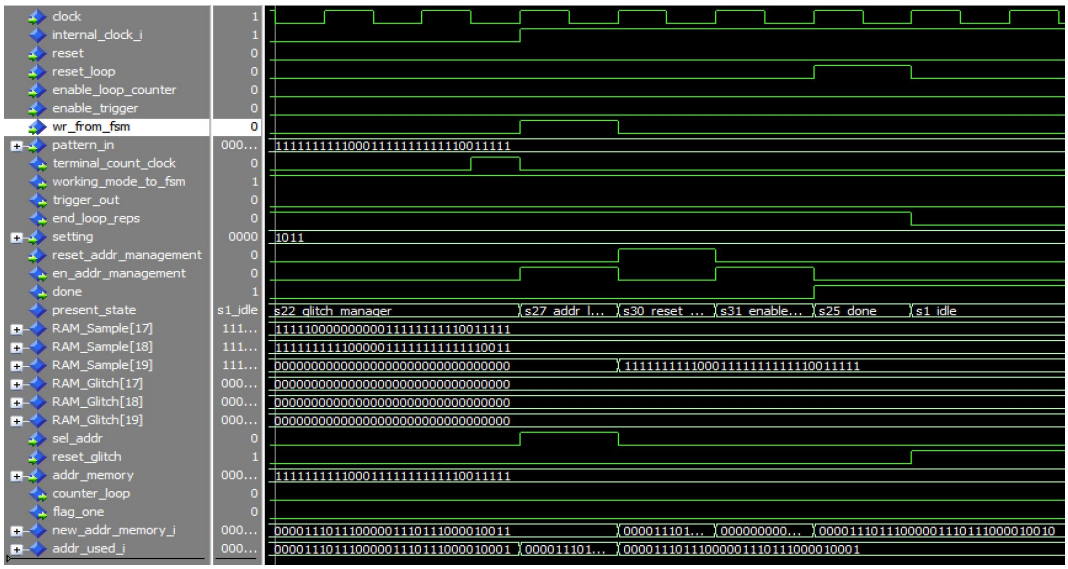


**Figura 4.24:** Simulazione del comportamento dell'analizzatore in modalità sequenziale - Ultima Analisi (Salvataggio in memoria dei campioni e dei glitch e aggiornamento dell'indirizzo successivo).

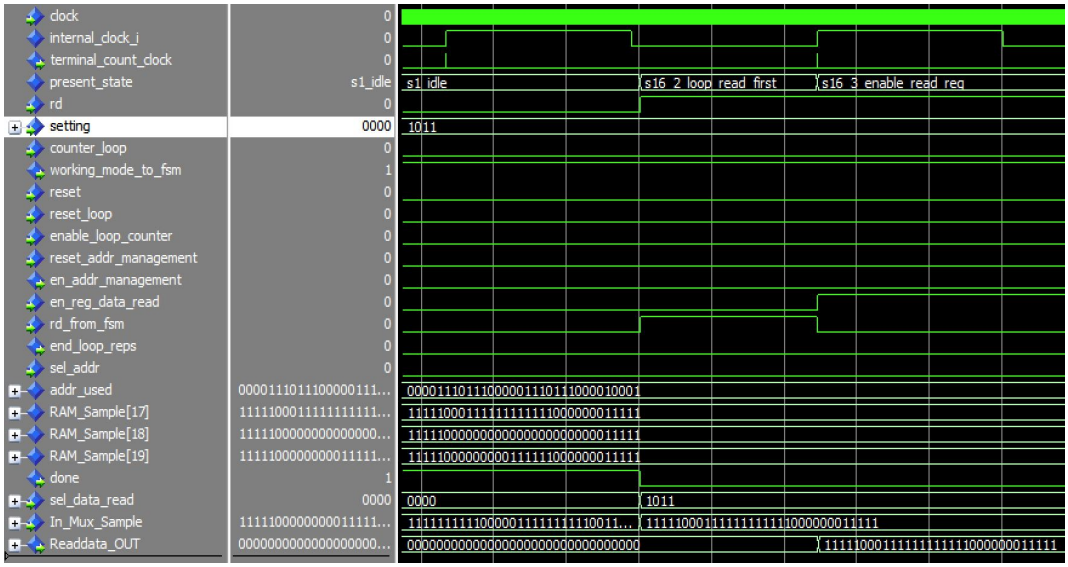


**Figura 4.25:** Simulazione del comportamento dell'analizzatore in modalità sequenziale durante la lettura dei campioni. (Prima lettura - Parte 1)

