



UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato in **Architettura dei Sistemi Digitali**

Elaborato finale

Anno Accademico 2024/25

Studenti

Filomena Vigliotti

matr. M63001734

Ciro Scognamilgio

matr. M6300

Antonio Sirignano

matr. M63001732

Indice

1 Esercizio 1	1
1.1 Multiplexer 16:1	1
1.1.1 Progetto e architettura	3
1.1.2 Implementazione	8
1.1.3 Simulazione	12
1.1.4 Implementazione 2.0	15
1.2 Rete di interconnessione a 16 ingressi e 4 uscite . . .	19
1.2.1 Progettazione	20
1.2.2 Implementazione	23
1.2.3 Simulazione	29
1.3 Implementazione su board del punto precedente . . .	34
1.3.1 Traccia	35
1.3.2 Implementazione	36
1.3.3 Funzionamento	40
2 Esercizio 2 - Sistema ROM+M	42
2.1 Progettazione	43
2.2 Implementazione	43
2.3 Simulazione	47

2.4	Implementazione su board	50
2.4.1	Traccia	50
2.4.2	Implementazione	50
3	Esercizio 3	53
3.1	Riconoscitore di sequenze	53
3.1.1	Progettazione e architettura	54
3.1.2	Implementazione	55
3.1.3	Simulazione	58
3.2	Implementazione su board del punto precedente . . .	62
4	Esercizio 4	63
4.1	Shift Register - Approccio comportamentale	63
4.1.1	Progetto e architettura	63
4.1.2	Implementazione	65
4.1.3	Simulazione	66
4.2	Shift Register - Approccio strutturale	69
4.2.1	Progetto e architettura	69
4.2.2	Implementazione	72
4.2.3	Simulazione	77
5	Esercizio 5	81
5.1	Cronometro	81
5.2	Implementazione su board del punto precedente . . .	81
6	Esercizio 6	82
6.1	Sistema di lettura - elaborazione - scrittura PO_PC .	82
6.1.1	Traccia	82

6.1.2	Progettazione	83
6.1.3	Implementazione	85
6.1.4	Simulazione	92
6.2	Implementazione su board del punto precedente . . .	95
6.2.1	Traccia	95
	Bibliografia	97

Capitolo 1

Esercizio 1

1.1 Multiplexer 16:1

Un multiplexer è una **macchina combinatoria**, ovvero una macchina la cui uscita in un determinato istante di tempo dipende solo dall'ingresso nel medesimo istante, e quindi realizza una funzione del tipo:

$$U = f(I)$$

dove I e U rappresentano rispettivamente gli insiemi limitati dei valori di ingresso e di uscita.

Il Multiplexer realizza una connessione $n:1$, ovvero connette n sorgenti a un'unica destinazione sulla base di segnali di selezione.

Un **Multiplexer lineare** è composto da n segnali in ingresso e n segnali di selezione. Tale dispositivo convoglia uno specifico segnale in ingresso verso l'uscita solo se il corrispondente segnale di selezione

è alto. Uno svantaggio di un dispositivo di questo tipo è il numero eccessivo di fili per i segnali di selezione. Per risolvere ciò si può aggiungere un **Decoder**, un altro dispositivo notevole, che riceve in ingresso una parola codice di n bit e presenta in uscita la sua rappresentazione decodificata di 2^n bit.

Unendo un Multiplexer lineare a un Decoder, l'architettura diventa quella in figura, e si ottiene un componente definito **Multiplexer indirizzabile**, che diversamente da quello lineare, prende solo 2 segnali di selezione in ingresso. Un MUX indirizzabile è a sua volta una macchina notevole, caratterizzata da 2^n ingressi, n ingressi di selezione e un'unica uscita.

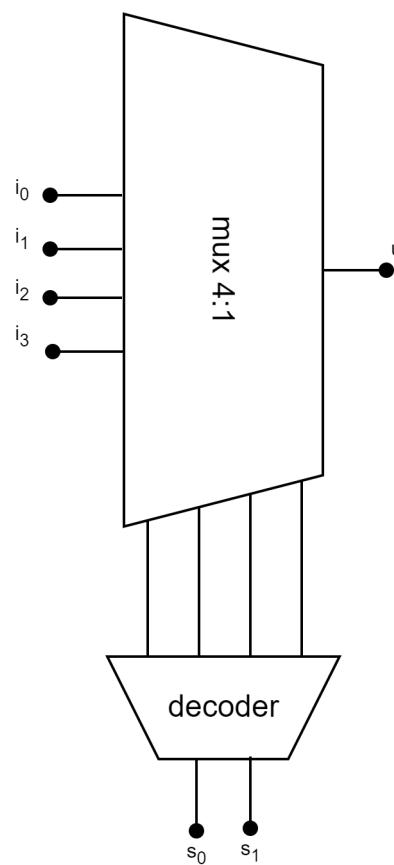


Figura 1.1: multiplexer indirizzabile

Si vuole ora progettare un multiplexer indirizzabile 16:1, utilizzando un approccio per composizione, a partire da multiplexer 4:1. Tale multiplexer è rappresentato di seguito.

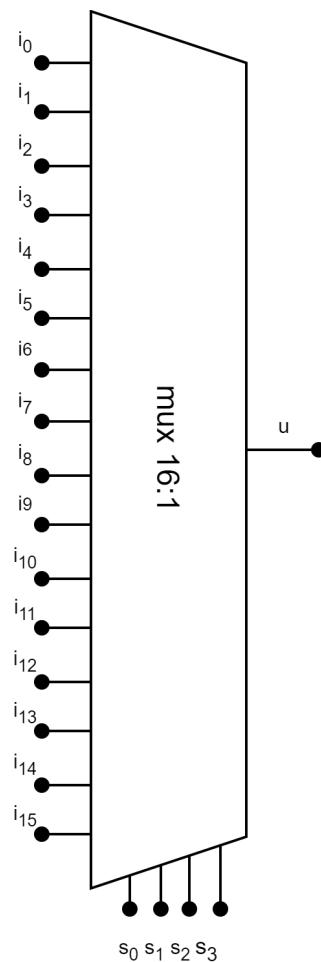


Figura 1.2: multiplexer 16:1

1.1.1 Progetto e architettura

Dapprima si utilizza un approccio per composizione per realizzare un multiplexer 4:1 con multiplexer 2:1.

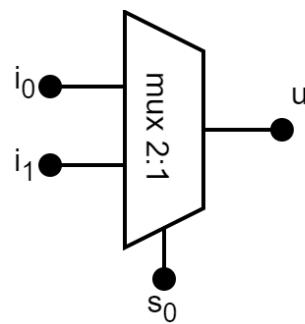


Figura 1.3: multiplexer 2:1

Il primo componente che si realizza è un multiplexer 2:1, caratterizzato dalla seguente tabella di verità:

s₀	i₁	i₀	u
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Tabella 1.1: Tabella di verità di un Mux 2:1

da cui si ottiene l'equazione:

$$u = (i_0 \text{ AND } \bar{s}_0) \text{ OR } (i_1 \text{ AND } s_0)$$

Il successivo componente da costruire è un multiplexer 4:1.

