

Relazione Prova Finale:

Progetto di Reti Logiche

Professore Fornaciari William

**Indice:**

Conclusioni 7

Ottimizzazioni 7

Copertura dei cammini 7

Reset 7

Stati della FSM 7

Risultato della sintesi 7

Introduzione 2

Funzionamento 2

Interfaccia del componente 3

Scopo del progetto 2

Specifiche del progetto 2

Scelte progettuali 4

Descrizione generale 4

Scelte implementative 5

Segnali e registri implementati 4

Stati della FSM 6

Testing 6

Altri test 7

Casi limite 6

Test generali 6

# 1 Introduzione

## 1.1 Scopo del progetto

Lo scopo di questo progetto VHDL è l’implementazione di un modulo hardware, mediante il linguaggio di programmazione VHDL, in grado di interfacciarsi con una memoria e indirizzare dati da essa verso uno dei quattro canali di uscita disponibili. Le indicazioni circa il canale da utilizzare e l’indirizzo di memoria a cui accedere vengono forniti mediante un ingresso seriale da un bit, mentre le uscite del sistema, ovvero i succitati canali, forniscono tutti i bit della parola di memoria in parallelo.

## 1.2 Specifiche del progetto

Il modulo da implementare ha **due ingressi primari da 1 bit** (**W** e **START**) e **5 uscite primarie**. Le uscite sono le seguenti: quattro da 8 bit (**Z0, Z1, Z2, Z3**) e una da 1 bit (**DONE**). Inoltre, il modulo ha un segnale di clock **CLK**, unico per tutto il sistema e un segnale di reset **RESET** anch’esso unico.

Per quanto riguarda il segnale W, la sua analisi permette di:

da un lato, di identificare correttamente l’uscita desiderata su cui scrivere dei dati presi dalla memoria tramite i primi 2 bit

dall’altro, di ricavare, tramite gli n bit restanti, l’indirizzo di memoria a cui accedere per estrapolare i dati desiderati

## 1.3 Funzionamento

All’istante iniziale, quello relativo al **reset** del sistema, le uscite sono inizializzate a zero:

quindi Z0, Z1, Z2 e Z3 a 0000 0000 mentre DONE è 0.

I dati in ingresso, ottenuti come sequenze sull’**ingresso primario seriale W, servono** per identificare:

* l’uscita coi primi 2 bit di intestazione
* indirizzo della memoria con i restanti N bit.

All’indirizzo di memoria è memorizzato il **messaggio da 8 bit** che deve essere indirizzato verso un canale di **uscita**.

L’uscita ottenuta analizzando i primi due bit di W, si riferisce ad uno dei 4 **canali d’uscita** (Z0, Z1, Z2 o Z3) sul quale deve essere indirizzato il messaggio. Il primo bit è il bit più significativo del canale di uscita, il secondo quello meno significativo, più in dettaglio:

* 00 identifica Z0
* 01 identifica Z1
* 10 identifica Z2
* 11 identifica Z3.

Gli **N** bit di indirizzo possono variare da 0 fino ad un **massimo di 16 bit**. Gli indirizzi di memoria sono tutti di 16 bit.

Se il numero di bit di N è inferiore a 16, l’indirizzo viene **esteso** con **0 sui bit più significativi**. Ad esempio:

* (N = 7) = 1010111 –> 0000000001010111
* (N = 16) = 1110000001010111 –> 1110000001010111
* (N = 0) = 0000000000000000 –> 0000000000000000

Tutti i bit su W devono essere letti sul fronte di salita del clock.

La sequenza di ingresso è valida quando il segnale START è alto (=1) e termina quando il segnale START è basso (=0).  
Il segnale START rimane alto per almeno di 2 cicli di clock e non più di 18 cicli di clock (2 bit del canale e 16 bit per il massimo numero di bit per indirizzare la memoria). Si assuma questa condizione sempre verificata (non è necessario gestire il caso in cui il segnale di START rimanga attivo meno di 2 cicli di clock o più di 18).

Come già detto le uscite Z0, Z1, Z2 e Z3 sono inizialmente 0, così come il segnale DONE mentre, quando DONE passa a uno, anche il canale sul quale viene mandato il messaggio cambia valore, a differenza dei restanti canali che rimangano inalterati. In sintesi, quindi: quando è 0 tutti i canali Z0, Z1, Z2 e Z3 devono essere a zero (32 bit a 0). Contemporaneamente alla scrittura del messaggio sul canale, il segnale DONE passa da 0 passa a 1 e rimane attivo per un solo ciclo di clock (dopo 1 ciclo di clock DONE passa da 1 a 0). In pratica quando DONE=1 il canale associato al messaggio cambierà il suo valore, mentre gli altri canali mostreranno l’ultimo valore trasmesso derivato dai messaggi ad essi associati.