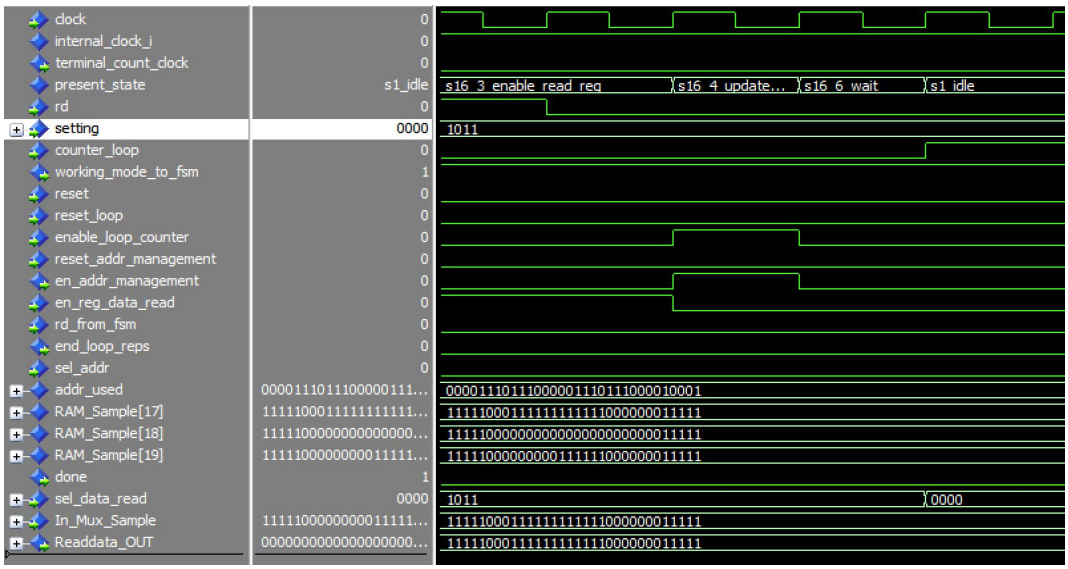


**Figura 4.26:** Simulazione del comportamento dell'analizzatore in modalità sequenziale durante la lettura dei campioni. (Prima lettura - Parte 2)

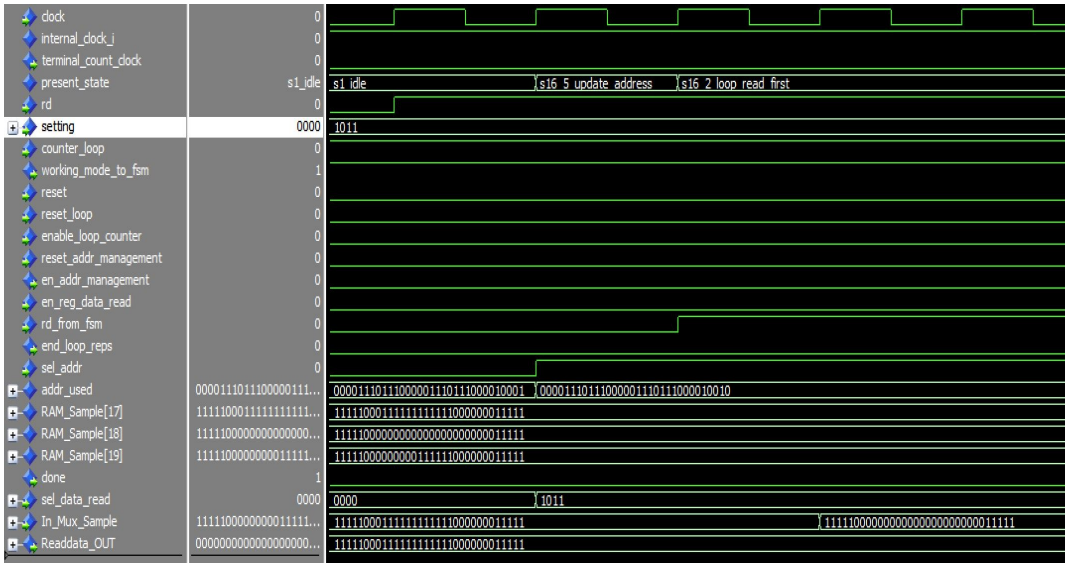


**Figura 4.27:** Simulazione del comportamento dell'analizzatore in modalità sequenziale durante la lettura dei campioni (Inizio seconda lettura).