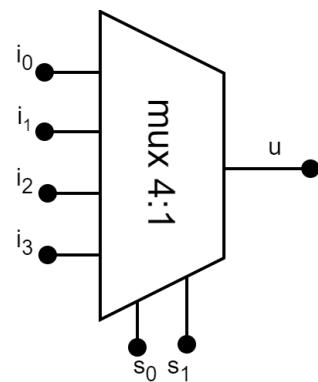


Figura 1.4: multiplexer 4:1

Per composizione, a partire da 3 multiplexer 2:1, si può ottenere un multiplexer 4:1

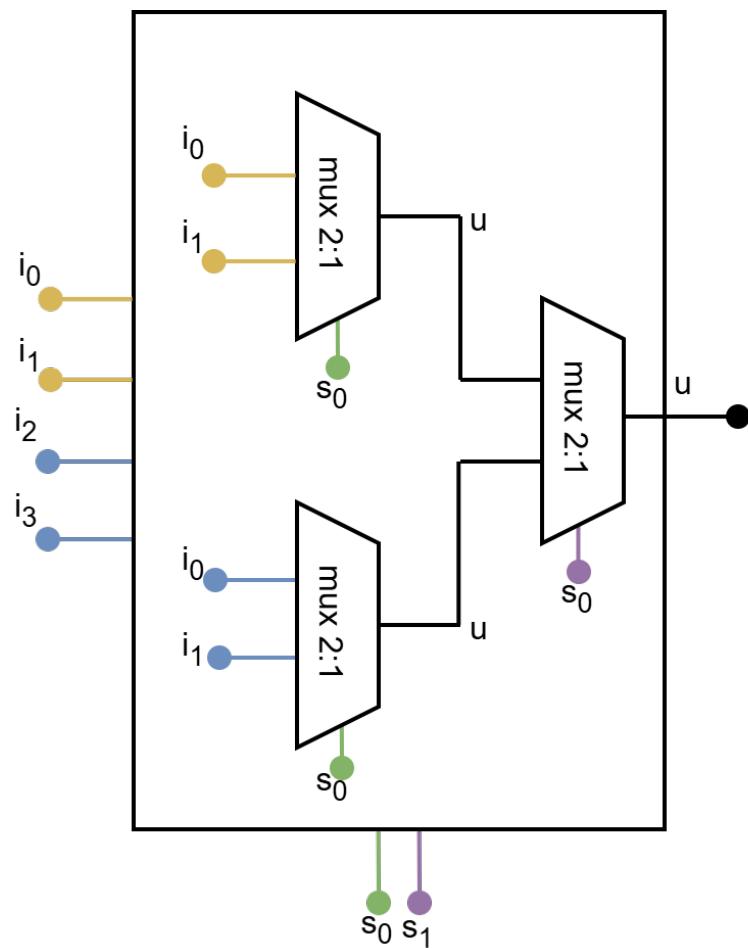


Figura 1.5: multiplexer 4:1 per composizione di multiplexer 2:1

I 4 ingressi entrano in due multiplexer 2:1, che prendono due ingressi e producono un'uscita ciascuno; tali uscite vengono immesse nel terzo multiplexer, che produrrà l'unico output finale. Concettualmente, si divide la selezione in ingresso al multiplexer esterno in due parti:

- la parte meno significativa (indicata dal colore verde) s_0 , viene posta in ingresso ai multiplexer del primo stadio e seleziona per ciascuno un filo in uscita;
- la parte più significativa (indicata dal colore viola) s_1 entra nel multiplexer del secondo stadio e decide quale dei due fili, provenienti dai due blocchi precedenti, sarà immessa in uscita.

In maniera analoga si procede con la progettazione del multiplexer 16:1.

Anche in questo caso, sono stati usati dei colori per identificare i collegamenti tra le componenti.

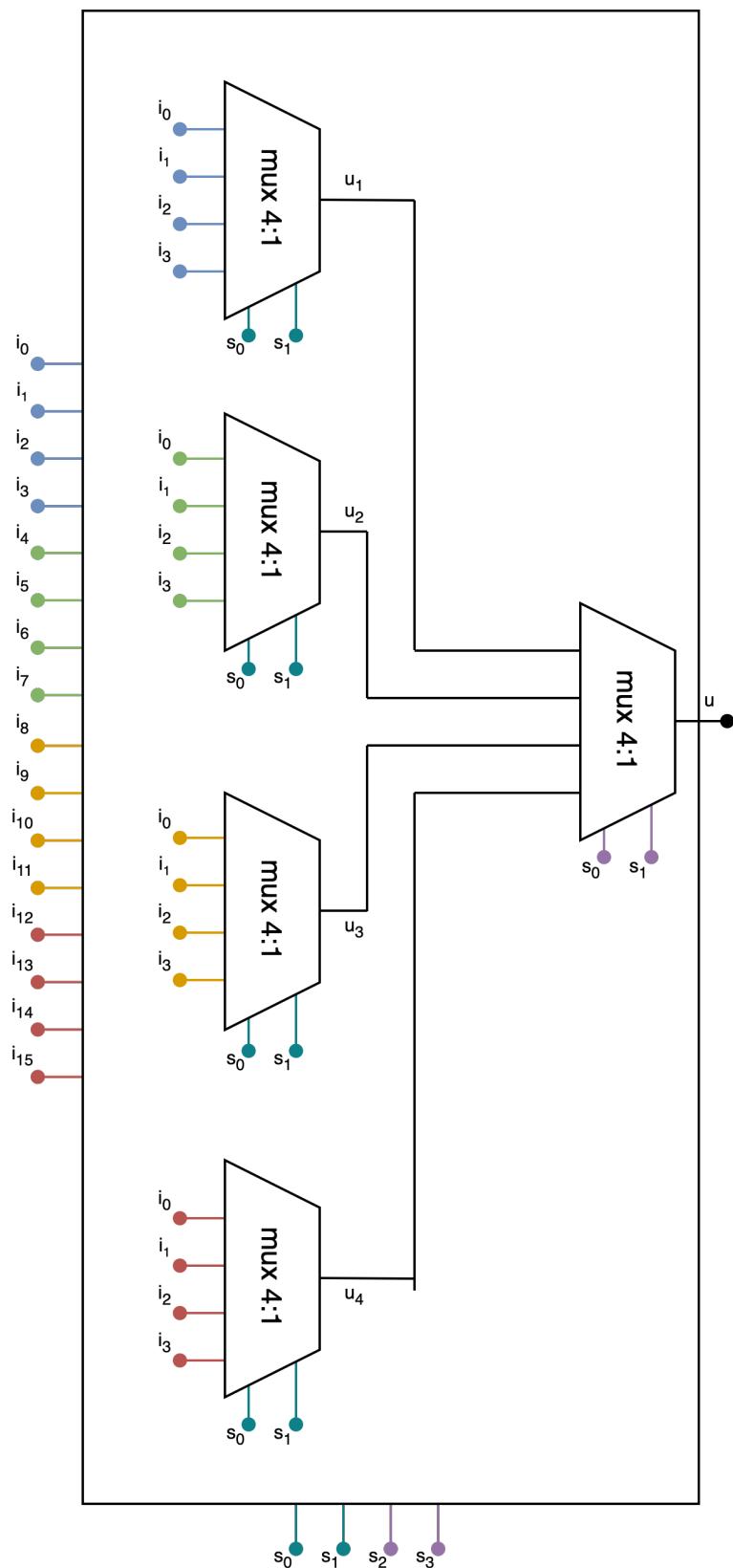


Figura 1.6: multiplexer 16:1 per composizione di multiplexer 4:1

1.1.2 Implementazione

Per l'implementazione si procede con un approccio di tipo strutturale, iniziando quindi dalla codifica del multiplexer 2:1, e, a partire da questo si compongono dispositivi sempre più complessi fino ad arrivare all'obiettivo del multiplexer 16:1.

