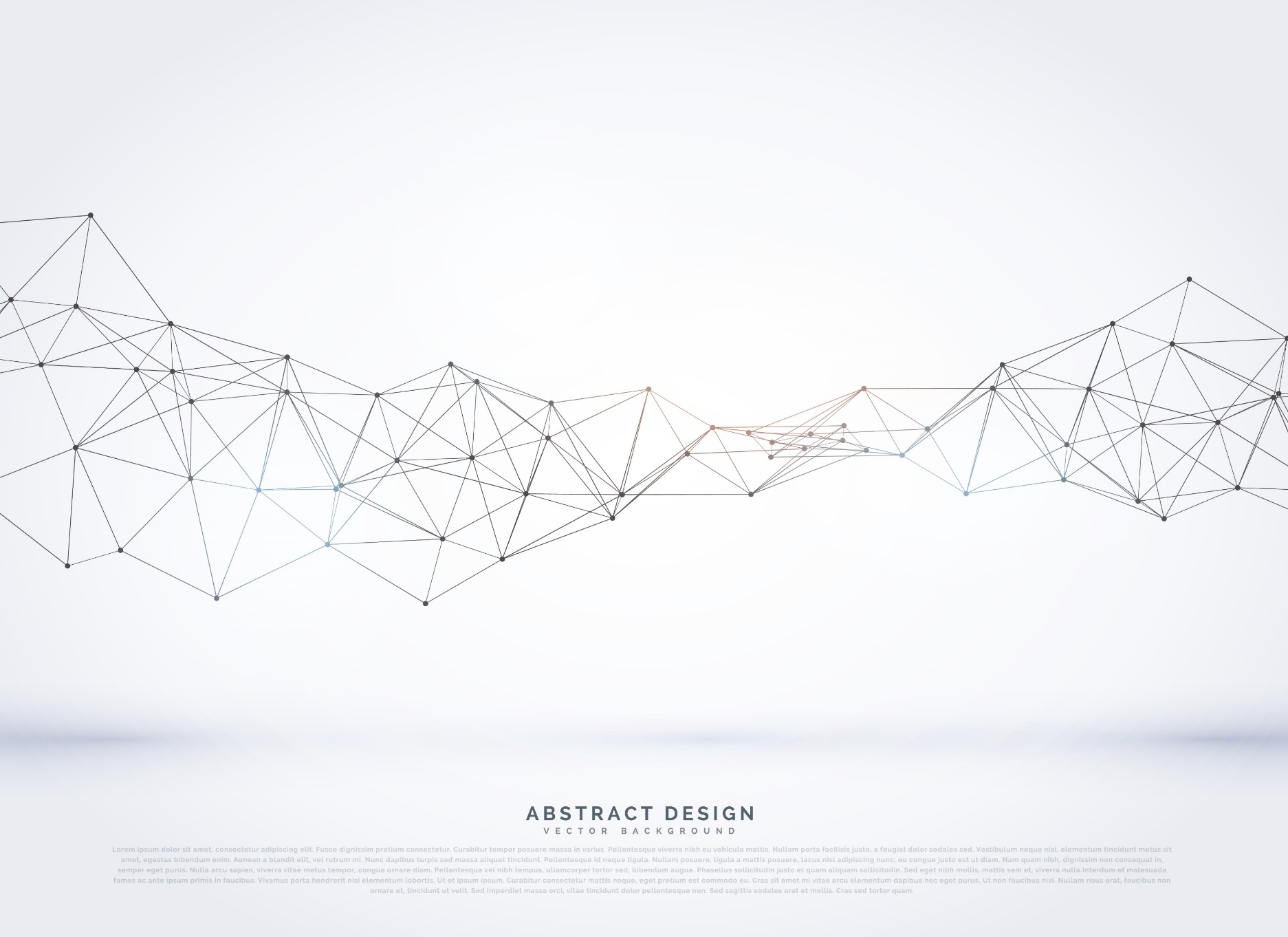


Corso di **Architettura dei Sistemi Digitali** – Prof. **Mazzocca**

Anno Accademico 2021-2022

Work Projects (Altro nome)

Nostri nomi

**INDICE**

[1 Multiplexer 5](#_Toc87719040)

[1.1 Progettazione del Multiplexer indirizzabile 16:1 5](#_Toc87719041)

[1.1.1 Codice VHDL del Multiplexer 7](#_Toc87719042)

[1.1.2 La simulazione del Multiplexer 10](#_Toc87719043)

[1.2 Progettazione della Rete di Interconnessione 16:4 12](#_Toc87719044)

[1.2.1 Codice VHDL della Rete di Interconnessione 16:4 13](#_Toc87719045)

[1.2.2 La simulazione della Rete 14](#_Toc87719046)

[1.2.3 C’è un’altra soluzione? Perché queste scelte? Sviluppo futuro 14](#_Toc87719047)

[2 Encoder BCD 15](#_Toc87719048)

[2.1 Progettazione del Encoder BCD 15](#_Toc87719049)

[2.1.1 Codice VHDL del Encoder BCD 16](#_Toc87719050)

[2.1.2 La simulazione del Encoder BCD 18](#_Toc87719051)

[2.1.3 Implementazione sulla board tramite i led 20](#_Toc87719052)

[2.1.4 Implementazione sulla board tramite il display 23](#_Toc87719053)

[2.1.5 C’è un’altra soluzione? Perché queste scelte? Sviluppo futuro 26](#_Toc87719054)

[3 Riconoscitore a 2 modalità 27](#_Toc87719055)

[3.1 Progettazione del Riconoscitore di sequenza a 2 modalità 27](#_Toc87719056)

[3.1.1 Schematico del Riconoscitore di sequenza a 2 modalità 29](#_Toc87719057)

[3.1.2 Codice VHDL del Riconoscitore di sequenza a 2 modalità 30](#_Toc87719058)

[3.1.3 La simulazione del Riconoscitore di sequenza a 2 modalità 35](#_Toc87719059)

# **1 Multiplexer**

Si realizza un Multiplexer seguendo le specifiche richieste dal cliente:

* **1.1** Progettare, implementare in VHDL e testare mediante simulazione un **multiplexer indirizzabile 16:1**, utilizzando un approccio di progettazione per composizione a partire da **multiplexer 4:1.**
* **1.2** Utilizzando il componente sviluppato al punto precedente, progettare, implementare in VHDL e testare mediante simulazione una **rete di interconnessione a 16 sorgenti e 4 destinazioni**.
* **1.3** Sintetizzare ed implementare su board il progetto della rete di interconnessione sviluppato al punto 1.2, utilizzando gli switch per fornire gli input di selezione e i led per visualizzare i 4 bit di uscita. Per quanto riguarda i 16 bit dato in input, essi possono essere precaricati nel sistema oppure immessi anch’essi mediante switch, sviluppando in questo secondo caso un’apposita rete di controllo per l’acquisizione.

## **1.1 Progettazione del Multiplexer indirizzabile 16:1**

Per la progettazione del Multiplexer in questione, come dalle specifiche, si utilizza un approccio di progettazione modulare, ovvero implementando componenti più piccoli e comporli. In questo caso, per la realizzazione del Multiplexer indirizzabile 16:1 si utilizzano cinque Multiplexer 4:1, che a loro volta saranno implementati come composizione di tre Multiplexer 2:1 notevoli.

2:1

4:1

16:1

**FIG. 1.0** – Concetto grafico

Il **Mux 2:1** è una macchina combinatoria notevole, indirizzabile avente una sola linea di selezione e due ingressi.