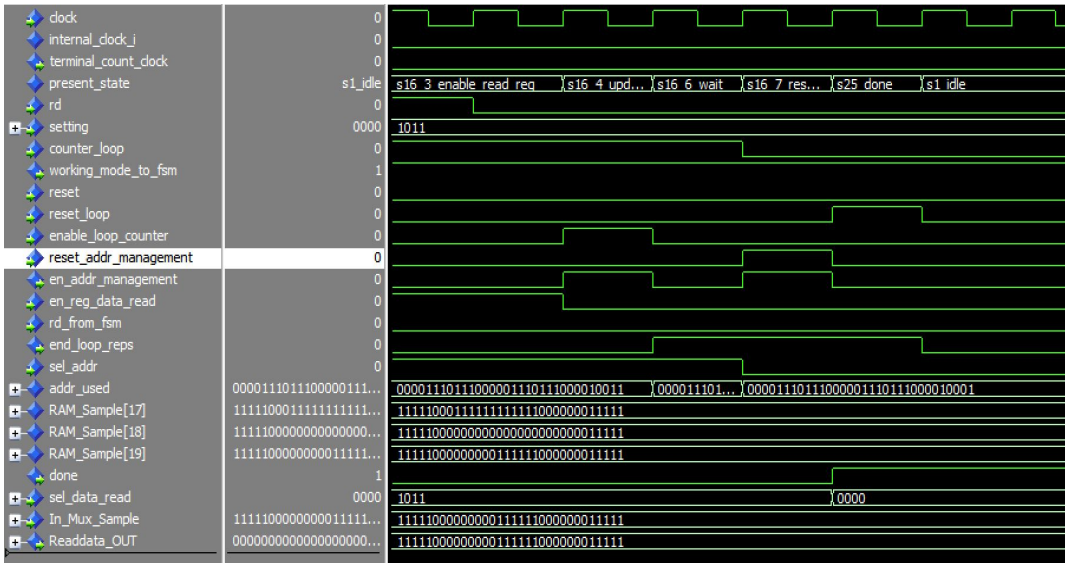


**Figura 4.28:** Simulazione del comportamento dell'analizzatore in modalità sequenziale durante la lettura dei campioni(Fine ultima lettura)

Osserviamo infine cosa accade al termine dell'ultima operazione di lettura (Figura 4.28). Una volta che il segnale *rd* viene disattivato, si andrà nello stato *s*16_4_*updtae_loop* che aggiorna il contatore del loop manager. Dopo di che si andrà nello stato di attesa, affinche possa avvenire il controllo sul flag *end_loop*. Quando questo flag è attivo, vuol dire che è stato raggiunto il numero massimo di ripetizioni consecutive. Per questa ragione, lo stato successivo non sarà *s*1_*idle* come nel caso successivo, ma *s*16_7_*reset_address_manager*, in cui viene resettato il loop manager. Dopo di che si va nello stato *s*25_*done* per concludere l'operazione di lettura.

## 4.6 Simulazione del sistema Bridge-Analizzatore

A questo punto possiamo considerare terminata l'analisi dell'analizzatore di stati logici inteso come componente, ma possiamo osservare come si comporta a livello di sistema, cioè con il componente bridge_spi_to_avalon collegato.

Osserviamo in modo superficiale come si comporta il sistema in modalità one-shot.

La modalità con cui si interagisce con i registri è quella spiegata nei capitoli precedenti. Si osservi come, una volta terminata la scrittura dei registri, viene inviato tramite il bridge_spi_to_avalon il pattern da analizzare. Questo viene analizzato, vengono ricercati i glitch e vengono salvati i campioni nelle varie memorie. Dopo di che sono state effettuate le letture dei dati salvati in memoria. Si osservi come in figura 4.29, nella memoria riservata ai sample, è stato memorizzato il pattern digitale inviato dall'spi. Questo vuol dire che il segnale è stato campionato nel modo corretto. Per quanto riguarda i glitch, si osservi come il dato memorizzato sia 0x00000000, in quanto il glitch detector non ha individuato alcun glitch. Questi due vettori, sono