Mux 2:1 Di seguito il codice riguardante il Mux 2:1.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.all;
3
4 entity mux_21 is
5     port
6         (
7             i0, i1: in std_logic;
8             s0: in std_logic;
9             u: out std_logic
10        );
11 end mux_21;
12
13 architecture rtl of mux_21 is
14 begin
15     u <= i0 when s0='0' else
16         i1 when s0='1' else
17         ' ';
18 end rtl;
```

Code 1.1: Multiplexer 2:1 in VHDL

L'interfaccia del componente ha come ingressi `i0` ed `i1`, come selezione `s0` e come uscita `u`.

Al seguito della definizione dell'interfaccia, si definisce il comportamento dell'entità, che risponde alla tabella della verità 1.1.

Mux 4:1 Si prosegue con il Mux 4:1.

Come anticipato, viene costruito a partire da tre mux 2:1.

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 entity mux_41 is
5     port
6     (
7         i: in std_logic_vector(0 to 3);
8         s: in std_logic_vector(1 downto 0);
9         u: out std_logic
10    );
11 end mux_41;
12
13 architecture structural of mux_41 is
14     signal u_mid: std_logic_vector(0 to 1);
15
16     component mux_21 is
17         port (
18             i0, i1: in std_logic;
19             s0: in std_logic;
20             u: out std_logic
21         );
22     end component;
23
24 begin
25     mux0to1: FOR k IN 0 TO 1 GENERATE
26         m: mux_21
27             port map
28             (
29                 i(k*2),
30                 i(k*2 + 1),
31                 s(0),
32                 u_mid(k)
33             );
34     end GENERATE;
35
36     mux_2: mux_21
37         port map
38         (
39             u_mid(0),
40             u_mid(1),
41             s(1),
```

```
42          u  
43      );  
44  
45 end structural;
```

Code 1.2: Multiplexer 4:1 in VHDL

In quest'entità, l'interfaccia è dichiarata come segue:

- Il parametro `i` vettore di 4 elementi, ognuno corrispondente ad un ingresso del mux 4:1.
- Il parametro `s` vettore di 2 elementi, ognuno corrispondente ad un ingresso di selezione.
- Il parametro `u` corrispondente all'uscita del multiplexer.

A seguire si definisce la struttura del mux 4:1, utilizzando mux 2:1 come componenti.

Con il ciclo `for`, vengono stanziati i primi due mux 2:1, i quali riceveranno in ingresso rispettivamente, gli ingressi del mux 4:1 e la loro uscita è il vettore d'appoggio `u_mid`, il quale è talvolta l'ingresso del terzo mux 2:1.

Mux 16:1 In maniera analoga si procede con la costruzione del mux 16:1. Il codice è il seguente:

```
1 library IEEE;  
2 use IEEE.std_logic_1164.all;  
3  
4 entity mux_161 is
```

```
5      port
6      (
7          i: in std_logic_vector(0 to 15);
8          s: in std_logic_vector(3 downto 0);
9          u: out std_logic
10     );
11 end mux_161;
12
13 architecture structural of mux_161 is
14
15     signal u_mid: std_logic_vector(0 to 3);
16
17     component mux_41 is
18         port
19         (
20             i: in std_logic_vector(0 to 3);
21             s: in std_logic_vector(0 to 1);
22             u: out std_logic
23         );
24     end component;
25
26     begin
27         mux0to3: FOR k IN 0 TO 3 GENERATE
28             m: mux_41
29             port map
30             (
31                 i => i((k*4) to (k*4 + 3)),
32                 s => s(1 downto 0),
33                 u => u_mid(k)
34             );
35     end GENERATE;
36
37     mux_2: mux_41
38         port map
39         (
40             i => u_mid,
41             s => s(3 downto 2),
42             u => u
43         );
44
45 end structural;
```

Code 1.3: Multiplexer 16:1 in VHDL

1.1.3 Simulazione

Per la simulazione, vi è la necessità di un testbench, il quale generiamo in maniera automatica tramite software appositi.

In tale progetto la generazione viene effettuata tramite ChatGPT ed il codice è il seguente:

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.numeric_std.all; -- Libreria necessaria per `to_unsigned`
4
5 entity tb_mux_161 is
6 end tb_mux_161;
7
8 architecture behavior of tb_mux_161 is
9     -- Component declaration
10    component mux_161
11        port (
12            i: in std_logic_vector(0 to 15);
13            s: in std_logic_vector(3 downto 0);
14            u: out std_logic
15        );
16    end component;
17
18    -- Signals for testing
19    signal i: std_logic_vector(0 to 15);
20    signal s: std_logic_vector(3 downto 0);
21    signal u: std_logic;
22
23 begin
24     -- Instantiate the unit under test (UUT)
25     uut: mux_161
26         port map (
27             i => i,
28             s => s,
29             u => u
30         );
31
32     -- Test process
33     stim_proc: process
34         variable expected_output: std_logic; -- Variabile per il
            ↳ controllo
```

```

35      begin
36          -- Initialize inputs
37          i <= (others => '0');
38          s <= "0000";
39          wait for 10 ns;
40
41          -- Apply test cases
42          for sel in 0 to 15 loop
43              -- Set the ith bit of i to '1'
44              i <= (others => '0');
45              i(sel) <= '1';
46
47              -- Set the selector
48              s <= std_logic_vector(to_unsigned(sel, 4));
49
50              -- Aspetta che l'uscita si stabilizzi
51              wait for 10 ns;
52
53              -- Calcola l'uscita attesa
54              expected_output := i(sel);
55
56              -- Controlla se l'uscita è corretta
57              if u = expected_output then
58                  report "Test passed for s = " & integer'image(sel) &
59                  ", u = " & std_logic'image(u);
60              else
61                  report "Test failed for s = " & integer'image(sel) &
62                  ": expected = " &
63                  "→ std_logic'image(expected_output) &
64                  ", got = " & std_logic'image(u)
65                  severity error;
66              end if;
67          end loop;
68
69          -- Fine simulazione
70          report "All tests completed";
71          wait;
72      end process;
73
end behavior;

```

Code 1.4: Testbench multiplexer 16:1 in VHDL

Una volta generato ciò, utilizzando i software GHDL e GTKWA-

VE, vengono eseguiti i seguenti comandi:

```
[antoniosirignano@Antonios-MacBook-Pro Problema_1 % ghdl -a mux_2_1.vhdl
[antoniosirignano@Antonios-MacBook-Pro Problema_1 % ghdl -a mux_4_1.vhdl
[antoniosirignano@Antonios-MacBook-Pro Problema_1 % ghdl -a mux_16_1.vhdl
[antoniosirignano@Antonios-MacBook-Pro Problema_1 % ghdl -a tb_mux_16_1.vhdl
[antoniosirignano@Antonios-MacBook-Pro Problema_1 % ghdl -e tb_mux_161
antoniosirignano@Antonios-MacBook-Pro Problema_1 % ghdl -r tb_mux_161 --wave=mux_16_1.ghw

tb_mux_16_1.vhdl:58:17:@20ns:(report note): Test passed for s = 0, u = '1'
tb_mux_16_1.vhdl:58:17:@30ns:(report note): Test passed for s = 1, u = '1'
tb_mux_16_1.vhdl:58:17:@40ns:(report note): Test passed for s = 2, u = '1'
tb_mux_16_1.vhdl:58:17:@50ns:(report note): Test passed for s = 3, u = '1'
tb_mux_16_1.vhdl:58:17:@60ns:(report note): Test passed for s = 4, u = '1'
tb_mux_16_1.vhdl:58:17:@70ns:(report note): Test passed for s = 5, u = '1'
tb_mux_16_1.vhdl:58:17:@80ns:(report note): Test passed for s = 6, u = '1'
tb_mux_16_1.vhdl:58:17:@90ns:(report note): Test passed for s = 7, u = '1'
tb_mux_16_1.vhdl:58:17:@100ns:(report note): Test passed for s = 8, u = '1'
tb_mux_16_1.vhdl:58:17:@110ns:(report note): Test passed for s = 9, u = '1'
tb_mux_16_1.vhdl:58:17:@120ns:(report note): Test passed for s = 10, u = '1'
tb_mux_16_1.vhdl:58:17:@130ns:(report note): Test passed for s = 11, u = '1'
tb_mux_16_1.vhdl:58:17:@140ns:(report note): Test passed for s = 12, u = '1'
tb_mux_16_1.vhdl:58:17:@150ns:(report note): Test passed for s = 13, u = '1'
tb_mux_16_1.vhdl:58:17:@160ns:(report note): Test passed for s = 14, u = '1'
tb_mux_16_1.vhdl:58:17:@170ns:(report note): Test passed for s = 15, u = '1'
tb_mux_16_1.vhdl:69:9:@170ns:(report note): All tests completed
[antoniosirignano@Antonios-MacBook-Pro Problema_1 % gtkwave mux_16_1.ghw