Mux **2:1**

**I0**

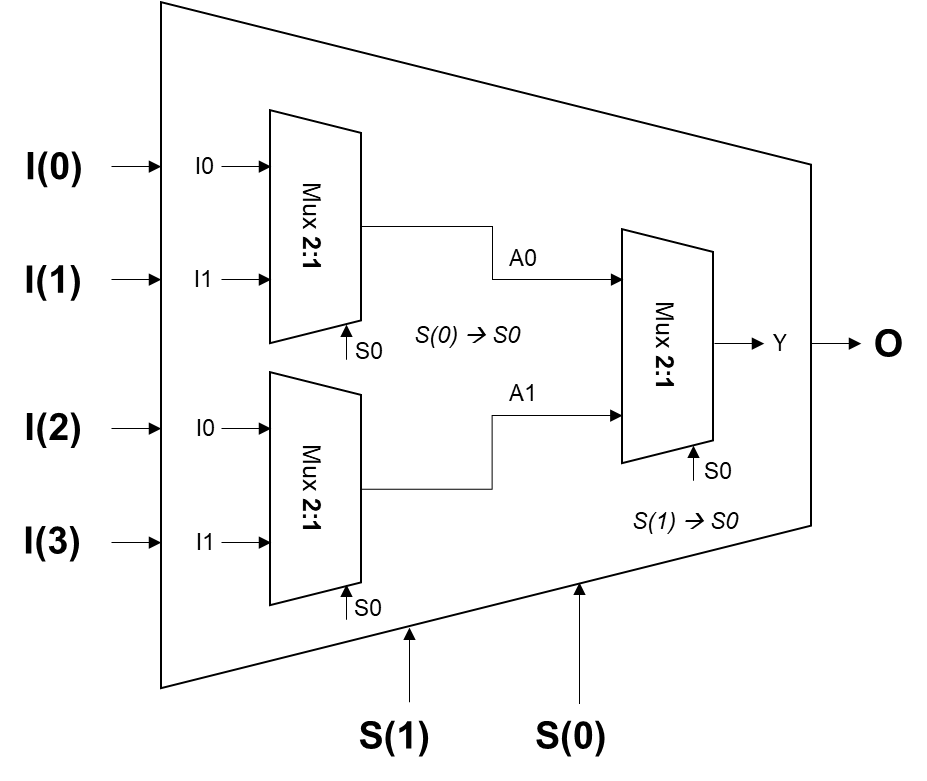
**Y**

**I1**

**S0**

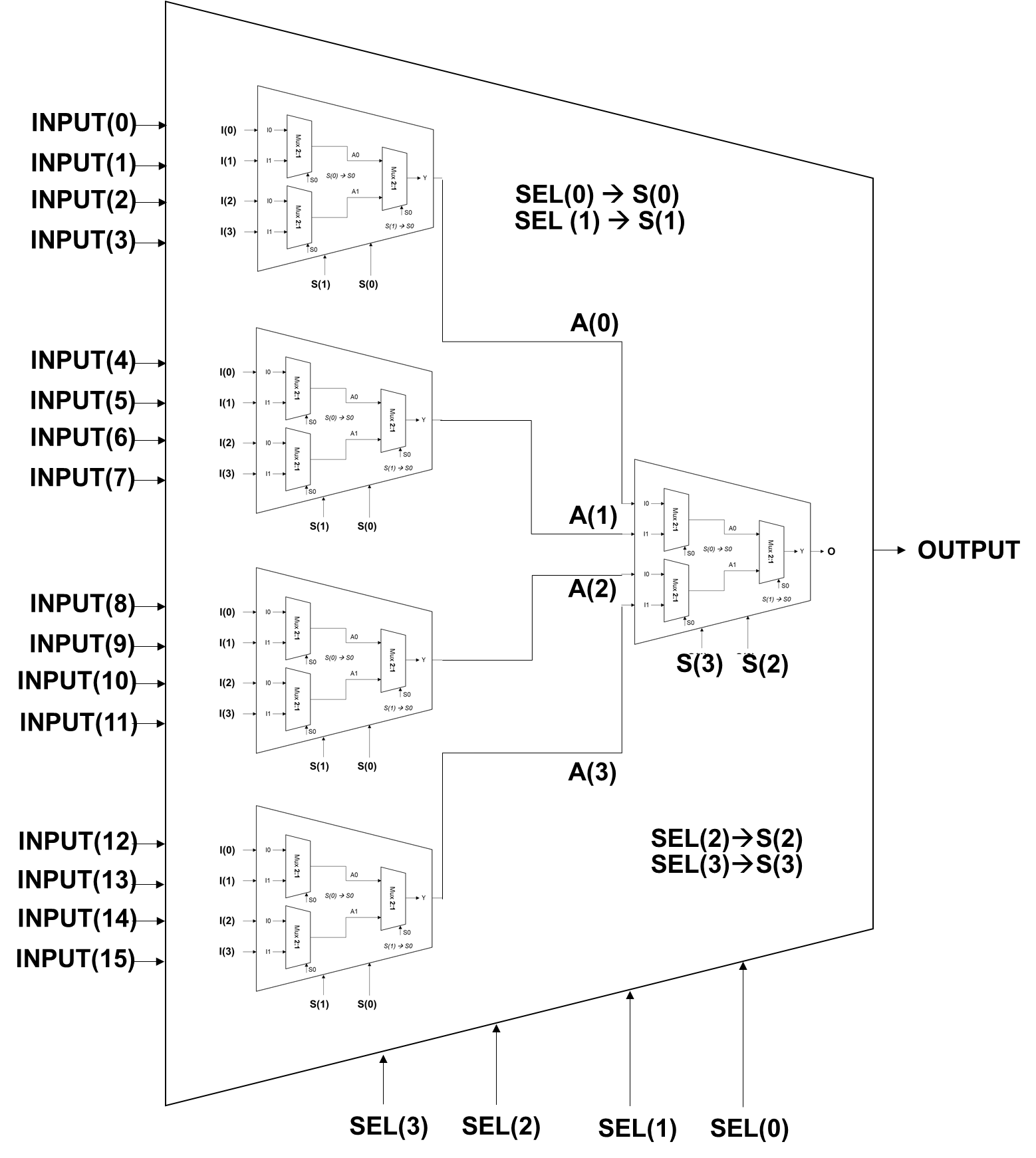
**FIG. 1.1** – Mux 2:1

Perché si utilizzano *tre* Mux 2:1 per la realizzazione del Mux 4:1? La risposta è nel seguente grafico:



**FIG. 1.2** – Mux 4:1

I **quattro** input mappati in ingresso ai primi due Mux 2:1 la cui uscita è determinata dall’input di selezione **S0**. I due segnali in uscita dai primi due Mux 2:1 entrano nell’ultimo Mux 2:1 selezionato dall’ingresso **S1** e l’uscita Y mappata in **O**.



**FIG. 1.3** – Mux 16:1

Si fa uso del Mux 4:1 composito per la realizzazione del Mux 16:1. Dalla **FIG 1.3**, si evincono i 16 ingressi di INPUT(0…15), mappati per ogni 4 input ad ogni Mux 4:1 composito. Le uscite sono gestite dai segnali interni A(0…3) i quali vanno in ingresso all’ultimo Mux 4:1 composito. L’uscita dell’ultimo Mux 4:1 composito corrisponde all’uscita complessiva del sistema.

### **1.1.1 Codice VHDL del Multiplexer**

Si inizia dal componente notevole, il **Multiplexer 2:1 (multiplexer\_2\_1.vhd)**:

Si definisce l’entità del MUX 2:1

entity **multiplexer\_2\_1** is

Port ( **I0** : in STD\_LOGIC;

**I1** : in STD\_LOGIC;

**S0** : in STD\_LOGIC;

**Y** : out STD\_LOGIC

);

end **multiplexer\_2\_1**;

E l’achitettura:

architecture **behavioral** of **multiplexer\_2\_1** is

begin

Y <= **I0** when S0 = **'0'** else

**I1** when S0 = **'1'** else

'-';

end behavioral;

Dopo l’implementazione del MUX 2:1, si passa all’implementazione del **Multiplexer 4:1 (multiplexer\_4\_1.vhd)** riutilizzando proprio il **multiplexer\_2\_1.**

entity **multiplexer\_4\_1** is

Port (

**I**: in STD\_LOGIC\_VECTOR (0 to 3);

**S**: in STD\_LOGIC\_VECTOR (1 downto 0);

**O**: out STD\_LOGIC

);

end **multiplexer\_4\_1**;

architecture structural of **multiplexer\_4\_1** is

-- riutilizzo del componente **multiplexer\_2\_1**

component **multiplexer\_2\_1** is

port (

**I0** : in STD\_LOGIC;

**I1** : in STD\_LOGIC;

**S0** : in STD\_LOGIC;

**Y** : out STD\_LOGIC

);

end component;

-- segnali interni

signal **A** : STD\_LOGIC\_VECTOR (0 to 1);

begin

-- si definiscono i multiplexer che si interfacciano con l’ingresso

**mux0\_1**: FOR j in 0 to 1 **GENERATE** m: **multiplexer\_2\_1**

-- si mappano con gli ingressi indicati dalle figure

port **map**(

**I0** => I(j\*2),

**I1** => I(j\*2 +1),

**S0** => S(0),

**Y** => A(j)

);

end **GENERATE**;

-- si mappa il multiplexer collegato all’uscita

mux2: **multiplexer\_2\_1**

port **map**(

**I0** => A(0),

**I1** => A(1),

**S0** => S(1),

**Y** => O

);

end **structural**;

Dopo l’implementazione del MUX 4:1, si passa all’implementazione del **Multiplexer 16:1 (multiplexer\_16\_1.vhd)** riutilizzando proprio il **multiplexer\_4\_1.**

entity **multiplexer\_16\_1** is

Port (

**INPUT**: in STD\_LOGIC\_VECTOR(0 to 15);

**SEL**: in STD\_LOGIC\_VECTOR(3 downto 0);

**OUTPUT**: out STD\_LOGIC

);

end **multiplexer\_16\_1**;

architecture structural of **multiplexer\_16\_1** is

-- riutilizzo del componente **multiplexer\_4\_1**

component **multiplexer\_4\_1** is

Port (

**I**: in STD\_LOGIC\_VECTOR (0 to 3);

**S**: in STD\_LOGIC\_VECTOR (1 downto 0);

**O**: out STD\_LOGIC

);

end component;

-- segnali interni

signal **A** : STD\_LOGIC\_VECTOR (0 to 3);

begin

-- si definiscono I 4 mux che si interfacciano con l’ingresso

**mux0\_3**: FOR j IN 0 to 3 **GENERATE** m: multiplexer\_4\_1

-- si mappano i segnali con i segnali di ingresso esterni

Port map (

**I**(0 to 3) => INPUT(j\*4 to j\*4+3),

**S**(1 downto 0) => SEL(1 downto 0),

**O** => A(j)

);

end GENERATE;

-- si definisce il 5° mux che si interfaccia con l’uscita

**mux4**: multiplexer\_4\_1

-- si mappano i segnali con i segnali di ingresso interni

port map (

**I**(0 to 3) => A(0 to 3),

**S**(1 downto 0) => SEL(3 downto 2),

**O** => OUTPUT

);

end structural;

### **1.1.2 La simulazione del Multiplexer**

Come di consueto, la simulazione sarà fatta stimolando l’implementazione del **multiplexer\_16\_1** attraverso il *Testbench* (multiplexer\_16\_1\_TB.vhd)

entity **multiplexer\_16\_1\_TB** is

end **multiplexer\_16\_1\_TB**;

architecture Behavioral of **multiplexer\_16\_1\_TB** is

component **multiplexer\_16\_1** is

Port (

INPUT: in STD\_LOGIC\_VECTOR (0 to 15);

SEL: in STD\_LOGIC\_VECTOR (3 downto 0);

OUTPUT: out STD\_LOGIC

);

end component;

signal **inputs** : STD\_LOGIC\_VECTOR (0 to 15);

signal **selections** : STD\_LOGIC\_VECTOR (3 downto 0);

signal **outputs** : STD\_LOGIC;

begin

--Unit Under Test

uut: multiplexer\_16\_1

-- mapping dei segnali da simulare

port map (

INPUT => inputs,

SEL => selections,

OUTPUT => outputs

);

-- TEST

stim\_proc: process

begin

--selezione linea INPUT0

wait for 10 ns;

inputs <= "**1000000000000000**";

selections <= "**0000**";

wait for 10 ns;

assert outputs <= '1'

report "Errore sulla Linea 0"

severity failure;

--selezione linea INPUT6

wait for 10 ns;

inputs <= "**0000001000000000**";

selections <= "**0110**";

wait for 10 ns;

assert outputs <= '1'

report "Errore sulla Linea 6"

severity failure;

--selezione linea INPUT12

wait for 10 ns;

inputs <= "**0000000000001000**";

selections <= "**1100**";

wait for 10 ns;

assert outputs <= '1'

report "Errore sulla Linea 12"

severity failure;

--selezione linea INPUT3

wait for 10 ns;

inputs <= "**0001000000000000**";

selections <= "**0011**";

wait for 10 ns;

assert outputs <= '1'

report "Errore sulla Linea 3"

severity failure;

--selezione linea INPUT15

wait for 10 ns;

inputs <= "**0000000000000001**";

selections <= "1111";

wait for 10 ns;

assert outputs <= '1'

report "Errore sulla Linea 15"

severity failure;

end process;

end Behavioral;