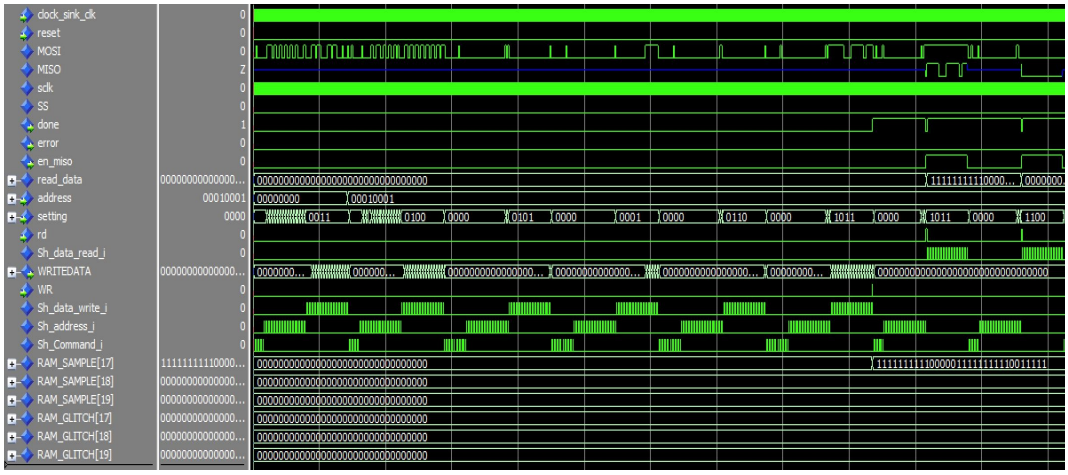


**Figura 4.29:** Simulazione del comportamento dell'analizzatore in modalità one-shot dal punto di vista sistemistico

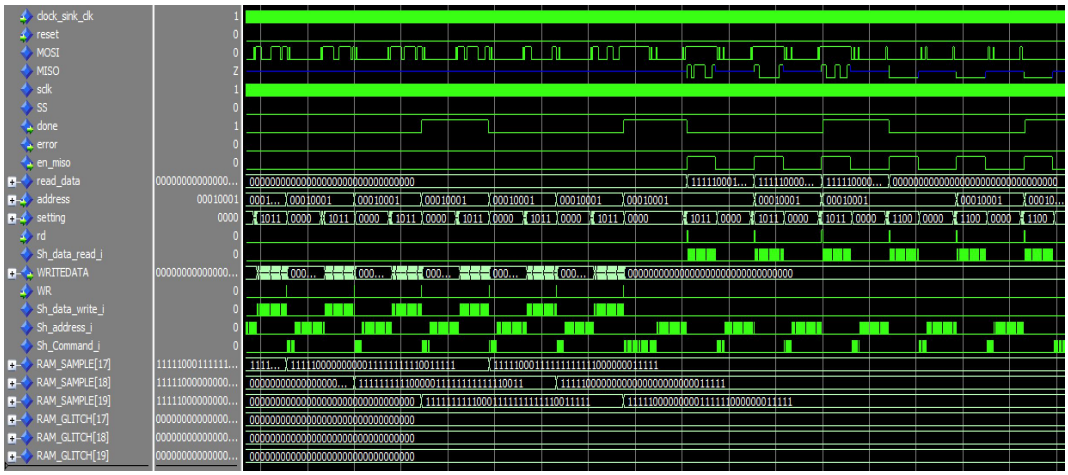


**Figura 4.30:** Simulazione del comportamento dell'analizzatore in modalità one-shot dal punto di vista sistemistico

stati successivamente letti ed inviati al microprocessore mediante la linea MISO dell'SPI.

In figura 4.30 viene invece mostrato il comportamento del sistema in modalità sequenziale.

La figura mostra chiaramente come siano stati salvati i campioni di tre pattern differenti in memoria, in tre allocazioni successive. Anche in questo caso non sono stati individuati glitch, pertanto la memoria assegnata ai glitch rimarrà vuota. Dopo la scrittura, è stata effettuata la lettura in modalità sequenziale, dei campioni appena scritti. Come nel caso precedente, sarà il segnale MISO che invierà al microprocessore i valori dei pattern acquisiti.