Il segnale START è garantito rimanere a 0 fino a che il segnale DONE non è tornato a 0.  
Il tempo massimo per produrre il risultato (ovvero il tempo trascorso tra START=0 e DONE=1) deve essere inferiore a 20 cicli di clock.

Il modulo deve essere progettato considerando che prima del primo START=1 (e prima di richiedere il corretto funzionamento del modulo) verrà *sempre* dato il RESET (RESET=1). Una seconda (o successiva) elaborazione con START=1 non dovrà invece attendere il reset del modulo. Ogni qual volta viene dato il segnale di RESET (RESET=1), il modulo viene re-inizializzato.

## 1.4 Interfaccia del componente

Il componente da descrivere deve avere la seguente interfaccia.

entity project\_reti\_logiche is

port (

i\_clk : in std\_logic;

i\_rst : in std\_logic;

i\_start : in std\_logic;

i\_w : in std\_logic;

o\_z0   : out std\_logic\_vector(7 downto 0);

o\_z1   : out std\_logic\_vector(7 downto 0);

o\_z2   : out std\_logic\_vector(7 downto 0);

o\_z3   : out std\_logic\_vector(7 downto 0);

o\_done : out std\_logic;

o\_mem\_addr : out std\_logic\_vector(15 downto 0);

i\_mem\_data : in std\_logic\_vector(7 downto 0);

o\_mem\_we : out std\_logic;

o\_mem\_en : out std\_logic

);

end project\_reti\_logiche;

In particolare:

* il nome del modulo **deve essere** project\_reti\_logiche e deve essere presente **una sola architettura** per ogni entità; la violazione di queste indicazioni comporta l’impossibilità di eseguire il Test Bench e una conseguente valutazione di zero;
* i\_clk è il segnale di CLOCK in ingresso generato dal Test Bench;
* i\_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
* i\_start è il segnale di START generato dal Test Bench;
* i\_w è il segnale W precedentemente descritto e generato dal Test Bench;
* o\_z0, o\_z1, o\_z2, o\_z3 sono i quattro canali di uscita;
* o\_done è il segnale di uscita che comunica la fine dell’elaborazione;
* o\_mem\_addr è il segnale (vettore) di uscita che manda l’indirizzo alla memoria;
* i\_mem\_data è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
* o\_mem\_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
* o\_mem\_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0.

# 2 Scelte progettuali

## 2.1 Descrizione generale

Il componente progettato ha alla base una *Finite State Machine* (FSM), che gestisce, mediante una serie finita di stati opportunamente programmati, il processo di scrittura dei dati provenienti dalla memoria sul corretto nastro di uscita.

Il processo inizia quando il segnale di START si alza.

I primi due stati della macchina sono dedicati alla lettura dei primi due bit dell’ingresso W. In seguito, la lettura dei bit restanti si svolge tutta in uno stato, che ha il compito di mettere a 1 il segnale di attivazione dello *shift register* volto al salvataggio dei bit di W.

Quando il segnale di START si abbassa, la FSM manda l’indirizzo salvato nello *shift register* di 16 bit alla memoria. fatto ciò, si aspetta un ciclo di clock per permettere alla memoria di elaborare la richiesta.

A questo punto, il segnale DONE passa da zero a uno e, contemporaneamente, la macchina a stati stampa sulla uscita corretta il nuovo valore che arriva dalla memoria, mentre le altre tre uscite passano da zero all’ultimo valore salvato in precedenza (se esiste).

Durante tutto il processo, nel caso in cui si alzasse il segnale di RESET, la FSM passerà ad uno stato di attesa dove vengono azzerati tutti i segnali precedenti e aspetta il prossimo segnale alto di START.

## 2.2 Segnali e registri implementati

* out\_port -> *shift register* di 2 bit in cui salvare I primi 2 bit del segnale i\_w necessari a identificare l’uscita corretta. A seguito del processo di shift, che trasferisce il precedente valore salvato in out\_port(0) in out\_port(1), il nuovo bit da salvare viene messo in posizione 0
* enable\_out\_port -> segnale usato per l’attivazione del registro out\_port
* mem\_address\_registry -> *shift register* di 17 bit in cui vengono salvati i restanti bit di i\_w. Si è optato per un registro con un bit in più, rispetto ai 16 previsti, per correggerne il funzionamento che si attivava per un ciclo di clock di troppo, sbagliando quindi a salvare l’indirizzo di memoria
* enable\_shift\_register -> segnale usato per l’attivazione del registro enable\_shift\_register