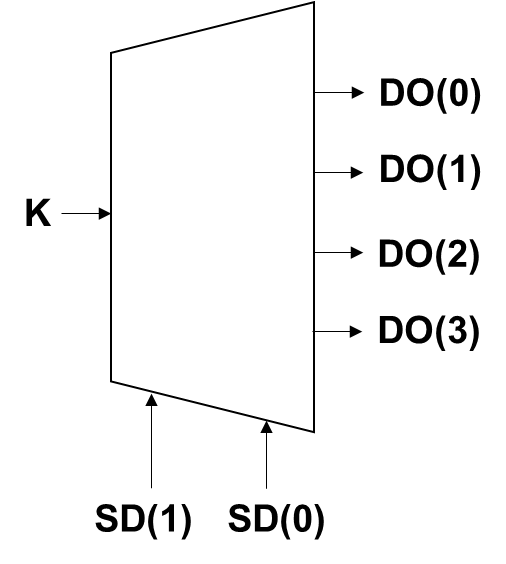


**FIG. 1.4** – Simulazione Multiplexer 16:1

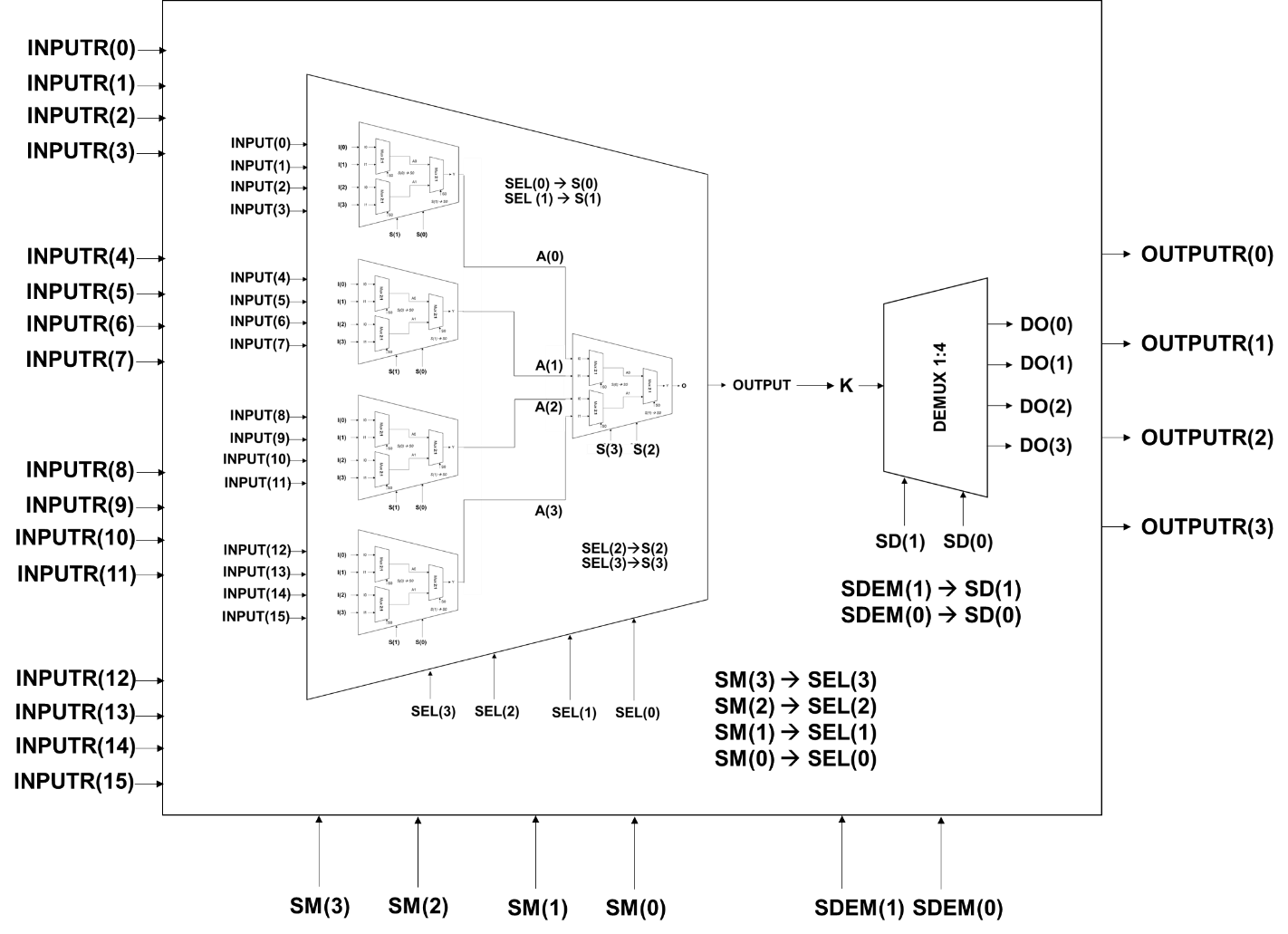
In definitiva, come è possibile notare dalla **FIG 1.4**, prendendo in esempio la simulazione all’istante 50ns si ha che l’**INPUT** è “000000000001000” e il **SEL** è “1100” ovvero viene selezionata la linea 12 (**c**) e quindi in uscita (**OUTPUT**) si avrà **1**.

## **1.2 Progettazione della Rete di** **Interconnessione 16:4**

Per la realizzazione di tale rete si utilizza il **MUX 16\_1** e con un’opportuna selezione inoltra il segnale in uscita del multiplexer in ingresso ad un **DEMUX 1\_4 (FIG. 1.5);** a sua volta, il demultiplexer lo inoltrerà su una delle quattro possibili destinazioni in modo tale da realizzare una vera e propria rete di interconnessione a 16 ingressi e 4 uscite.



**FIG. 1.5** – Demultiplexer 1\_4



**FIG. 1.6** – Rete di Interconnessione a 16 sorgenti e 4 destinazioni

Dalla **FIG 1.6** si trae l’approccio modulare proposto nel paragrafo **1** in cui ogni componente e composizione di altri componenti più semplici, in particolare gli ingressi della rete di interconnessione (**INPUTR[0…15]**) sono mappati agli ingressi del **MUX 16:1** (**INPUT[0…15 ]**). Di tali ingressi, soltanto uno viene posto in uscita (**OUTPUT**) sulla base del valore di selezione (**SM[0…3]**) indicato. Quest’ultimo è inoltrato all’ingresso del **DEMUX [1…4]** che in accordo con il valore **SD[0,1]** seleziona la destinazione di uscita (**OUTPUTR[0…3]**). Si noti dunque che il valore di **OUTPUTR** sarà formato da un valore di 4 bit di cui solo uno sarà alto (es. “0010”).

### **1.2.1 Codice VHDL della Rete di Interconnessione 16:4**

Si implementa inizialmente l’ulteriore componente della rete, ovvero il **DEMUX 1:4**:

entity **demultiplexer\_1\_4** is

Port (

**K** : in STD\_LOGIC;

**SD** : in STD\_LOGIC\_VECTOR (1 downto 0);

**DO** : out STD\_LOGIC\_VECTOR (0 to 3)

);

end **demultiplexer\_1\_4**;

architecture Behavioral of **demultiplexer\_1\_4** is

begin

-- l’uscita del demultiplexer\_1\_4 è un vettore di 4 bit pertanto in correlazione della selezione SD si utilizza la concatenazione tra K e i valori di bit rimanenti.

process (SD,K)

begin

if SD="**00**" then

DO <= **K&"000";**

elsif SD=**"01"** then

DO <= **'0'&K&"00"**;

elsif SD=**"10"** then

DO <= **"00"&K&'0'**;

elsif SD=**"11"** then

DO <= **"000"&K**;

end if;

end process;

end Behavioral;

A questo punto si passa all’implementazione generale della rete di interconnessione:

entity **rete\_16\_4** is

Port (

-- si inizializza INPUTR con tutti “1” poiché i switch presenti sulla board non sono sufficienti a rappresentare tutti gli ingressi della rete in questione (sia quelli delle linee e sia quelli di selezione complessivi)

**INPUTR** : in STD\_LOGIC\_VECTOR(0 to 15) := "1111111111111111";

**SM** : in STD\_LOGIC\_VECTOR(3 downto 0);

**SDEM** : in STD\_LOGIC\_VECTOR(1 downto 0);

**OUTPUTR** : out STD\_LOGIC\_VECTOR(0 to 3)

);

end **rete\_16\_4**;

architecture structural of **rete\_16\_4** is

component **multiplexer\_16\_1** is

Port (

**INPUT**: in STD\_LOGIC\_VECTOR(0 to 15);

**SEL**: in STD\_LOGIC\_VECTOR(3 downto 0);

**OUTPUT**: out STD\_LOGIC

);

end component;

component **demultiplexer\_1\_4** is

Port (

**K** : in STD\_LOGIC;

**SD** : in STD\_LOGIC\_VECTOR (1 downto 0);

**DO** : out STD\_LOGIC\_VECTOR (0 to 3)

);

end component;

signal **A** : STD\_LOGIC;

begin

mux: multiplexer\_16\_1

port map (

**INPUT**(0 to 15) => INPUTR(0 to 15),

**SEL**(3 downto 0) => SM(3 downto 0),

**OUTPUT** => A

);

demux: demultiplexer\_1\_4

port map(

**K** => A,

**SD**(1 downto 0) => SDEM(1 downto 0),

**DO**(0 to 3) => OUTPUTR(0 to 3)

);

end structural;

### **1.2.2 La simulazione della Rete**

**1.2.3 C’è un’altra soluzione? Perché queste scelte? Sviluppo futuro**

Implementazione Rete di Interconnessione con bottoni per rete di acquisizione.

**2 Encoder BCD**

Si realizza un riconoscitore di sequenza a 2 modalità seguendo le specifiche richieste dal cliente:

* 1. Progettare, implementare in VHDL e testare mediante simulazione una rete che, data in ingresso una stringa binaria X di 10 bit X9 X8 X7 X6 X5 X4 X3 X2 X1 X0 che corrisponde alla rappresentazione decodificata di una cifra decimale (cioè, una rappresentazione in cui ogni stringa contiene un solo bit alto), fornisce in uscita la rappresentazione Y della cifra mediante codifica Binary-Coded Decimal (BCD).