GTKWave Analyzer v3.4.0 (w)1999-2022 BSI

[0] start time.
[170000000] end time.
2024-11-25 18:54:06.970 gtkwave[78165:3049537] +[IMKClient subclass]: chose IMKClient_Modern
2024-11-25 18:54:06.970 gtkwave[78165:3049537] +[IMKInputSession subclass]: chose IMKInputSession_Modern
```

Figura 1.7: Comandi per la simulazione

Con l'esecuzione dell'ultimo comando, vi si apre una nuova finestra che permette la visualizzazione delle onde:

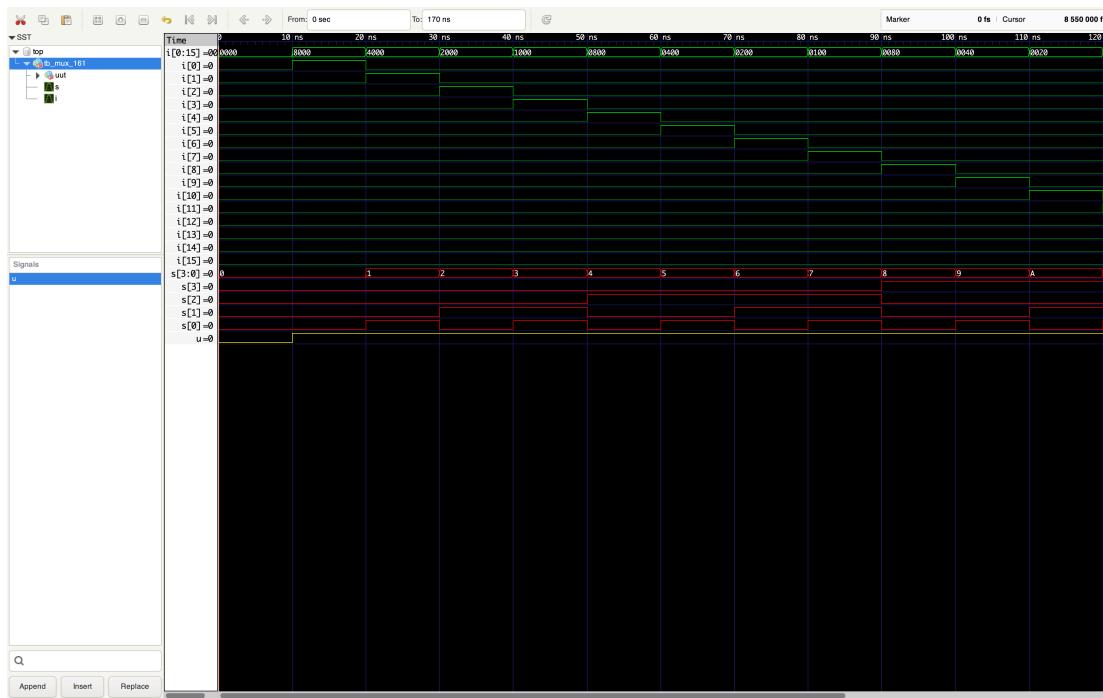


Figura 1.8: Risultati della simulazione: waveform

1.1.4 Implementazione 2.0

In alcuni casi, può essere utile specificare i singoli ingressi, in particolare quando gli ingressi provengono da fonti diverse; in tal caso, è preferibile l'implementazione che segue:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux16_1 is
5     port(
6         i0 : in STD_LOGIC;
7         i1 : in STD_LOGIC;
8         i2 : in STD_LOGIC;
9         i3 : in STD_LOGIC;
10        i4 : in STD_LOGIC;
11        i5 : in STD_LOGIC;
12        i6 : in STD_LOGIC;
13        i7 : in STD_LOGIC;
14        i8 : in STD_LOGIC;
15
16        s[3:0] : out STD_LOGIC;
17        s[2] : out STD_LOGIC;
18        s[1] : out STD_LOGIC;
19        s[0] : out STD_LOGIC;
20        u : out STD_LOGIC
21 );
22 end;
23
24 architecture Behavioral of mux16_1 is
25 begin
26     process(i0,i1,i2,i3,i4,i5,i6,i7,i8,s)
27     begin
28         if (i0 = '1') then
29             u <= i1;
30         elsif (i1 = '1') then
31             u <= i2;
32         elsif (i2 = '1') then
33             u <= i3;
34         elsif (i3 = '1') then
35             u <= i4;
36         elsif (i4 = '1') then
37             u <= i5;
38         elsif (i5 = '1') then
39             u <= i6;
40         elsif (i6 = '1') then
41             u <= i7;
42         elsif (i7 = '1') then
43             u <= i8;
44         else
45             u <= s;
46         end if;
47     end process;
48 end Behavioral;

```

```
14          i9 : in STD_LOGIC;
15          i10 : in STD_LOGIC;
16          i11 : in STD_LOGIC;
17          i12 : in STD_LOGIC;
18          i13 : in STD_LOGIC;
19          i14 : in STD_LOGIC;
20          i15 : in STD_LOGIC;
21          s0 : in STD_LOGIC;
22          s1 : in STD_LOGIC;
23          s2 : in STD_LOGIC;
24          s3 : in STD_LOGIC;
25          y0 : out STD_LOGIC
26      );
27 end mux16_1;
28
29 architecture structural of mux16_1 is
30     signal u0 : STD_LOGIC := '0';
31     signal u1 : STD_LOGIC := '0';
32     signal u2 : STD_LOGIC := '0';
33     signal u3 : STD_LOGIC := '0';
34
35 component mux_2_1
36     port(          a0          : in STD_LOGIC;
37                  a1          : in STD_LOGIC;
38                  s           : in STD_LOGIC;
39                  y           : out STD_LOGIC
40 );
41 end component;
42
43 component mux_4_1
44     port(          b0 : in STD_LOGIC;
45                 b1 : in STD_LOGIC;
46                 b2 : in STD_LOGIC;
47                 b3 : in STD_LOGIC;
48                 s0 : in STD_LOGIC;
49                 s1 : in STD_LOGIC;
50                 y0 : out STD_LOGIC
51 );
52 end component;
53
54 begin
55     mux_0: mux_4_1
56         Port map(    b0 => i0,
57                     b1 => i1,
58                     b2 => i2,
```

```
59          b3 => i3,
60          s0 => s0,
61          s1 => s1,
62          y0 => u0
63      );
64
65      mux_1: mux_4_1
66      Port map(  b0 => i4,
67                  b1 => i5,
68                  b2 => i6,
69                  b3 => i7,
70                  s0 => s0,
71                  s1 => s1,
72                  y0 => u1
73      );
74      mux_2: mux_4_1
75      Port map(  b0 => i8,
76                  b1 => i9,
77                  b2 => i10,
78                  b3 => i11,
79                  s0 => s0,
80                  s1 => s1,
81                  y0 => u2
82      );
83
84
85      mux_3: mux_4_1
86      Port map(  b0 => i12,
87                  b1 => i13,
88                  b2 => i14,
89                  b3 => i15,
90                  s0 => s0,
91                  s1 => s1,
92                  y0 => u3
93      );
94
95      mux_4: mux_4_1
96      Port map(  b0 => u0,
97                  b1 => u1,
98                  b2 => u2,
99                  b3 => u3,
100                 s0 => s2,
101                 s1 => s3,
102                 y0 => y0
103      );
```

104

105 **end structural;**

Code 1.5: Multiplexer 16:1 in VHDL: ingressi trattati separatamente

Ovviamente, la macchina sarà fatta allo stesso modo, come si può vedere dallo schematic generato da Vivado:

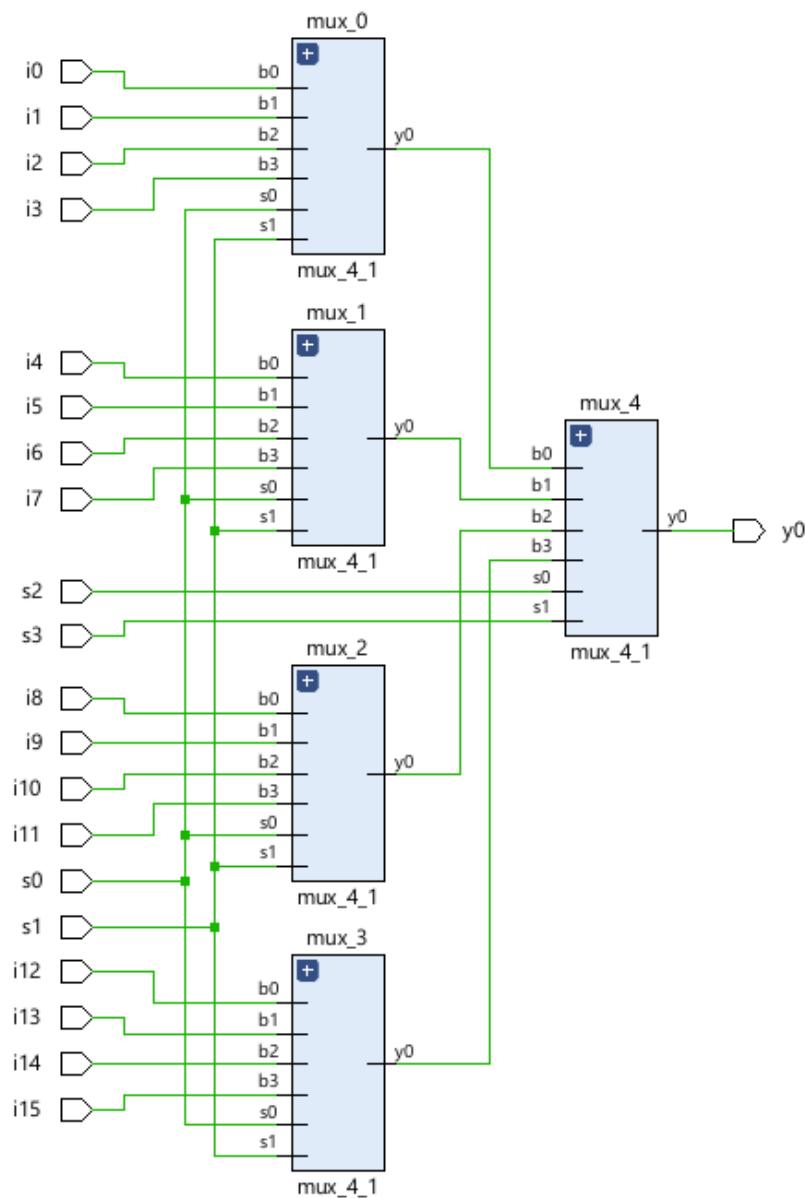


Figura 1.9: Schematic Vivado: Mux 16:1

Il multiplexer lavorerà allo stesso modo, con gli stessi risultati simulativi.

1.2 Rete di interconnessione a 16 ingressi e 4 uscite

Una rete di interconnessione è un tipo di rete di commutazione che permette di instradare i segnali da un insieme di ingressi a un insieme più ridotto di uscite. Tale rete può essere progettata attraverso un adeguato utilizzo di Multiplexer e Demultiplexer.

Nel caso in esame, si vuole progettare una rete che prenda 16 ingressi e restituisca 4 uscite. Si utilizza anche in questo caso un approccio per composizione, a partire dal Multiplexer 16:1 implementato nell'esercizio precedente, la cui uscita sarà posta in ingresso a un Demultiplexer 1:4.

La rete complessiva sarà fatta in questo modo:

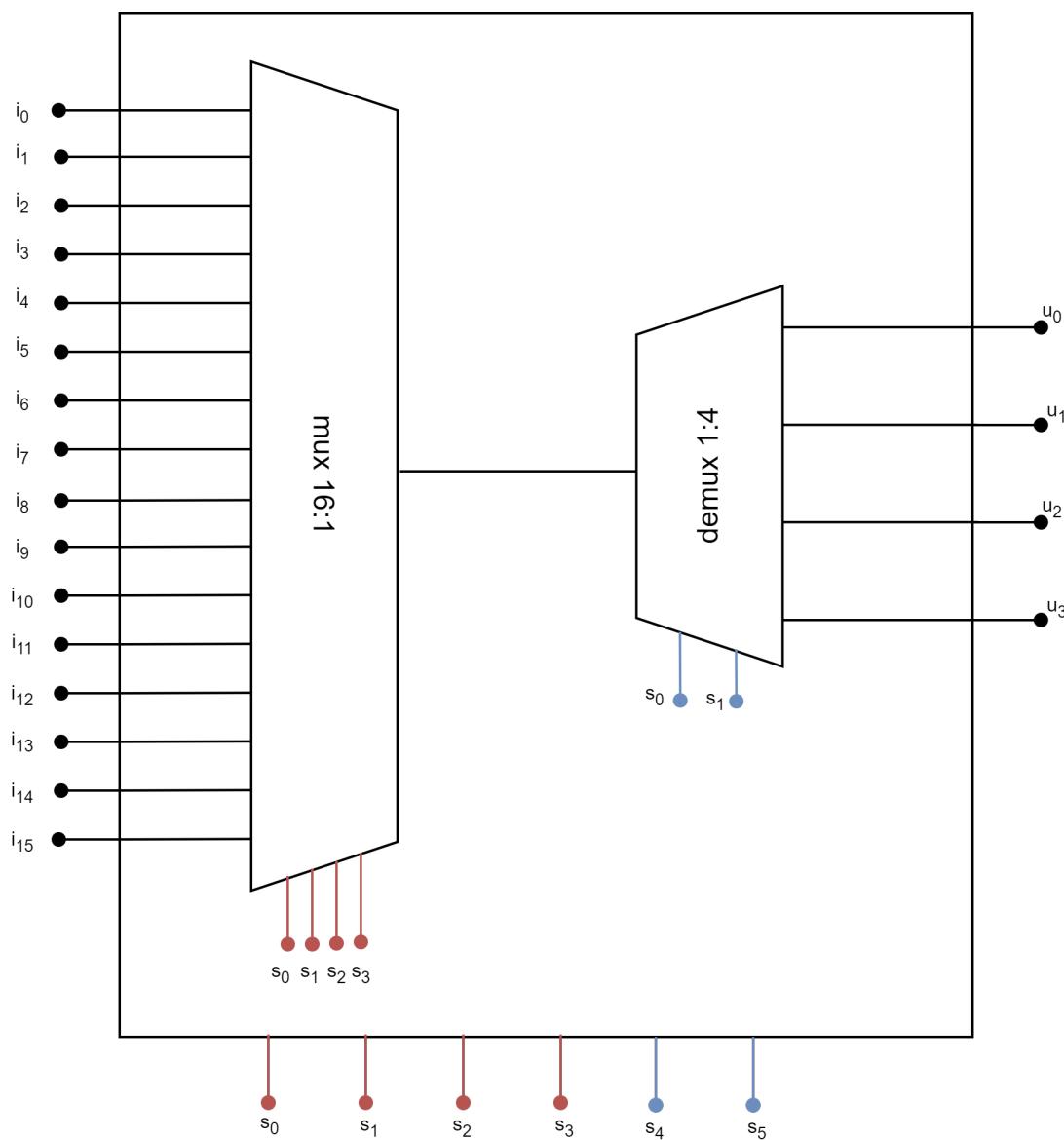


Figura 1.10: Rete di interconnessione

1.2.1 Progettazione

Anche in questo caso, prima di procedere all'implementazione della rete nel complesso, si costruisce il Demultiplexer 4:1 a partire da Demultiplexer 2:1.

Un Demultiplexer $1 : u$ è un dispositivo che prende un solo segnale

di ingresso, due segnali di selezione e a partire da essi restituisce u uscite.