* enable\_demultiplexer -> segnale usato per attivare il *demultiplexer*

* saved\_z0 -> registro di 8 bit usato come backup per salvare i precedenti valori di o\_z0 anche dopo che il canale in questione venga azzerato
* enable\_saved\_z0 -> segnale usato per attivare il processo di scrittura del registro saved\_z0
* saved\_z1 -> registro di 8 bit usato come backup per salvare i precedenti valori di o\_z1 anche dopo che il canale in questione venga azzerato
* enable\_saved\_z1 -> segnale usato per attivare il processo di scrittura del registro saved\_z1
* saved\_z2 -> registro di 8 bit usato come backup per salvare i precedenti valori di o\_z2 anche dopo che il canale in questione venga azzerato
* enable\_saved\_z2 -> segnale usato per attivare il processo di scrittura del registro saved\_z2
* saved\_z3 -> registro di 8 bit usato come backup per salvare i precedenti valori di o\_z3 anche dopo che il canale in questione venga azzerato
* enable\_saved\_z3 -> segnale usato per attivare il processo di scrittura del registro saved\_z3
* clone\_o\_done -> segnale usato per poter usufruire del segnale o\_done nei processi

## 2.3 Scelte implementative

L’organo motore del progetto è una macchina a stati gestita da due processi:

* il primo che regola il cambiamento degli stati della macchina.
* il secondo che gestisce i segnali.

La FSM passa da uno stato all’altro tramite una particolare condizione, specifica per ogni stato, che innesca il cambio oppure, in alcune occasioni, dopo un solo ciclo di clock.

Il cambiamento di stato è gestito tramite i *type signal*:

* curr\_state -> si occupa dello stato corrente. Il valore contenuto al suo interno è quello su cui si basano entrambi i processi della FSM per selezionare lo stato attualmente attivo. Il tutto avviene tramite uno switch case.
* next\_state -> qui viene salvato il nuovo stato della FSM che dovrà essere sostituito al precedente durante il successivo fronte di salita di i\_clk.

Per la selezione della corretta uscita da aggiornare si è optato per un *demultiplexer* che, tramite uno *switch case*, analizza il valore del registro out\_port, decidendo quale segnale tra i 4 di attivazione dei registri (enable\_saved\_z0, enable\_saved\_z1, enable\_saved\_z2, enable\_saved\_z3) attivare mentre gli altri vengono messi a 0.

Il *demu*, quindi, non modifica direttamente i registri in questione che sono invece gestiti tramite la loro logica interna al fine di evitare errori di accesso concorrente

In caso si alzi il segnale i\_rst, ogni registro procederà a resettarsi autonomamente mentre per tutti gli altri segnali uno stato della FSM dedicato al reset sarebbe stato ridondante in quanto tutti vengono a prescindere inizializzati a 0 prima dello switch case per l’attivazione del codice relativa allo stato corrente.

## 2.4 Stati della FSM

Prima dello *switch case* che decide lo stato della FSM vengono sempre azzerati tutti i segnali di controllo e i registri o\_z0, o\_z1, o\_z2 e o\_z3 per evitare *LATCH*

Gli stati sono:

CHOSE\_OUTPUT\_1 -> Stato che si occupa del salvataggio del primo bit di i\_w nel registro out\_port tramite l’attivazione del rispettivo segnale

CHOSE\_OUTPUT\_2 -> Stato che si occupa del salvataggio del secondo bit di i\_w nel registro out\_port tenendo alto il segnale precedentemente attivato enable\_out\_port

ASK\_MEM -> Stato che si occupa del salvataggio dei restanti bit di i\_w nello *shift register* mem\_address\_registry tramite attivazione del rispettivo segnale. La macchina. Astati rimane in questo stato fino all’abbassamento del segnale i\_start

READ\_MEM -> Stato della durata di un solo ciclo di clock in cui si accede alla memoria per prelevare il dato

WAITING\_FOR\_DONE -> Stato della durata di un solo ciclo di clock cui viene attivato il demultiplexer, tramite segnale per aggiornare i registri di backup (saved\_z0, saved\_z1, saved\_z2, saved\_z3)

SERIALIZE -> Stato della durata di un solo ciclo di clock in cui vengono stampati i valori aggiornati sulle uscite, come da logica richiesto, e contemporaneamente alzato il segnale o\_done

WAITING\_FOR\_START -> Stato di attesa che dura un ciclo di clock e prepara la macchina ad un possibile nuovo start azzerando i segnali