Input: 0000000001 à Output: 0000 (cifra 0)

Input: 0000000010 à Output: 0001 (cifra 1)

Input: 0000000100 à Output: 0010 (cifra 2)

….

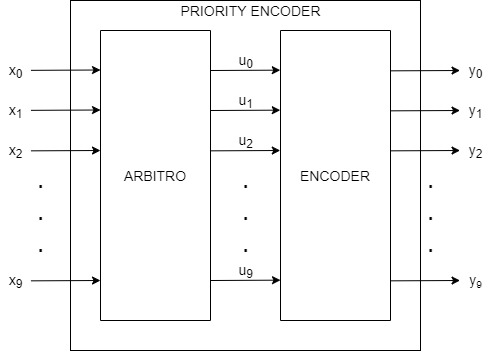
* 1. Sintetizzare ed implementare su board il progetto dell’encoder BCD utilizzando gli switch per fornire la stringa X in ingresso, e i led per visualizzare Y. Nel caso in cui si utilizzi una board dotata di soli 8 switch, è possibile sviluppare il progetto considerando X di soli 8 bit (la macchina sarà allora in grado di fornire in uscita la rappresentazione BCD delle cifre decimali da 0 a 7).
  2. Utilizzare un display a 7 segmenti per visualizzare la cifra decimale codificata da Y (pilotare opportunamente i catodi del display per visualizzare la cifra).

## **2.1 Progettazione del Encoder BCD**

L’encoder è un dispositivo che riceve in ingresso una parola ad n bit e ha come uscita una parola ad m bit, con m<n. L’ingresso è tendenzialmente decodifica mentre l’uscita è codifica; quindi, l’encoder riceve in ingresso n bit di cui sono uno è alto e dà in uscita codificata a bit.

Per la progettazione di un Encoder BCD si utilizza anche in questo caso un approccio modulare. Si compone, cioè, l’Encoder BCD tramite due componenti. Il primo componente è un arbitro di priorità e il secondo è l’encoder vero e proprio. L’arbitro di priorità ha lo scopo di evitare situazioni in cui, per errore o volutamente, in ingresso all’encoder si presentino n bit in cui più di uno sia alto, infatti l’arbitro ha come ingresso una parola di n bit e come uscita una parola n bit (la quale diventerà l’ingresso del decoder) ma, se in ingresso c’è più di un bit alto, allora l’arbitro andrà a considerare come valido solamente il bit più significativo, riportando in uscita quindi una parola di n bit in cui solo quello più significativo sarà alto.

Lo schema del Priority Encoder è mostrato in Figura 2.1.



**Figura 2.1**

Se la board a nostra disposizione non avesse avuto un numero sufficiente di switch su cui mappare i vari ingressi, le soluzioni potevano essere svariate:

* Creare un processo che gestisse gli input in più passi, ad esempio attraverso i bottoni, assegnando ad esempio 10/n segnali alla volta in base agli switch disponibili.
* Biforcare gli ingressi, cioè creare un dispositivo che prende un filo in ingresso e ne costruisce due.
* Precaricare nella memoria un insieme di sequenze.

Così come è stato implementato, il sistema comprende 10 ingressi mappati su altrettanti switch e 4 uscite, mappate prima su altrettanti led e successivamente visualizzabile tramite una singola cifra sul display.

### **2.1.1 Codice VHDL del Encoder BCD**

**CODICE ARBITRO**

Nel codice dell’arbitro, viene definito come deve essere l’output in modo tale che, anche se più di uno degli ingressi è alto, l’uscita avrà comunque solo e soltanto un bit alto. Viene trattato con un modello Dataflow.

entity **Arbiter** is

Port(

X : in STD\_LOGIC\_VECTOR(9 downto 0);

Y : out STD\_LOGIC\_VECTOR(9 downto 0)

);

end **Arbiter**;

architecture **Dataflow** of **Arbiter** is

begin

Y <= "1000000000" when X(9) = '1' else

"0100000000" when X(8) = '1' else

"0010000000" when X(7) = '1' else

"0001000000" when X(6) = '1' else

"0000100000" when X(5) = '1' else

"0000010000" when X(4) = '1' else

"0000001000" when X(3) = '1' else

"0000000100" when X(2) = '1' else

"0000000010" when X(1) = '1' else

"0000000001" when X(0) = '1' else

"----------";

end **Dataflow**;

**CODICE ENCODER**

Nel codice dell’encoder si specifica l’uscita codificata in base all’ingresso decodificato. Anch’esso è trattato col modello Dataflow.

entity **Encoder** is

Port(

X : in STD\_LOGIC\_VECTOR(9 downto 0);

Y : out STD\_LOGIC\_VECTOR(3 downto 0)

);

end **Encoder**;

architecture **Dataflow** of **Encoder** is

begin

with X select

Y <= "0000" when "0000000001",

"0001" when "0000000010",

"0010" when "0000000100",

"0011" when "0000001000",

"0100" when "0000010000",

"0101" when "0000100000",

"0110" when "0001000000",

"0111" when "0010000000",

"1000" when "0100000000",

"1001" when "1000000000",

"----" when others;

end **Dataflow**;

**CODICE PRIORITY ENCODER**

Nel priority encoder invece, si usa un modello strutturale, in quanto si utilizzano i componenti prima specificati (Arbiter e Encoder) utilizzando proprio il costrutto COMPONENT. Si vanno poi a mappare i vari ingressi/uscite dei vari componenti nel modo illustrato nello schema.

entity **PriorityEncoder** is

Port(

X : in STD\_LOGIC\_VECTOR(9 downto 0);

Y : out STD\_LOGIC\_VECTOR(3 downto 0)

);

end **PriorityEncoder**;

architecture **Structural** of **PriorityEncoder** is

COMPONENT Arbiter IS

Port(

X : in STD\_LOGIC\_VECTOR(9 downto 0);

Y : out STD\_LOGIC\_VECTOR(9 downto 0)

);

END COMPONENT;

COMPONENT Encoder IS

Port(

X : in STD\_LOGIC\_VECTOR(9 downto 0);

Y : out STD\_LOGIC\_VECTOR(3 downto 0)

);

END COMPONENT;

signal u : STD\_LOGIC\_VECTOR(9 downto 0);

begin

A : Arbiter

PORT MAP(

X => X,

Y => u

);

E : Encoder

PORT MAP(

X => u,

Y => Y

);

end **Structural**;

### **2.1.2 La simulazione del Encoder BCD**

**CODICE TESTBENCH**

Il codice usato per testare il Priority Encoder è il seguente:

entity PriorityEncoder\_TB is

end PriorityEncoder\_TB;

architecture Behavioral of PriorityEncoder\_TB is

COMPONENT PriorityEncoder

Port(

X : in STD\_LOGIC\_VECTOR(9 downto 0);

Y : out STD\_LOGIC\_VECTOR(3 downto 0)

);

END COMPONENT;

signal input : STD\_LOGIC\_VECTOR(9 downto 0) := "0000000000";

signal output : STD\_LOGIC\_VECTOR(3 downto 0);

begin

uut : PriorityEncoder PORT MAP (

X => input,

Y => output

);

stim\_proc: process

begin

wait for 100 ns;

input <= "1010101010";

wait for 50 ns;

input <= "0101011111";

wait for 50 ns;

input <= "0000101010";

wait for 50 ns;

input <= "0010111111";

wait for 50 ns;

input <= "0001110111";

wait for 50 ns;

input <= "0000000001";

wait for 50 ns;

input <= "0000010101";

wait for 50 ns;

input <= "0000000010";

wait for 50 ns;

input <= "1000000000";

wait for 50 ns;

input <= "0000000100";

wait for 50 ns;

input <= "0000000000";

wait;

end process;

end Behavioral;

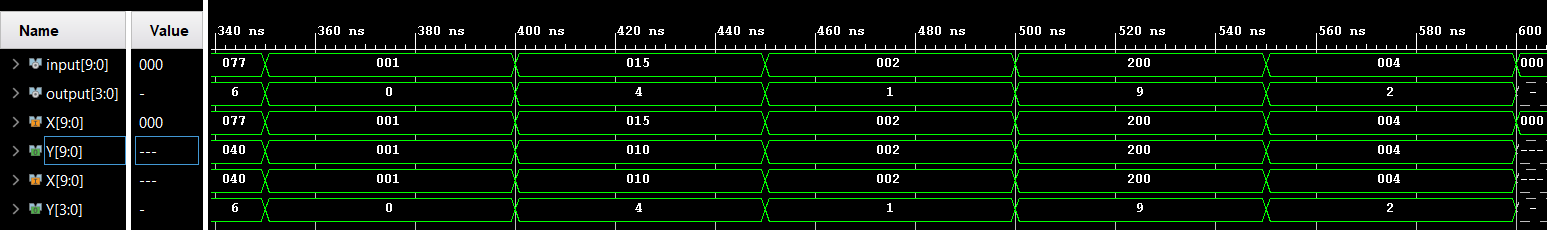
Come si può vedere, sono stati inseriti sia input con un solo bit alto, sia input con più di un bit alto, per testare il funzionamento dell’arbitro.

Il risultato è mostrato in Figura 2.2 nell’intervallo 100-340ns e in Figura 2.3 nell’intervallo 340-600ns:

Immagine che contiene testo, monitor, schermo, screenshot

Descrizione generata automaticamente

**Figura 2.2**



**Figura 2.3**

Dalla simulazione si deve specificare che input[9:0] si riferisce all’input immesso dall’utente e output[3:0] si riferisce all’output dell’intero Priority Encoder, successivamente X[9:0] si riferisce all’input dell’arbitro (che corrisponde all’input dell’utente) e Y[9:0] si riferisce all’output dell’arbitro, che corrisponde al successivo X[9:0] che sarebbe l’input dell’encoder, ed infine c’è Y[3:0] che è l’uscita dell’encoder interno, che corrisponde all’uscita finale del Priority Encoder.

Dopo un’attesa di 100ns, vengono inseriti gli input uno ad uno con intervallo di 50ns.

Il primo input è *2aa,* cioè “1010101010”, cioè il primo input presenta più bit alti. Come si può notare, l’uscita dell’arbitro risulta essere invece *200*, cioè “1000000000”, ovvero va a considerare solo il primo bit più significativo, azzerando tutti i successivi, esattamente come dovrebbe fare. Siccome ad essere alto è il nono bit più significativo, allora in uscita si presenta la configurazione “9”, che in binario sarebbe “1001”.