## 4.7 Risultato ottenuto

Una volta simulato il sistema ed aver verificato che il comportamento sia coerente con le specifiche richieste e progettate, si procede con il test sulla scheda VirtLAB. Anche

in questo caso ci si è avvalsi del terminale dal quale interfacciarsi al sistema. Una volta collegata la VirtLAB in modo corretto ed aver programmato sia il microcontrollore che l'FPGA, apparirà la schermata presente in figura 4.31.

```
*****************************************
**     VirtLAB Logic Analyzer (32bit)     **
*****************************************

>?
Usage:
    HOW WRITE AND READ ON REGISTER:
    wrrrrrrrrdddddddd ---> Write 32 bit data 'dddddddd' into register 'rrrrrrrr'
    rrrrrrrr          ---> Read 32 bit data from register 'rrrrrrrr'
    Register number (rrrrrrrr) is 32 bit, in hexadecimal format
    Data (dddddddd) is 32 bit, in hexadecimal format



    ADDRESS REGISTERS:

    CLOCK_DIVIDER_REGISTER      ==> 00000001
    CLOCK_SELECTION_REGISTER    ==> 00000002
    ADDR_REGISTER               ==> 00000003
    TRIGGER_MASK_REGISTER       ==> 00000005
    TRIGGER_CONDITION_REGISTER  ==> 00000006
    WORKING_MODE_REGISTER       ==> 00000008
    MAX_SEQUENCE_REGISTER       ==> 0000000a
    SAMPLE_ON_MEMORY            ==> 0000000b
    GLITCH_ON_MEMORY            ==> 0000000c

    All others are UNUSED!


    OTHER USEFULL INFORMATION FOR DATA TO WRITE INSIDE REGISTERS!!
    1) CLOCK_DIVIDER_REGISTER:   Divider       != 00000000
    2) CLOCK_SELECTION_REGISTER: Internal_Clock = 00000000
                                 External_Clock = 00000001
    3) ADDR_REGISTER:            Max value     = 000000FF
    4) WORKING_MODE_REGISTER:    One_Shot      = 00000000
                                 Sequence      = 00000001
    5) MAX_SEQUENCE_REGISTER:    Minimum value = 00000001
```

**Figura 4.31:** Interfaccia testuale dell'Analizzatore di stati logici

Questa interfaccia testuale è molto simile a quella del generatore di pattern. La prima sezione indica come scrivere le istruzioni, che è lo stesso di quanto visto in precedenza. La seconda sezione mostra invece gli indirizzi associati ai vari registri di dato e controllo. La terza sezione invece indica alcune informazioni importanti per il corretto utilizzo dell'analizzatore di stati logici, ovvero:

- Indica che il divisore di clock deve essere settato con un valore diverso da zero, altrimenti si entrerà in uno stato di errore.
- Indica che se si vuole utilizzare il clock interno della macchina, è necessario settare il registro corrispondente con il valore 0. Se invece si vuole utilizzare un clock esterno, deve essere settato con il valore 1.
- Indica che l'indirizzo più grande in memoria è 0x000000FF, che corrisponde alla locazione numero 255. Questo perchè la memoria progettata è solo di 256 locazioni e quindi sono sufficienti 8 bit di indirizzo, rispetto a 32 a disposizione.

Se i primi bit venissero settati con un valore diverso da 0, questi non avrebbero effetto.

- Indica che per selezionare la modalità di funzionamento one-shot, è sufficiente settare il valore 0 nell'ultimo bit del working_mode_reg. Se si vuole utilizzare la modalità sequenziale è necessario settare questo bit con il valore 1.
- Indica che il valore minimo da inserire nel registro max_sequence_register è 0x00000001

Osserviamo il tipico utilizzo dell'analizzatore in modalità one-shot (figura 4.32).

```
>w0000000100000001
Writing 00000001 to register 00000001
>w0000000200000000
Writing 00000000 to register 00000002
>w0000000300000000
Writing 00000000 to register 00000003
>w0000000500000001
Writing 00000001 to register 00000005
>w0000000600000001
Writing 00000001 to register 00000006
>w0000000800000000
Writing 00000000 to register 00000008
>w0000000b123ab6f1
Writing 123ab6f1 to register 0000000b
>r0000000b
Reading from register 0000000b: 123ab6f1
>r0000000c
Reading from register 0000000c: 00000000
```