# 3 Testing

## 3.1 Test generali

Oltre al test bench fornito (tb\_example\_23\_agg), sono stati provati altri test generali per testare il funzionamento del programma. Tra le situazioni verificate ci sono:

* Immagine che contiene schermata, testo, circuito

  Descrizione generata automaticamenteprogramma con i\_start alto per 18 cicli di clock in modo da avere l’indirizzo di memoria completamente preso da i\_w senza bisogno di essere esteso
* programma con i\_start altro per soli 2 cicli di clock, dove quindi si aggiorna solo il registro out\_port relativo al canale di uscita
* programma con multipli cicli in cui il segnale i\_start si alza e si abbassa per testare quindi il reset dei segnali mentre i registri di backup non vengono sovrascritti

## 3.2 Casi limite

Sono stati condotti anche test specifici, progettati per mettere il componente in situazioni estreme al fine di verificarne la robustezza e l'affidabilità. Sono stati analizzati scenari in cui il sistema viene sottoposto a condizioni di stress come:

* reset quando i\_start è alto
* scrittura su tutte le uscite
* sovrascrittura delle uscite

Questi test hanno confermato che il modulo VHDL è in grado di gestire situazioni critiche senza comportamenti anomali o errori. L'attenzione ai casi limite è cruciale per garantire stabilità e coerenza del progetto in ogni circostanza, garantendo così la sua affidabilità nelle applicazioni reali.

## 3.3 Altri test

Oltre ai test specifici progettati per individuare i casi limite precedentemente menzionati, la suite di test è stata ampliata mediante l'implementazione di test randomici per valutare ulteriormente il comportamento del componente. Il tutto è stato fatto tramite ad uno script Python programmato per generare automaticamente una serie di test casuali, ciascuno con input diversi, al fine di esaminare il modulo da diverse prospettive.

Il processo di generazione dei test casuali inizia con lo script Python che crea una serie di dati di input variabili, simulando situazioni reali e scenari inaspettati. Questi dati includono valori casuali, variazioni nei segnali di controllo e condizioni di start/stop diverse. Gli input casuali sono quindi applicati al modulo VHDL, e ne vengono registrati i risultati.

Dopo l'esecuzione dei test, i risultati ottenuti vengono raccolti e registrati in una nuova fonte di simulazione. Questa nuova fonte di simulazione contiene i dati di input generati dallo script Python, i risultati previsti e quelli effettivamente ottenuti dal modulo VHDL. Ciò consente una verifica dettagliata delle prestazioni del componente, identificando eventuali comportamenti imprevisti o errori che potrebbero emergere in situazioni reali.

L'uso di test randomici insieme a test specifici mirati fornisce una panoramica completa delle capacità del modulo VHDL, aiutando a garantire che il componente sia robusto e affidabile in una vasta gamma di scenari operativi.

# 4 Conclusioni

## Immagine che contiene testo, schermata, numero, Carattere Descrizione generata automaticamente4.1 Risultato della sintesi

Il componente sintetizzato supera correttamente tutti i test utilizzati nelle 3 simulazioni: *Behavioural*, *Post Synthesis Functional* e *Post-Synthesis Timing*.

## 4.2 Ottimizzazioni

### 4.2.1 Stati della FSM

All’inizio il progetto comprendeva meno stati nella FSM. Si è deciso durante il lavoro di aumentarne il numero, creando nuovi stati che passano da uno all’altro senza una condizione specifica, ma della durata di un solo ciclo di clock. Questo è stato fatto al fine di garantire un funzionamento più lineare dando a tutti i componenti la possibilità di aggiornarsi senza sovrascriversi a vicenda.

### 4.2.2 Reset

Inizialmente la macchina a stati era munita di uno stato dedicato al reset, col compito di resettare manualmente tutti i registri e i segnali in un unico punto del codice contemporaneamente. Per comodità e per la rimozione di alcuni *latch* l’idea è stata scartata optando poi per dei registri che si resettano indipendentemente nel caso di i\_rst alto.

### 4.2.3 Copertura dei cammini

Al fine di rimuovere tutti i latch dovuti all’ambiguità di *if* e *switch,* si è pensato di risolvere prima con una semplice aggiunta di rami *else*, nel primo caso, e di *case default*, nel secondo. A seguito però, a causa della persistenza del problema, si è virato verso una soluzione più semplice e meccanica: l’aggiunta, prima dello *switch case* per gli stati della FSM, di una riga per ogni segnale (e per i canali di uscita) di azzeramento, in modo che, a prescindere dallo stato corrente, tutti i valori siano sempre specificati e mai lasciati ambigui.