Gli input successivi confermano la correttezza della macchina, sia che essi siano configurati esattamente, cioè con un solo bit alto, sia in caso contrario.

### **2.1.3 Implementazione sulla board tramite i led**

Per poter implementare l’utilizzo sulla board, bisogna decidere su che switch mappare gli ingressi e su che led mappare le uscite. Si decide arbitrariamente di mappare i 10 ingressi sugli switch da V10 a U18 mentre i le uscite verranno mappate sui led da N14 a H17. Le linee da decommentare e modificare sono:

##Switches

set\_property -dict { PACKAGE\_PIN U18 IOSTANDARD LVCMOS33 } [get\_ports { X[9] }]; #IO\_L17N\_T2\_A13\_D29\_14 Sch=sw[6]

set\_property -dict { PACKAGE\_PIN R13 IOSTANDARD LVCMOS33 } [get\_ports { X[8] }]; #IO\_L5N\_T0\_D07\_14 Sch=sw[7]

set\_property -dict { PACKAGE\_PIN T8 IOSTANDARD LVCMOS18 } [get\_ports { X[7] }]; #IO\_L24N\_T3\_34 Sch=sw[8]

set\_property -dict { PACKAGE\_PIN U8 IOSTANDARD LVCMOS18 } [get\_ports { X[6] }]; #IO\_25\_34 Sch=sw[9]

set\_property -dict { PACKAGE\_PIN R16 IOSTANDARD LVCMOS33 } [get\_ports { X[5] }]; #IO\_L15P\_T2\_DQS\_RDWR\_B\_14 Sch=sw[10]

set\_property -dict { PACKAGE\_PIN T13 IOSTANDARD LVCMOS33 } [get\_ports { X[4] }]; #IO\_L23P\_T3\_A03\_D19\_14 Sch=sw[11]

set\_property -dict { PACKAGE\_PIN H6 IOSTANDARD LVCMOS33 } [get\_ports { X[3] }]; #IO\_L24P\_T3\_35 Sch=sw[12]

set\_property -dict { PACKAGE\_PIN U12 IOSTANDARD LVCMOS33 } [get\_ports { X[2] }]; #IO\_L20P\_T3\_A08\_D24\_14 Sch=sw[13]

set\_property -dict { PACKAGE\_PIN U11 IOSTANDARD LVCMOS33 } [get\_ports { X[1] }]; #IO\_L19N\_T3\_A09\_D25\_VREF\_14 Sch=sw[14]

set\_property -dict { PACKAGE\_PIN V10 IOSTANDARD LVCMOS33 } [get\_ports { X[0] }]; #IO\_L21P\_T3\_DQS\_14 Sch=sw[15]

## LEDs

set\_property -dict { PACKAGE\_PIN H17 IOSTANDARD LVCMOS33 } [get\_ports { Y[0] }]; #IO\_L18P\_T2\_A24\_15 Sch=led[0]

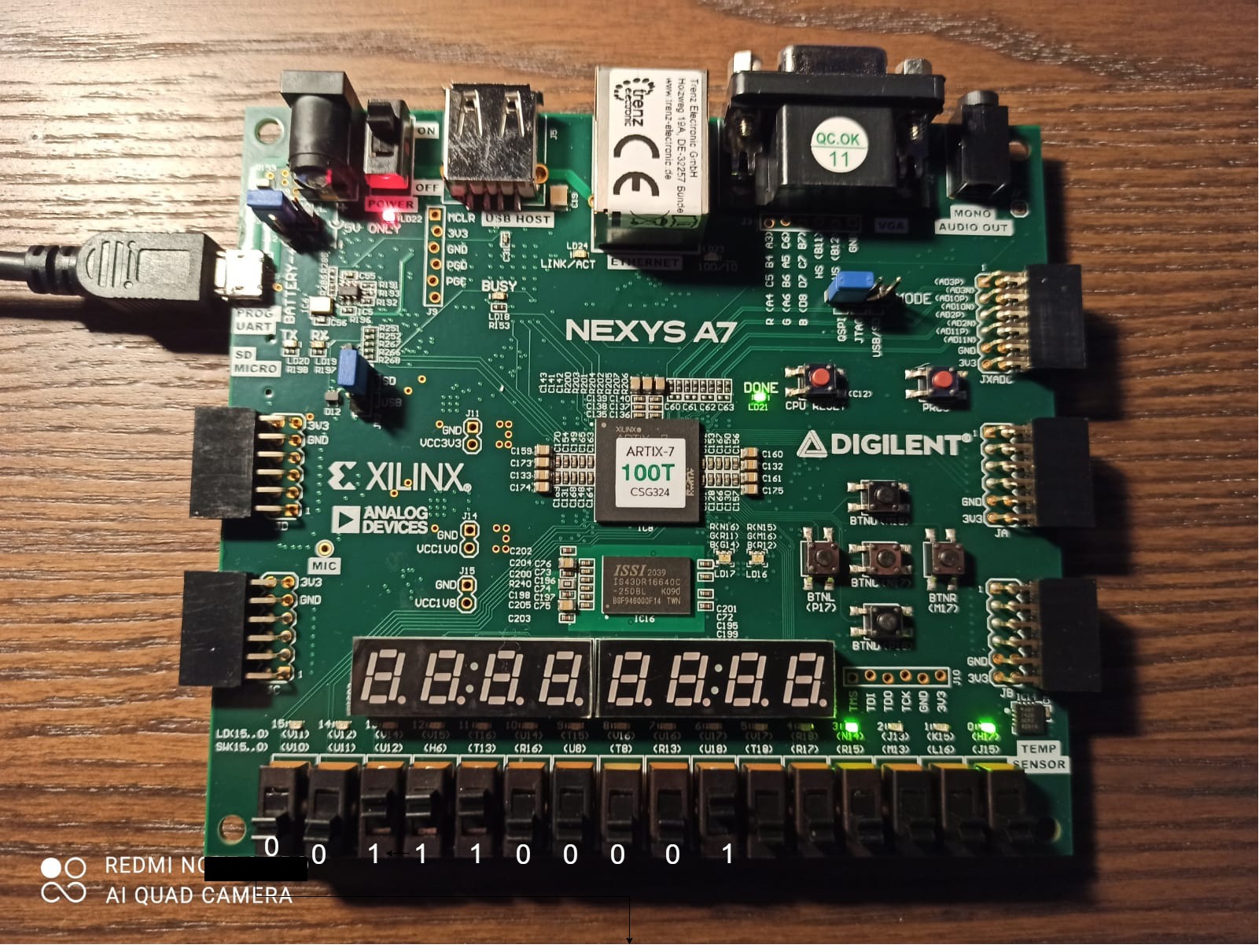
set\_property -dict { PACKAGE\_PIN K15 IOSTANDARD LVCMOS33 } [get\_ports { Y[1] }]; #IO\_L24P\_T3\_RS1\_15 Sch=led[1]

set\_property -dict { PACKAGE\_PIN J13 IOSTANDARD LVCMOS33 } [get\_ports { Y[2] }]; #IO\_L17N\_T2\_A25\_15 Sch=led[2]

set\_property -dict { PACKAGE\_PIN N14 IOSTANDARD LVCMOS33 } [get\_ports { Y[3] }]; #IO\_L8P\_T1\_D11\_14 Sch=led[3]

Come esempio, riportiamo quelli di Figura 2.4 e 2.5. In particolare, possiamo vedere in Figura 2.4 che la configurazione di ingresso risulta avere solo un bit alto, cioè quello più significativo mappato sullo switch U18. La configurazione di ingresso è quindi 1000000000. In questo caso non è necessario l’intervento dell’arbitro e l’uscita risulta corretta, 1001.

Per vedere se l’arbitro funziona o meno, vediamo la Figura 2.5, in cui, mantenendo U18 alzato, sono stati alzati anche gli switches U12, H6, T13. Notiamo che la configurazione dell’uscita non varia; ciò vuol dire che l’arbitro considera solo il bit più significativo se in ingresso ci sono più bit alti. Il Priority Encoder funzionano correttamente.



**Figura 2.4**

Immagine che contiene testo, elettronico, circuito

Descrizione generata automaticamente

**Figura 2.5**

### **2.1.4 Implementazione sulla board tramite il display**

Per poter implementare l’utilizzo del display sulla board, è necessario modificare la composizione del progetto e aggiungere una nuova entità, denominata proprio *display*. Ad ogni numero in ingresso si associa la giusta configurazione degli anodi e dei catodi.

Nota: il seguente metodo di utilizzo del display va bene esclusivamente se si deve utilizzare una solo cifra. Se il progetto avesse preveduto l’utilizzo di più cifre, l’utilizzo del display avrebbe dovuto essere gestito diversamente.

entity display is

Port (

number: in STD\_LOGIC\_VECTOR(3 downto 0);

anodi: out STD\_LOGIC\_VECTOR(7 downto 0);

catodi: out STD\_LOGIC\_VECTOR(7 downto 0)

);

end display;

architecture Behavioral of display is

begin

process (number)

begin

case number is

when "0000" =>

catodi <= "11000000"; --0

anodi <= "11111110";

when "0001" =>

catodi <= "11111001"; --1

anodi <= "11111110";

when "0010" =>

catodi <= "10100100"; --2

anodi <= "11111110";

when "0011" =>

catodi <= "10110000"; --3

anodi <= "11111110";

when "0100" =>

catodi <= "10011001"; --4

anodi <= "11111110";

when "0101" =>

catodi <= "10010010"; --5

anodi <= "11111110";

when "0110" =>

catodi <= "10000010"; --6

anodi <= "11111110";

when "0111" =>

catodi <= "11111000"; --7

anodi <= "11111110";

when "1000" =>

catodi <= "10000000"; --8

anodi <= "11111110";

when "1001" =>

catodi <= "10010000"; --9

anodi <= "11111110";

when "1010" =>

catodi <= "10100000"; --a

anodi <= "11111110";

when "1011" =>

catodi <= "10000011"; --b

anodi <= "11111110";

when "1100" =>

catodi <= "11000110"; --c

anodi <= "11111110";

when "1101" =>

catodi <= "10100001"; --d

anodi <= "11111110";

when "1110" =>

catodi <= "10000110"; --e

anodi <= "11111110";

when "1111" =>

catodi <= "10001110"; --f

anodi <= "11111110";

when others =>

catodi <= "11000000"; --x

anodi <= "11111101";

end case;

end process;

end Behavioral;

Altra modifica che bisogna fare, riguarda le porte di ingresso dell’entita *PriorityEncoder*, che devono essere aggiornate aggiungendo un vettore per i catodi e uno per gli anodi.

entity PriorityEncoder is

Port(

X : in STD\_LOGIC\_VECTOR(9 downto 0);

CAT : out STD\_LOGIC\_VECTOR(7 downto 0);

AN : out STD\_LOGIC\_VECTOR(7 downto 0)

);

end PriorityEncoder;

Ovviamente, l’entità *display* deve essere aggiunta come componente all’interno dell’architecture del Priority Encoder, con le seguenti modifiche al comportamento:

signal u : STD\_LOGIC\_VECTOR(9 downto 0);

signal uu : STD\_LOGIC\_VECTOR(3 downto 0);

begin

A : Arbiter

PORT MAP(

X => X,

Y => u

);

E : Encoder

PORT MAP(

X => u,

Y => uu

);

D : display

PORT MAP(

number => uu,

anodi => AN,

catodi => CAT

);

Infine, bisogna modificare il file *.xdc* commentando le righe inerenti all’utilizzo dei led e decommentando ed associando correttamente quelle relative al display.