Un Demultiplexer 2:1 è un dispositivo fatto in questo modo:

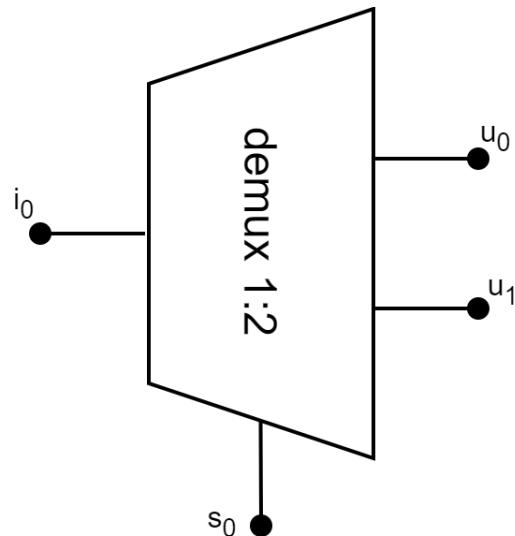


Figura 1.11: Demux 1:2

Tale componente è caratterizzato dalla seguente tabella di verità:

s₀	i₀	u₀	u₁
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

Tabella 1.2: Tabella di verità di un Demux 2:1

Da cui si ricavano le seguenti equazioni relative alle uscite:

$$u_0 = (i_0 \text{ AND } \bar{s}_0)$$

$$u_1 = (i_0 \text{ AND } s_0)$$

A partire dalla composizione di dispositivi di questo tipo, si può realizzare un Demultiplexer 1:4, come rappresentato in figura.

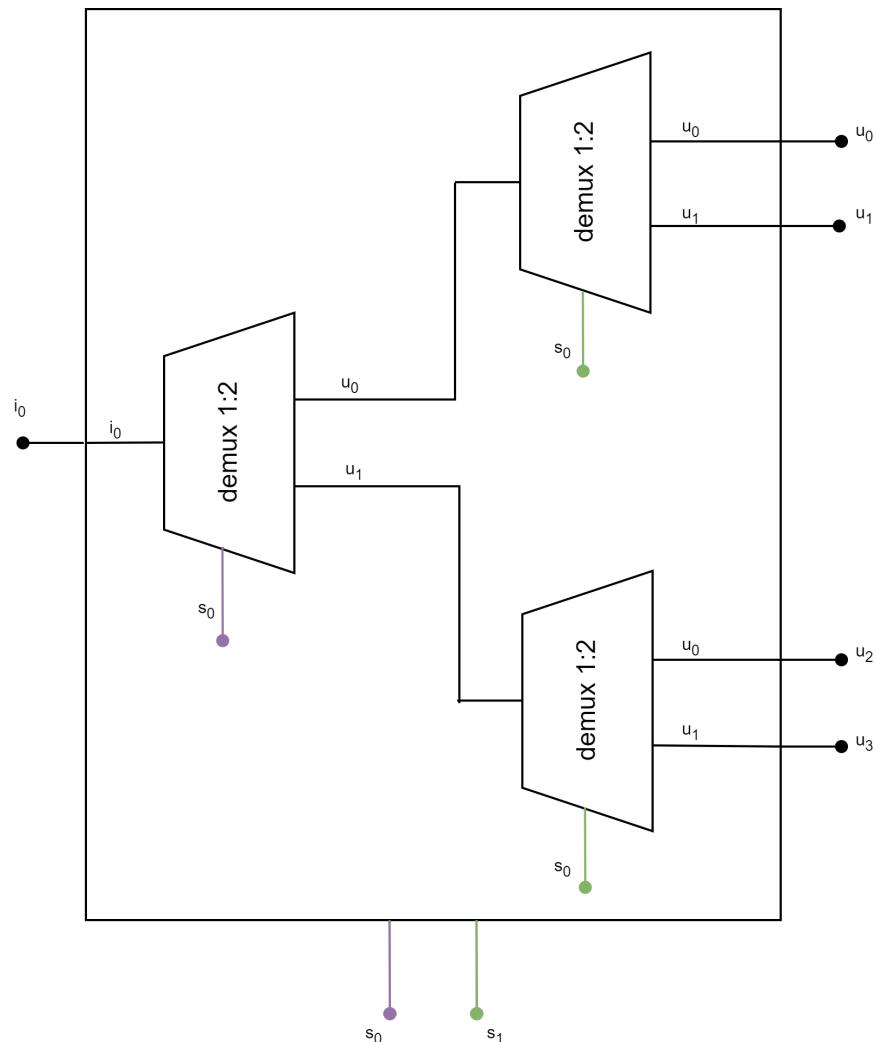


Figura 1.12: Demux 1:4 composto a partire da Demux 1:2

Utilizzando il Demultiplexer appena progettato, al cui ingresso si fa corrispondere l'uscita del Multiplexer 16:1, progettato nell'esercizio precedente, si ottiene la rete di interconnessione, così formata:

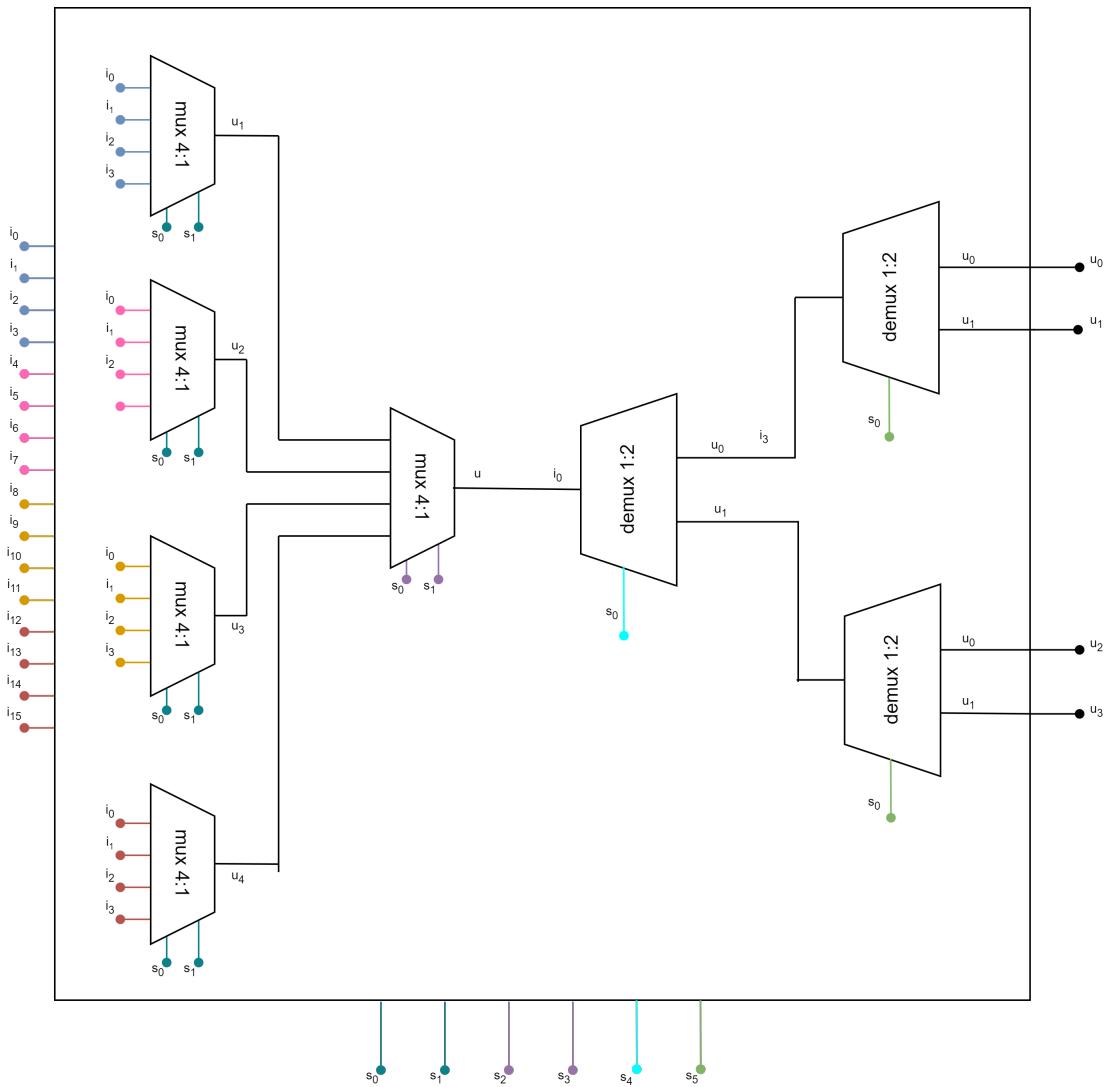


Figura 1.13: Rete di interconnessione: funzionamento interno

Nell'immagine, i colori sono stati usati per rendere più chiari i collegamenti tra segnali.