**Figura 4.32:** Test dell'analizzatore di stati logici in modalità one-shot

Per utilizzare in modo corretto l'analizzatore di stati logici in modalità one-shot, è necessario scrivere nei vari registri i dati da utilizzare per le impostazioni. Il primo registro che è necessario scrivere è il *clock_divider_reg*, nel quale viene salvato un valore diverso da 0. In questo caso specifico è stato impostato il valore 0x00000001. Il secondo registro che è stato scritto è quello relativo alla locazione di memoria. In questo caso specifico è stato scritto il valore 0x00000003. Il terzo registro che è stato scritto è quello relativo alla maschera del trigger. Per rendere l'operazione più semplice e comprensibile è stato scelto il valore 0x00000001, in modo tale da mascherare tutti i bit del pattern ad eccezione del primo. Il quarto registro che è stato scritto è quello relativo alla condizione del trigger. Per lo stesso motivo, la condizione scelta è stata 0x00000001. Il quinto registro scritto è quello relativo al pattern da analizzare. In questo caso specifico è stato salvato il valore 0x123ab6f1, valore che rispetta la condizione di trigger. Questo dato verrà quindi campionato e salvato nella memoria $RAM\_SAMPLE$. Inoltre verrà analizzato per individuare la presenza di glitch, la cui posizione verrà salvata nella memoria $RAM\_GLITCH$. Per osservare i campioni salvati in memoria, l'istruzione successiva rappresenta la lettura del dato presente all'indirizzo scritto in *addr_reg*. Come si può osservare, il dato che viene letto è lo stesso inviato durante l'operazione precedente. Infine, l'ultima istruzione scritta è quella relativa alla lettura di eventuali glitch individuati, che in questo caso specifico sono assenti.

Osserviamo adesso il tipico utilizzo dell'analizzatore di stati logici in modalità sequenziale in figura 4.33.

```
>w0000000100000001
Writing 00000001 to register 00000001
>w0000000200000000
Writing 00000000 to register 00000002
>w0000000300000005
Writing 00000005 to register 00000003
>w0000000500000001
Writing 00000001 to register 00000005
>w0000000600000001
Writing 00000001 to register 00000006
>w0000000800000001
Writing 00000001 to register 00000008
>w0000000a00000003
Writing 00000003 to register 0000000a
>w0000000b00000001
Writing 00000001 to register 0000000b
>w0000000b00000011
Writing 00000011 to register 0000000b
>w0000000b00000111
Writing 00000111 to register 0000000b
>r0000000b
Reading from register 0000000b: 00000001
>r0000000b
Reading from register 0000000b: 00000011
>r0000000b
Reading from register 0000000b: 00000111
>r0000000c
Reading from register 0000000c: 00000000
>r0000000c
Reading from register 0000000c: 00000000
>r0000000c
Reading from register 0000000c: 00000000
```

**Figura 4.33:** Test dell'analizzatore di stati logici in modalità sequenziale

Il funzionamento dell'analizzatore di stati logici in modalità sequenziale è molto simile alla modalità one-shot. I registri impostati per la modalità one-shot devono essere impostati anche in questa modalità. Oltre a questi verrà scritto il registro *working_mode_reg* con il valore 0x00000001 per indicare la modalità sequenziale, il registro *max_loop_rep* con il valore 0x00000003 (in questo caso, ma è sufficiente un qualsiasi numero maggiore di 0x00000000). Semplicemente impostando questi due registri, siamo passati dalla modalità one-shot a quella sequenziale. A questo punto è possibile inviare i dati da analizzare in modo sequenziale. Per farlo è stato inviato il comando di acquisizione del pattern tre volte consecutive con i valori 0x00000001, 0x00000011 e 0x00000111. Questi tre valori sono stati salvati in modo automatico nelle locazioni di memoria 0x00000005, 0x00000006 e 0x00000007. Per constatare la corretta acquisizione di questi segnali nelle tre locazioni consecutive, sono state eseguite 3 letture consecutive in $RAM\_SAMPLE$ ed anche in questo caso abbiamo ottenuto i valori inviati in precedenza. Per concludere l'esposizione sull'utilizzo dell'analizzatore di stati logici, è stata effettuata una tripla lettura su $RAM\_GLITCH$, ottenendo che anche in questi casi non sono stati individuati glitch.

### 4.7.1 Report Post-Sintesi

La sintesi del "Pattern Generator" ha evidenziato alcune caratteristiche che è opportune riportare anche in questo documento. In particolare, è opportuno mostrare il numero di elementi logici utilizzati per la sintesi, la quantità di memoria occupata e la frequenza massima di funzionamento senza l'utilizzo di tecniche di ottimizzazione. I valori sono riportati nella tabella (4.5).

| **Memoria occupata** | **Elementi Logici** | **Frequenza Massima** |
|---|---|---|
| 16384/608256 (3%) | 1395/24624(6%) | 45,17 MHz |

**Tabella 4.5:** Compilation Report relativo al "Logic Analyzer"

Si osservi come sia la memoria utilizzata, sia gli elementi logici sono ben al di sotto del limite strutturale imposto dall'FPGA. Utilizzare solamente il'3% della memoria disponibile indica che il progetto è poco esigente in termini di memoria e lascia molto margine per future espansioni o aggiunta di nuove funzionalità. Inoltre anche il numero di "LE" è soddisfacente(6%) in quanto dimostra un utilizzo efficiente delle risorse, senza eccessivo spreco. Per quanto riguarda la massima frequenza(45,17MHz) è anche questa soddisfacente per il target a cui è destinato il progetto. In ogni caso, se fosse necessario aumentare la frequenza, si potrebbero applicare tecniche di pipeline o timing optimization per aumentarne la frequenza.