##7 segment display

set\_property -dict { PACKAGE\_PIN T10 IOSTANDARD LVCMOS33 } [get\_ports { CAT[0] }]; #IO\_L24N\_T3\_A00\_D16\_14 Sch=ca

set\_property -dict { PACKAGE\_PIN R10 IOSTANDARD LVCMOS33 } [get\_ports { CAT[1] }]; #IO\_25\_14 Sch=cb

set\_property -dict { PACKAGE\_PIN K16 IOSTANDARD LVCMOS33 } [get\_ports { CAT[2] }]; #IO\_25\_15 Sch=cc

set\_property -dict { PACKAGE\_PIN K13 IOSTANDARD LVCMOS33 } [get\_ports { CAT[3] }]; #IO\_L17P\_T2\_A26\_15 Sch=cd

set\_property -dict { PACKAGE\_PIN P15 IOSTANDARD LVCMOS33 } [get\_ports { CAT[4] }]; #IO\_L13P\_T2\_MRCC\_14 Sch=ce

set\_property -dict { PACKAGE\_PIN T11 IOSTANDARD LVCMOS33 } [get\_ports { CAT[5] }]; #IO\_L19P\_T3\_A10\_D26\_14 Sch=cf

set\_property -dict { PACKAGE\_PIN L18 IOSTANDARD LVCMOS33 } [get\_ports { CAT[6] }]; #IO\_L4P\_T0\_D04\_14 Sch=cg

set\_property -dict { PACKAGE\_PIN H15 IOSTANDARD LVCMOS33 } [get\_ports { CAT[7] }]; #IO\_L19N\_T3\_A21\_VREF\_15 Sch=dp

set\_property -dict { PACKAGE\_PIN J17 IOSTANDARD LVCMOS33 } [get\_ports { AN[0] }]; #IO\_L23P\_T3\_FOE\_B\_15 Sch=an[0]

set\_property -dict { PACKAGE\_PIN J18 IOSTANDARD LVCMOS33 } [get\_ports { AN[1] }]; #IO\_L23N\_T3\_FWE\_B\_15 Sch=an[1]

set\_property -dict { PACKAGE\_PIN T9 IOSTANDARD LVCMOS33 } [get\_ports { AN[2] }]; #IO\_L24P\_T3\_A01\_D17\_14 Sch=an[2]

set\_property -dict { PACKAGE\_PIN J14 IOSTANDARD LVCMOS33 } [get\_ports { AN[3] }]; #IO\_L19P\_T3\_A22\_15 Sch=an[3]

set\_property -dict { PACKAGE\_PIN P14 IOSTANDARD LVCMOS33 } [get\_ports { AN[4] }]; #IO\_L8N\_T1\_D12\_14 Sch=an[4]

set\_property -dict { PACKAGE\_PIN T14 IOSTANDARD LVCMOS33 } [get\_ports { AN[5] }]; #IO\_L14P\_T2\_SRCC\_14 Sch=an[5]

set\_property -dict { PACKAGE\_PIN K2 IOSTANDARD LVCMOS33 } [get\_ports { AN[6] }]; #IO\_L23P\_T3\_35 Sch=an[6]

set\_property -dict { PACKAGE\_PIN U13 IOSTANDARD LVCMOS33 } [get\_ports { AN[7] }]; #IO\_L23N\_T3\_A02\_D18\_14 Sch=an[7]

Immagine che contiene testo, elettronico

Descrizione generata automaticamente

Riportiamo un esempio dell’utilizzo del display, in cui è possibile vedere che il decimo switch è alto, infatti il display visualizza il numero 9, e continua a visualizzare 9 anche se sono alti altri switch di peso inferiore; ciò significa che l’arbitro funziona bene.

**2.1.5 C’è un’altra soluzione? Perché queste scelte? Sviluppo futuro**

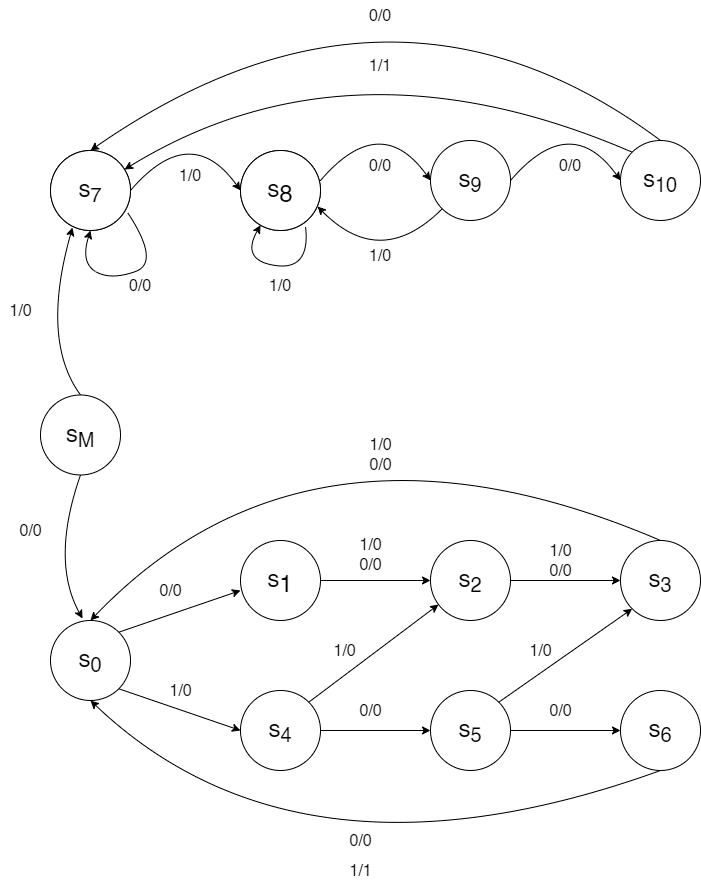
# **3 Riconoscitore a 2 *modalità***

Si realizza un riconoscitore di sequenza a 2 modalità seguendo le specifiche richieste dal cliente:

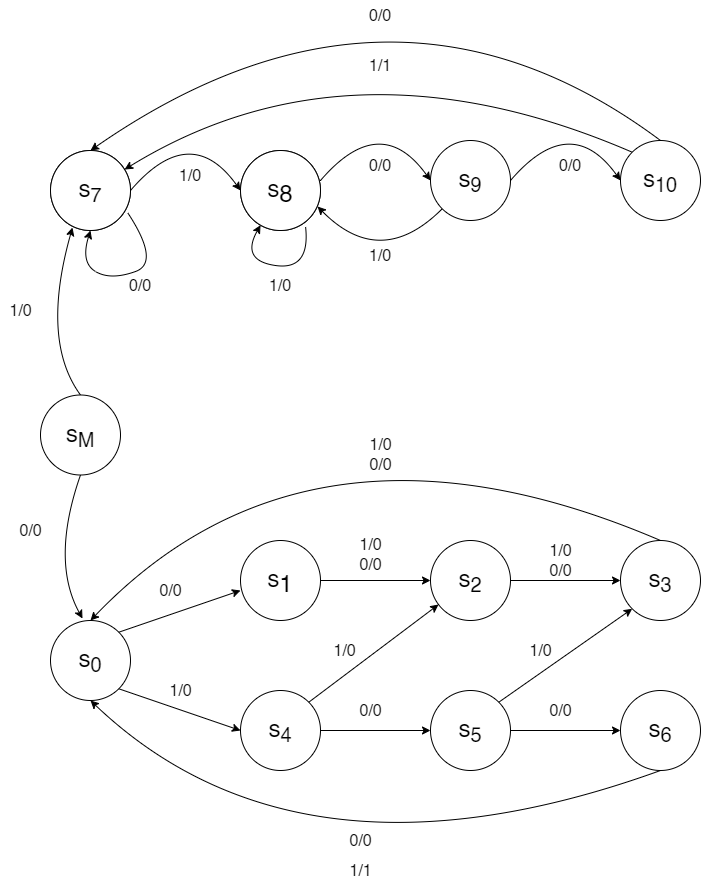
* **3.1** - Progettare, implementare in VHDL e testare mediante **simulazione** una macchina in grado di riconoscere la sequenza **1001**. La macchina prende in ingresso un segnale binario i che rappresenta il dato, un segnale A di tempificazione e un segnale M di modo, che ne disciplina il funzionamento, e fornisce un’uscita Y alta quando la sequenza viene riconosciuta. In particolare,
* se **M=0**, la macchina valuta i bit seriali in ingresso a gruppi di 4,
* se **M=1**, la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta.
* **3.2** - Sintetizzare e implementare su board la rete sviluppata al punto precedente, utilizzando uno switch S1 per codificare l’input i e uno switch S2 per codificare il modo M, in combinazione con due bottoni B1 e B2 utilizzati rispettivamente per acquisire l’input da S1 e S2 in sincronismo con il segnale di tempificazione A, che deve essere ottenuto a partire dal clock del board. Infine, l’uscita Y può essere codificata utilizzando un led.