1.2.2 Implementazione

Si inizia mostrando l'implementazione del Demultiplexer 1:2, fatta seguendo un'architettura di tipo Dataflow.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity demux1_2 is
5     port(
6         a0 :in STD_LOGIC;
7         s0 :in STD_LOGIC;
8         y0 :out STD_LOGIC;
9         y1 :out STD_LOGIC
10    );
11 end demux1_2;
12
13
14
15 architecture dataflow of demux1_2 is
16
17 begin
18     y0 <= (not s0 AND a0);
19     y1 <= (s0 AND a0);
20
21
22 end dataflow;
```

Code 1.6: Demultiplexer 1:2

Come mostrato dalla figura 1.12 presente nella fase di progettazione, a partire da 3 demux 1:2 si può realizzare un demux 1:4 seguendo un approccio di tipo strutturale. Segue il codice:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity demux1_4 is
5     port(
6         i0 :in STD_LOGIC;
7         s0 :in STD_LOGIC;
8         s1 :in STD_LOGIC;
9         y0 :out STD_LOGIC;
10        y1 :out STD_LOGIC;
11        y2 :out STD_LOGIC;
12        y3 :out STD_LOGIC
```

```
13      );
14 end demux1_4;
15
16 architecture structural of demux1_4 is
17   signal u0: STD_LOGIC := '0';
18   signal u1: STD_LOGIC := '0';
19
20
21 component demux1_2
22   port (
23     a0: in STD_LOGIC;
24     s0: in STD_LOGIC;
25     y0: out STD_LOGIC;
26     y1: out STD_LOGIC
27   );
28 end component;
29
30 begin
31
32   demux0: demux1_2
33     Port map(
34       a0 =>i0,
35       s0 =>s0,
36       y0 =>u0,
37       y1 =>u1
38     );
39   demux1: demux1_2
40     Port map(
41       a0 =>u0,
42       s0 =>s1,
43       y0 =>y0,
44       y1 =>y1
45     );
46   demux2: demux1_2
47     Port map(
48       a0 =>u1,
49       s0 =>s1,
50       y0 =>y2,
51       y1 =>y3
52     );
53
54 end structural;
```

Code 1.7: Demultiplexer 1:4

Tramite un'appropriata connessione del Multiplexer realizzato nell'esercizio precedente e il Demux 1:4, si ottiene la rete di interconnessione richiesta:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity interc16_4 is
5     port( i0 : in STD_LOGIC;
6             i1 : in STD_LOGIC;
7             i2 : in STD_LOGIC;
8             i3 : in STD_LOGIC;
9             i4 : in STD_LOGIC;
10            i5 : in STD_LOGIC;
11            i6 : in STD_LOGIC;
12            i7 : in STD_LOGIC;
13            i8 : in STD_LOGIC;
14            i9 : in STD_LOGIC;
15            i10 : in STD_LOGIC;
16            i11 : in STD_LOGIC;
17            i12 : in STD_LOGIC;
18            i13 : in STD_LOGIC;
19            i14 : in STD_LOGIC;
20            i15 : in STD_LOGIC;
21            s0 : in STD_LOGIC;
22            s1 : in STD_LOGIC;
23            s2 : in STD_LOGIC;
24            s3 : in STD_LOGIC;
25            s4 : in STD_LOGIC;
26            s5 : in STD_LOGIC;
27            y0 : out STD_LOGIC;
28            y1 : out STD_LOGIC;
29            y2 : out STD_LOGIC;
30            y3 : out STD_LOGIC
31        );
32    end interc16_4;
33
34 architecture structural of interc16_4 is
35 signal a0 : STD_LOGIC;
36
37 component mux16_1
38     port(
39             i0 : in STD_LOGIC;
```

```
40          i2 : in STD_LOGIC;
41          i3 : in STD_LOGIC;
42          i4 : in STD_LOGIC;
43          i5 : in STD_LOGIC;
44          i6 : in STD_LOGIC;
45          i7 : in STD_LOGIC;
46          i8 : in STD_LOGIC;
47          i9 : in STD_LOGIC;
48          i10 : in STD_LOGIC;
49          i11 : in STD_LOGIC;
50          i12 : in STD_LOGIC;
51          i13 : in STD_LOGIC;
52          i14 : in STD_LOGIC;
53          i15 : in STD_LOGIC;
54          s0 : in STD_LOGIC;
55          s1 : in STD_LOGIC;
56          s2 : in STD_LOGIC;
57          s3 : in STD_LOGIC;
58          y0 : out STD_LOGIC
59      );
60  end component;
61
62 component demux1_4
63     port(      i0          : in STD_LOGIC;
64                 s0          : in STD_LOGIC;
65                 s1          : in STD_LOGIC;
66                 y0          : out STD_LOGIC;
67                 y1          : out STD_LOGIC;
68                 y2          : out STD_LOGIC;
69                 y3          : out STD_LOGIC
70      );
71  end component;
72
73
74
75 begin
76     mux_0: mux16_1
77         Port map(
78             i0 => i0,
79             i1 => i1,
80             i2 => i2,
81             i3 => i3,
82             i4 => i4,
83             i5 => i5,
84             i6 => i6,
```