Si osservi inoltre come questi valori siano leggermente più alti rispetti a quelli riscontrati nel "Pattern Generator". Questo è certamente dovuto alla presenza di una seconda memoria RAM.

# Capitolo 5

# Conclusioni

## 5.1 Conclusioni

L'obiettivo di questo studio consiste nel realizzare delle architetture modulari da utilizzare sulla FPGA della scheda VirtLab sviluppata dal Politecnico di Torino. L'importanza di questa scheda è emersa durante la pandemia del 2020, che ha costretto gli studenti a non poter utilizzare i vari strumenti da laboratorio a causa delle misure restrittive.

In questo studio è stato progettato come lavoro preliminare un blocco che funzionasse da ponte tra il microcontrollore e l'FPGA presenti sulla scheda. Il compito svolto è principalmente basato sulla traduzione dall'interfaccia SPI all'interfaccia Avalon Memory-Mapped. L'obiettivo raggiunto in questa fase è stato quello di riuscire a scrivere e leggere dei dati all'interno di una memoria RAM. Questo lavoro è stato necessario poiché fondamento dei sistemi sviluppati nelle due fasi successive.

Concluso il progetto del "Bridge SPI to Avalon", lo studio è proseguito con la progettazione di un generatore di pattern digitali, ovvero un blocco che, in funzione delle impostazioni settate dall'utente, fornisca sui pin di uscita I/O i pattern richiesti. Ogni aspetto del progetto, dalla traduzione delle specifiche richieste fino al test reale sulla scheda, è stato approfondito in questo documento.

L'ultima sezione di questo lavoro è incentrata sullo sviluppo di un analizzatore di stati logici. Il ruolo svolto da questa architettura è quello di ricevere in ingresso un segnale, il quale verrà analizzato e campionato. I campioni raccolti e gli eventuali glitch verranno salvati in memoria RAM dedicate, al fine di poterne consultare i valori.

Ciò che è stato svolto in questo studio è quindi il progetto di architetture a partire dalle fasi iniziali, fino al test fisico sulla scheda. Il risultato ottenuto è quindi l'aggiunta di due funzionalità all'interno della scheda, indispensabili in un laboratorio di elettronica. Gli studenti, mediante queste architetture, saranno in grado di generare delle sequenze logiche personalizzabili utilizzando semplicemente la scheda ed un PC. Queste sequenze logiche potranno essere sfruttate, ad esempio, per fornire ad altri DUT, dei valori precisi per verificarne il comportamento, effettuare

test e debug. Al contrario, l'analizzatore di stati logici risulta fondamentale per acquisire dei segnali da altri DUT e verificarne la stabilità.

I risultati ottenuti dimostrano la versatilità e l'efficacia dell'approccio utilizzato, facilitando l'integrazione di nuove periferiche e migliorando la flessibilità della scheda Virtlab.

## 5.2 Possibili sviluppi futuri

I progetti svolti, pur essendo funzionanti, hanno la possibilità di essere migliorati in futuro, in modo da rendere il sistema più versatile e potente. Ecco alcune strade interessanti che potrebbero essere percorse in futuro:

- Aggiunta di ulteriori interfacce compatibili con il generatore di pattern e l'analizzatore di stati logici. In particolare potrebbe risultare utile sviluppare dei $Bridge\_UART\_to\_AvalonMM$ o $Bridge\_I2C\_to\_AvalonMM$.
- Creazione di un'interfaccia grafica (GUI) per configurare i parametri delle architetture in modo più intuitivo.
- Espansione delle funzionalità del generatore di pattern, ad esempio aggiungendo la possibilità di caricare dei pattern pseudo-casuali, utili per testare la robustezza dei sistemi digitali.
- Espansione delle funzionalità dell'analizzatore, ad esempio aggiungendo ulteriori canali di acquisizione e permettendo la manipolazione di questi segnali effettuando operazioni aritmetiche sui campioni ottenuti.
- Ottimizzazione del timing e delle risorse utilizzate.