## **3.1 Progettazione del Riconoscitore di sequenza a 2 modalità**

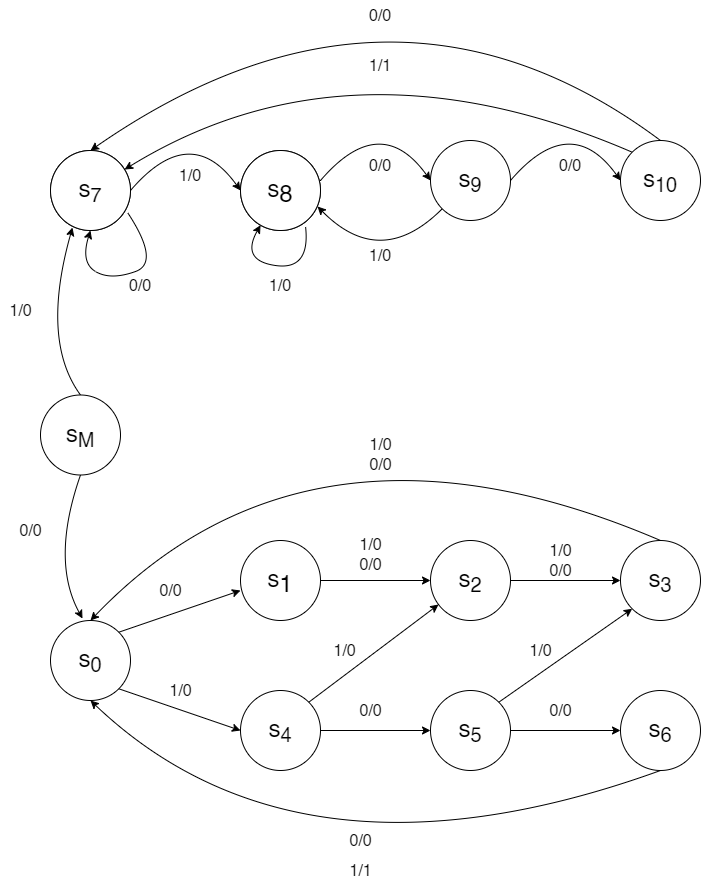
Per la realizzazione della macchina, si utilizza **un approccio comportamentale** dove viene descritto il comportamento della macchina come definito dai relativi automi. Le specifiche del progetto richiedono la valutazione dei bit in “2 modi” determinato dal segnale M di modo fornendo un’uscita Y alta quando la sequenza viene riconosciuta.   
Dalle indicazioni delle specifiche, la sequenza da riconoscere è la **1001**, pertanto si costruiscono le due macchine sequenziali:



**FIG. 1** - Automa che valuta i bit seriali in ingresso uno alla volta



**FIG. 2** - Automa che valuta i bit seriali in ingresso a gruppi di 4 bit.



**FIG. 3** - Automa con modo

Dalla **FIG.3** si evince dunque il funzionamento del riconoscitore. Partendo dallo stato iniziale **SM**, lo stato che determina il modo dell’automa, se M = 1, si valuta la sequenza **un** bit alla volta, se M = 0 si valuta la sequenza **a gruppi di 4 bit.**

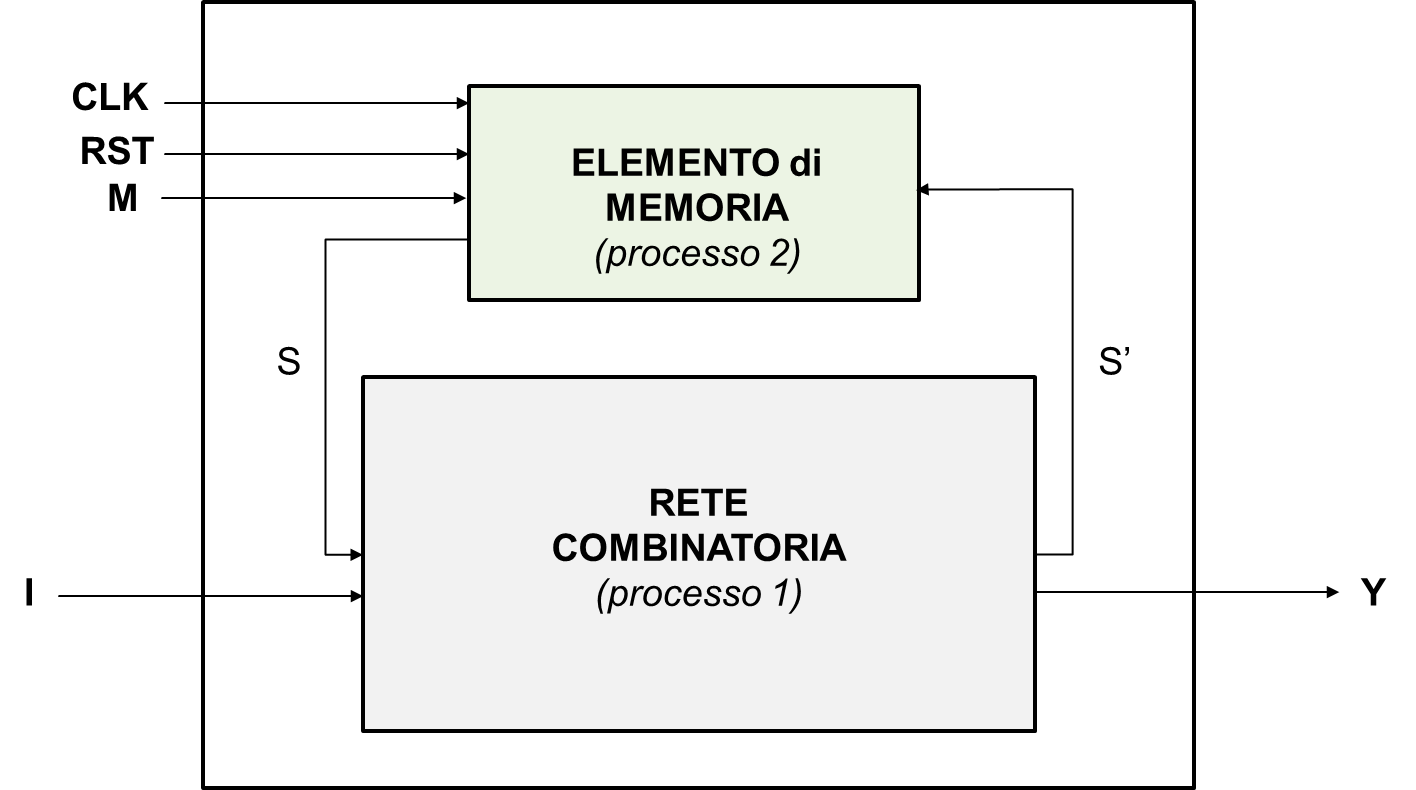
Dalle **FIGURE** si nota inoltre che è stato utilizzato il modello di **Mealy** dove la variazione dell’uscita a fronte di ingresso e di stato è:

(SERVE SCRIVERE ALTRO?)

### **3.1.1 Schematico del Riconoscitore di sequenza a 2 modalità**

Il modello del seguente riconoscitore è diviso da una rete combinatoria (la parte del sistema che calcolerà le funzioni di transizione da uno stato all’altro) e l’elemento di memoria (la retroazione) utile per la gestione SINCRONA, tramite RST (Reset) di M (Modo) tempificato dal CLK (Clock).

(SERVE SCRIVERE ALTRO?)



**FIG. 4** – Schematico del riconoscitore di sequenza

### **3.1.2 Codice VHDL del Riconoscitore di sequenza a 2 modalità**

Dallo schematico in **FIGURA 4**, si estrapolano i “process” da implementare in VHDL, nel caso in esame si sono utilizzati dunque due processi.

Si realizzano dunque due implementazioni: Design Sources per l’implementazione dei processi e un Simulation Sources per il TESTBENCH.

Si definisce **l’entità** composto da 4 porte in ingresso (I, RST, M, CLK) e l’uscita (Y).

**-- L’entità**

entity **Riconoscitore\_Mealy\_2\_Modi\_01** is

port( i: in std\_logic;

RST, CLK: in std\_logic;

M: in std\_logic;

Y: out std\_logic

);

end **Riconoscitore\_Mealy\_2\_Modi\_01**;

Nell’**architecture** si enumerano i 10 diversi stati che compongono entrambi gli automi e si pone come stato iniziale, come già detto, lo stato SM.

-- architettura

architecture **Behavioral** of **Riconoscitore\_Mealy\_2\_Modi\_01** is

-- enumerazione stati

type **stato** is **(SM, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10)**;

-- stato iniziale SM

signal **stato\_corrente** : stato := **SM**;

signal **stato\_prossimo** : stato;

Il processo **combinatorio** sarà così costituito:

begin

stato\_uscita: **process**(stato\_corrente, i)

begin

-- se M = 0 : Riconoscitore di Sequenza con valutazione di sequenza per gruppi di 4 bit

-- se M = 1 : Riconoscitore di Sequenza con valutazione di sequenza ad ogni singolo bit

case **stato\_corrente** is

when **S0** =>

if(i = '0' ) then

**stato\_prossimo** <= S1;

Y <= '0';

else

**stato\_prossimo** <= S4;

Y <= '0';

end if;

when **S1** =>

if( i = '0' or i = '1' ) then

**stato\_prossimo** <= S2;

Y <= '0';

end if;

when **S2** =>

if( i = '0' or i = '1' ) then

**stato\_prossimo** <= S3;

Y <= '0';

end if;

when **S3** =>

if( i = '0' or i = '1' ) then

**stato\_prossimo** <= S0;

Y <= '0';

end if;

when **S4** =>

if( i = '0' ) then

**stato\_prossimo** <= S5;

Y <= '0';

else

**stato\_prossimo** <= S2;

Y <= '0';

end if;

when **S5** =>

if( i = '0' ) then

**stato\_prossimo** <= S6;

Y <= '0';

else

**stato\_prossimo** <= S3;

Y <= '0';

end if;

when **S6** =>

if( i = '0' ) then

**stato\_prossimo** <= S0;

Y <= '0';

else

**stato\_prossimo** <= S0;

Y <= '1';

end if;

when **S7** =>

if( i = '0' ) then

**stato\_prossimo** <= S7;

Y <= '0';

else

**stato\_prossimo** <= S8;

Y <= '0';

end if;

when **S8** =>

if( i = '0' ) then

**stato\_prossimo** <= S9;

Y <= '0';

else

**stato\_prossimo** <= S8;

Y <= '0';

end if;

when **S9** =>

if( i = '0' ) then

**stato\_prossimo** <= S10;

Y <= '0';

else

**stato\_prossimo** <= S8;

Y <= '0';

end if;

when **S10** =>

if( i = '0' ) then

**stato\_prossimo** <= S7;

Y <= '0';

else

**stato\_prossimo** <= S7;

Y <= '1';

end if;

when **others** =>

**stato\_prossimo** <= S7;

Y <= '0';

end **case**;

end **process**;

Si è usato il costrutto **CASE-WHEN** per definire cosa fa la macchina sequenziale in ogni stato. Ogni valore dell’ingresso ‘I’ fa evolvere la macchina verso un nuovo stato specificando dunque una determinata uscita Y.

Il processo **di memorizzazione** sarà:

mem: **process** (CLK)

begin

if(CLK'**event** and CLK = '1') then

if(RST = '1') then

**stato\_corrente** <= SM;

if(M = '0') then

**stato\_corrente** <= S0;

else

**stato\_corrente** <= S7;

end if;

else

**stato\_corrente** <= **stato\_prossimo**;

end **if**;

end **if**;

end **process**;

end **Behavioral**;

L’ istruzione consente di cambiare la modalità del riconoscitore soltanto quando il **RST** è abilitato a **1**. Se M = 0, si imposta lo stato corrente del riconoscitore a 4 bit, se M = 1, si imposta lo stato corrente del riconoscitore a 1 bit.​

L’ assegnazione **stato\_corrente** <= **stato\_prossimo**; consente di retroazionare lo stato prossimo nello stato corrente​

(SERVE ALTRO?)

Il TB ci permette di simulare il funzionamento della macchina in particolare si può apprezzare il comportamento del Riconoscitore nei casi in cui M = 0 o 1 e come le abilitazioni del RST influiscano sul cambio di modalità della macchina.

entity **Riconoscitore\_Mealy\_2\_Modi\_01\_TB** is

-- Port ( );

end **Riconoscitore\_Mealy\_2\_Modi\_01\_TB**;

architecture Behavioral of **Riconoscitore\_Mealy\_2\_Modi\_01\_TB** is

COMPONENT **Riconoscitore\_Mealy\_2\_Modi\_01\_TB**

PORT(

i : IN std\_logic;

CLK,RST,M : IN std\_logic;

Y : OUT std\_logic

);

END **COMPONENT**;

-- Inputs

signal i : std\_logic := '0';

signal CLK : std\_logic := '0';

signal RST : std\_logic := '0';

signal M : std\_logic := '1';

--Outputs

signal Y : std\_logic;

-- Clock period definitions

constant CLK\_period : time := 10 ns;

begin

-- Instantiate the Unit Under Test (UUT)

-- Qui si specifica quale architecture simulare di quelle definite nel progetto corrente

uut: entity work.**Riconoscitore\_Mealy\_2\_Modi\_01**(Behavioral) port map(

i => i,

CLK => CLK,

RST => RST,

M => M,

Y => Y

);

-- Clock process definitions

CLK\_process :process

begin

CLK <= '0';

wait for CLK\_period/2;

CLK <= '1';

wait for CLK\_period/2;

end process;

-- Stimulus process

stim\_proc: process

begin

-- hold reset state for 100 ns.

wait for 100 ns;

-- start da 200 (100 + CLK\_period\*10)

wait for CLK\_period\*10;

-- insert stimulus here

-- sequenza di bit in ingresso 0010010000101001

i<='0';

wait for 10 ns;

i<='0';

wait for 10 ns;

i<='1';

wait for 10 ns;

i<='0';

wait for 10 ns;

i<='0';

wait for 10 ns;

i<='1';

wait for 10 ns;

i<='0';

wait for 10 ns;

--C'è il cambio di M e viene abilitato il reset (RST<='1')

M<='0';

RST<='1';

wait for 10 ns;

RST<='0';

i<='0';

wait for 10 ns;

i<='0';

wait for 10 ns;

i<='1';

wait for 10 ns;

i<='0';

wait for 10 ns;

i<='1';

wait for 10 ns;

i<='0';

wait for 10 ns;

i<='0';

wait for 10 ns;

i<='1';

wait for 10 ns;

-- C'è il cambio di M e viene abilitato il reset (RST<='1')

M<='1';

RST<='1';

wait for 10 ns;

RST<='0';

i<='0';

wait for 10 ns;

i<='0';

wait for 10 ns;

i<='1';

wait for 10 ns;

i<='0';

wait for 10 ns;

i<='1';

wait for 10 ns;

i<='0';

wait for 10 ns;

i<='0';

wait for 10 ns;

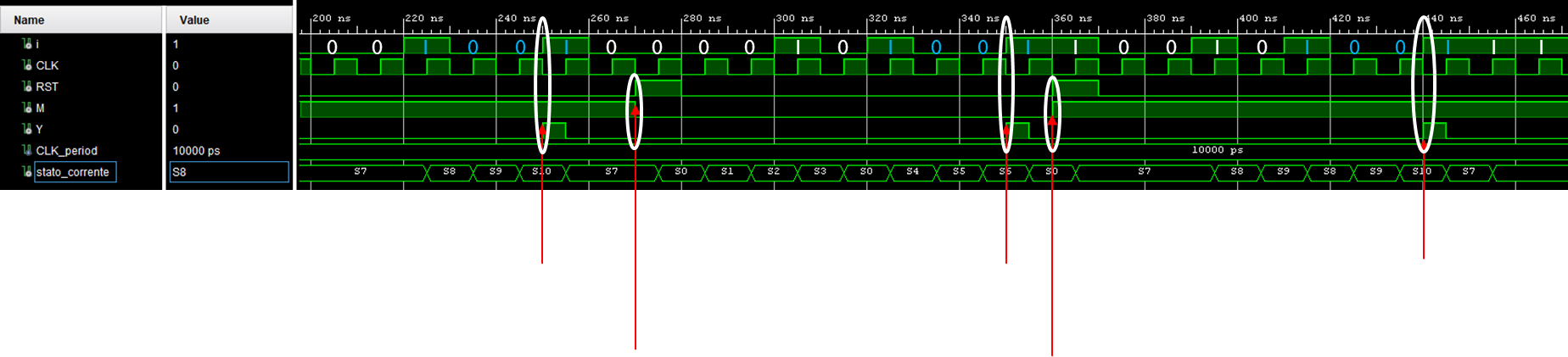
i<='1';

wait;

end **process**;

end **Behavioral**;

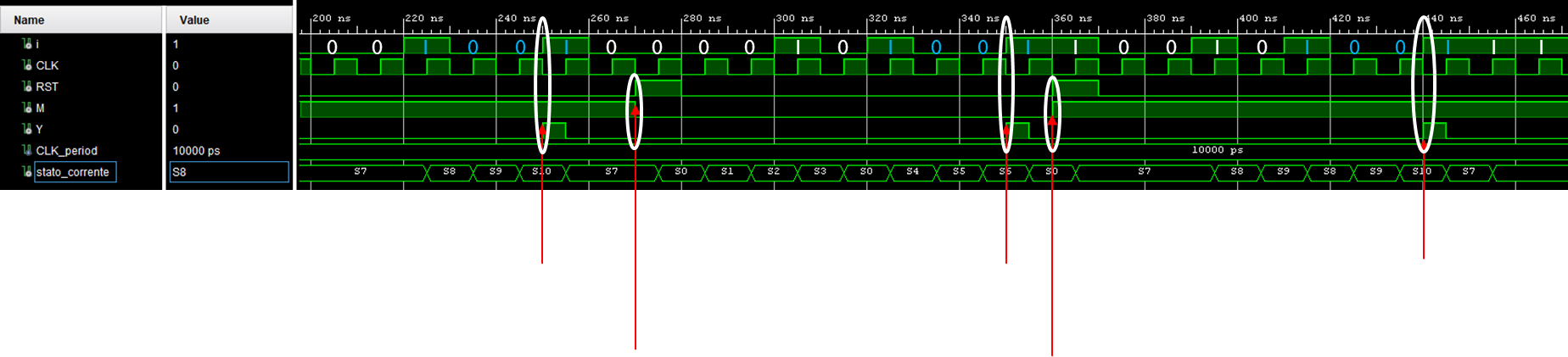
### **3.1.3 La simulazione del Riconoscitore di sequenza a 2 modalità**

Seguendo il TESTBENCH, la simulazione sarà:

A ***270ns*** si ha la transizione di M: 1 à 0 e RST: 0 à 1

(RST torna a 0 dopo 10ns)

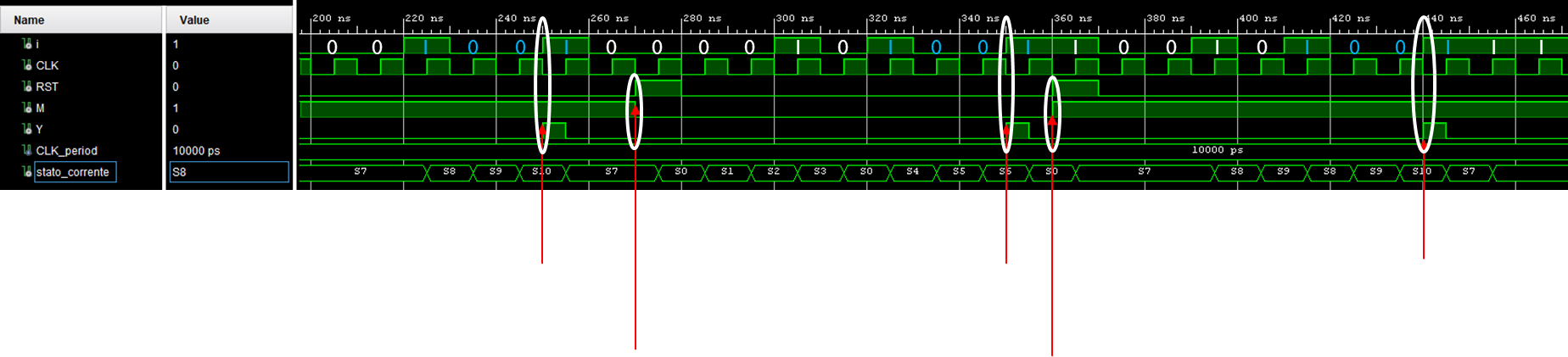
A ***250ns*** si ha la transizione di Y: 0 à 1 (M=1)



A ***350ns*** si ha la transizione di Y: 0 à 1 (M=0)

A ***360ns*** si ha la transizione di M: 0 à 1 e RST: 0 à 1

(RST torna a 0 dopo 10ns)



A ***440ns*** si ha la transizione di Y: 0 à 1 (M=1)

VEDERE LA QUESTIONE DEGLI ASSERT