In conclusione, questo studio fornisce agli utilizzatori della scheda VirtLab dei nuovi strumenti pratici e delle nuove funzionalità utilizzabili da subito, pur lasciando la possibilità di aggiungere migliorie ed ulteriori sviluppi futuri.

# Bibliografia


[1] Massimo Ruo Roch e Maurizio Martina. «VirtLAB: A Low-Cost Platform for Electronics Lab Experiments». In: *Sensors* 22.13 (2022). ISSN: 1424-8220. DOI: 10.3390/s22134840. URL: https://www.mdpi.com/1424-8220/22/13/4840 (cit. a p. 2).

[2] Massimo Ruo Roch. «VirtLab 1.2». In: *VirtLab documentation* (apr. 2021) (cit. a p. 4).

[3] Piyu Dhaker. «Introduction to SPI Interface». In: *Analog Devices. Inc* 52.09 (set. 2018). URL: https://www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf (cit. a p. 8).

[4] Università del Salento. *Il protocollo SPI*. Rapp. tecn. Università del Salento. URL: https://www.unisalento.it/documents/20152/4871996/SLIDES+LEZIONE+7-3Dispensa+SPI+Interface-EVIDENZIATA.pdf/d14ef5a0-91c4-09b6-1983-00d66067cbe6?version=1.0&download=true (cit. a p. 8).

[5] Intel Corporation. «Avalon® Interface Specifications». In: (gen. 2022). URL: https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/memory-mapped-interfaces.html (cit. a p. 11).

[6] Andrew Stuart Tanenbaum. «Architettura dei calcolatori. Un approccio strutturale». In: (2006), pp. 224–230 (cit. a p. 12).

[7] Donins U. e Osis J. «Unified modeling language». In: *Elsevier eBooks* (2017), pp. 3–51. DOI: https://doi.org/10.1016/b978-0-12-805476-5.00001-0 (cit. a p. 23).

# Ringraziamenti

Finalmente questo lungo percorso è giunto alla conclusione. Ci sono stati momenti difficili in cui un piccolo sassolino in mezzo la strada sembrava una montagna insuperabile, ma anche e soprattutto momenti bellissimi. Sono stati tali perché al mio fianco ho avuto delle persone speciali che mi hanno aiutato e non hanno mai smesso di starmi vicino.

Un ringraziamento speciale va ai miei genitori, che hanno permesso questo splendido viaggio. Il loro sostegno è stato fondamentale sotto tutti i punti di vista. Grazie per aver creduto in me, nelle mie capacità e nella persona che sono diventato lontano da voi.

Un ringraziamento va anche alla mia sorellina, che con il suo carattere forte è riuscita difendermi dalle grinfie di mia madre nei momenti di difficoltà. Avere qualcuno come te nella propria vita è un privilegio.

Un ringraziamento ai miei nonni, che con le loro telefonate e i loro "pacchi da giù" mi hanno fatto sentire a casa pur essendo lontano chilometri. Vi voglio bene.

Un ringraziamento va a Marco e Vincenzo, i miei migliori amici. Vi ho incontrato all'inizio di questa strada e da allora siamo stati sempre insieme. Non ci sono parole per descrivere il bene che vi voglio... anzi: SIETE DEI CANI!

Ringrazio tantissimo il mio collega Antonio, con cui ho affrontato questo viaggio fianco a fianco, esame dopo esame. Senza di te probabilmente sarei ancora in triennale ed invece siamo riusciti a platinare questo storia.

Ringrazio Antonello, Alessio, Salvo e Lorenzo. Voi siete stati più che semplici coinquilini. Per me siete come dei fratelli.

Un ringraziamento a tutti gli amici che ho conosciuto in questi anni qui a Torino e a quelli che ho lasciato ad Augusta. Siete stati la miglior compagnia che potessi desiderare.

Un ringraziamento speciale va a Miki, l'amore della mia vita. Averti incontrata ha cambiato le mie priorità ed il modo di vedere le cose. Grazie per avermi sostenuto, consolato e fatto forza quando ne avevo bisogno. Senza il tuo aiuto probabilmente non ce l'avrei mai fatta. Ti amo <3

Infine un ringraziamento a me stesso, per essere riuscito a tenere duro e per aver terminato quello che avevo iniziato. Ah, la mia testa dura...