```

85          i7 => i7,
86          i8 => i8,
87          i9 => i9,
88          i10 => i10,
89          i11 => i11,
90          i12 => i12,
91          i13 => i13,
92          i14 => i14,
93          i15 => i15,
94          s0 => s0,
95          s1 => s1,
96          s2 => s2,
97          s3 => s3,
98          y0 => a0
99      );
100
101     demux0: demux1_4
102         Port map(
103             i0 => a0,
104             s0 => s4,
105             s1 => s5,
106             y0 => y0,
107             y1 => y1,
108             y2 => y2,
109             y3 => y3
110         );
111
112 end structural;

```

Code 1.8: Rete di interconnessione 16:4 in VHDL

La rete realizzata è osservabile come schematic generato da Vivado:

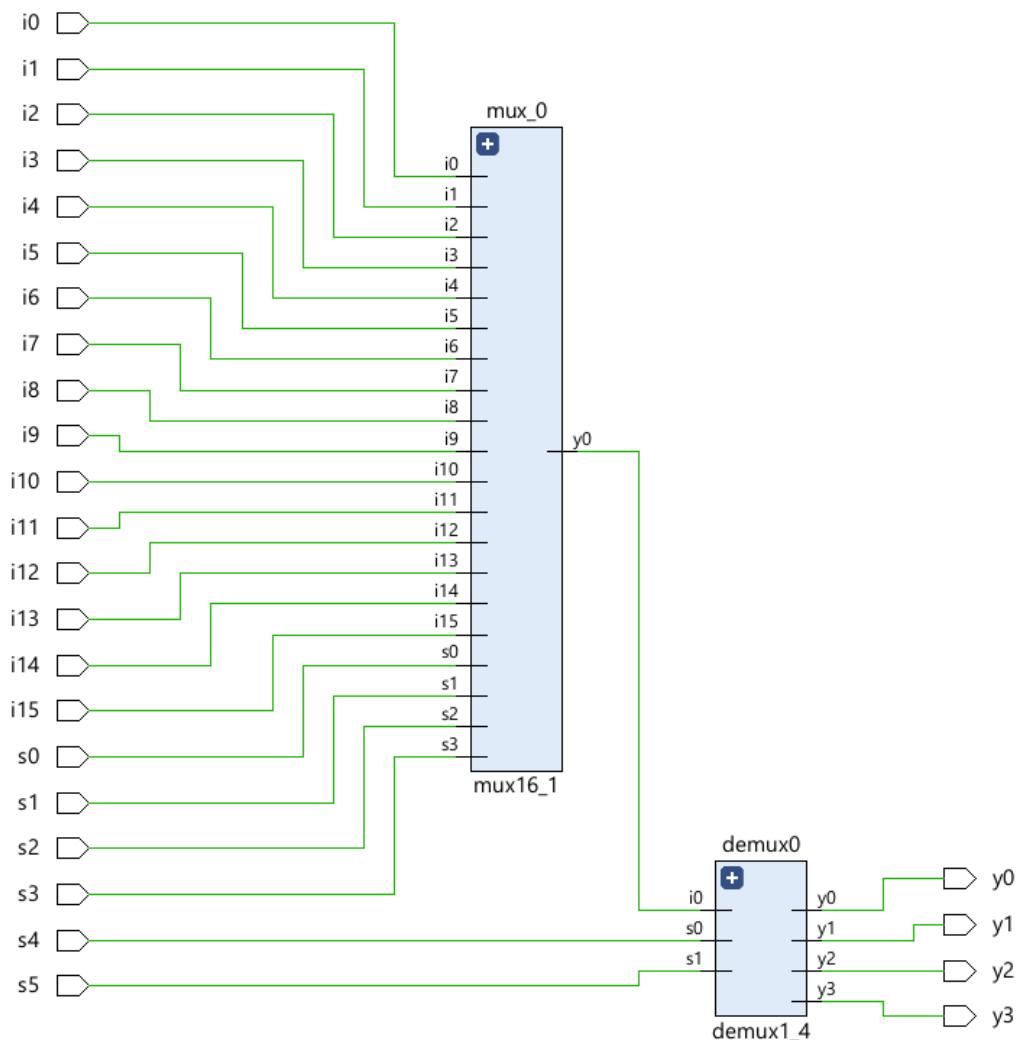


Figura 1.14: Rete di interconnessione: schematic

1.2.3 Simulazione

Per procedere con la simulazione della rete realizzata, si utilizza un tesbench. Tale testebnch è stato realizzato tramite il sito Doulos, e sono stati manualmente aggiunti diversi casi di test:

```

1 library IEEE;
2 use IEEE.Std_logic_1164.all;
3 use IEEE.Numeric_Signed.all;
4

```

```
5  entity interc16_4_tb is
6  end;
7
8  architecture bench of interc16_4_tb is
9
10 component interc16_4
11     port( i0 : in STD_LOGIC;
12             i1 : in STD_LOGIC;
13             i2 : in STD_LOGIC;
14             i3 : in STD_LOGIC;
15             i4 : in STD_LOGIC;
16             i5 : in STD_LOGIC;
17             i6 : in STD_LOGIC;
18             i7 : in STD_LOGIC;
19             i8 : in STD_LOGIC;
20             i9 : in STD_LOGIC;
21             i10 : in STD_LOGIC;
22             i11 : in STD_LOGIC;
23             i12 : in STD_LOGIC;
24             i13 : in STD_LOGIC;
25             i14 : in STD_LOGIC;
26             i15 : in STD_LOGIC;
27             s0 : in STD_LOGIC;
28             s1 : in STD_LOGIC;
29             s2 : in STD_LOGIC;
30             s3 : in STD_LOGIC;
31             s4 : in STD_LOGIC;
32             s5 : in STD_LOGIC;
33             y0 : out STD_LOGIC;
34             y1 : out STD_LOGIC;
35             y2 : out STD_LOGIC;
36             y3 : out STD_LOGIC
37         );
38     end component;
39
40     signal i0: STD_LOGIC;
41     signal i1: STD_LOGIC;
42     signal i2: STD_LOGIC;
43     signal i3: STD_LOGIC;
44     signal i4: STD_LOGIC;
45     signal i5: STD_LOGIC;
46     signal i6: STD_LOGIC;
47     signal i7: STD_LOGIC;
48     signal i8: STD_LOGIC;
49     signal i9: STD_LOGIC;
```

```
50  signal i10: STD_LOGIC;
51  signal i11: STD_LOGIC;
52  signal i12: STD_LOGIC;
53  signal i13: STD_LOGIC;
54  signal i14: STD_LOGIC;
55  signal i15: STD_LOGIC;
56  signal s0: STD_LOGIC;
57  signal s1: STD_LOGIC;
58  signal s2: STD_LOGIC;
59  signal s3: STD_LOGIC;
60  signal s4: STD_LOGIC;
61  signal s5: STD_LOGIC;
62  signal y0: STD_LOGIC;
63  signal y1: STD_LOGIC;
64  signal y2: STD_LOGIC;
65  signal y3: STD_LOGIC ;

66
66
67 begin
68
69     uut: interc16_4 port map ( i0    => i0,
70                               i1    => i1,
71                               i2    => i2,
72                               i3    => i3,
73                               i4    => i4,
74                               i5    => i5,
75                               i6    => i6,
76                               i7    => i7,
77                               i8    => i8,
78                               i9    => i9,
79                               i10   => i10,
80                               i11   => i11,
81                               i12   => i12,
82                               i13   => i13,
83                               i14   => i14,
84                               i15   => i15,
85                               s0    => s0,
86                               s1    => s1,
87                               s2    => s2,
88                               s3    => s3,
89                               s4    => s4,
90                               s5    => s5,
91                               y0    => y0,
92                               y1    => y1,
93                               y2    => y2,
94                               y3    => y3 );
```

```

95
96 stimulus: process
97 begin
98
99   -- Inizializzazione segnali
100  i0 <= '0'; i1 <= '0'; i2 <= '0'; i3 <= '0';
101  i4 <= '0'; i5 <= '0'; i6 <= '0'; i7 <= '0';
102  i8 <= '0'; i9 <= '0'; i10 <= '0'; i11 <= '0';
103  i12 <= '0'; i13 <= '0'; i14 <= '0'; i15 <= '0';
104  s0 <= '0'; s1 <= '0'; s2 <= '0'; s3 <= '0';
105  s4 <= '0'; s5 <= '0';
106  wait for 10 ns;
107
108  -- Test Case 1: Selezione i0
109  i0 <= '1'; s0 <= '0'; s1 <= '0'; s2 <= '0'; s3 <= '0'; s4 <=
110    ↪ '0'; s5 <= '0';
111  wait for 10 ns;
112
113  -- Test Case 2: Selezione i3
114  i3 <= '1'; s0 <= '1'; s1 <= '1'; s2 <= '0'; s3 <= '0'; s4 <=
115    ↪ '0'; s5 <= '0';
116  wait for 10 ns;
117
118  -- Test Case 3: Selezione i7
119  i7 <= '1'; s0 <= '1'; s1 <= '1'; s2 <= '1'; s3 <= '0'; s4 <=
120    ↪ '0'; s5 <= '0';
121  wait for 10 ns;
122
123  -- Test Case 4: Selezione i12
124  i12 <= '1'; s0 <= '0'; s1 <= '0'; s2 <= '0'; s3 <= '1'; s4 <=
125    ↪ '1'; s5 <= '0';
126  wait for 10 ns;
127
128  -- Test Case 5: Selezione i15
129  i15 <= '1'; s0 <= '1'; s1 <= '1'; s2 <= '1'; s3 <= '1'; s4 <=
130    ↪ '1'; s5 <= '0';
131  wait for 10 ns;
132
133  -- Test Case 6: Nessun ingresso attivo
134  i0 <= '0'; i1 <= '0'; i2 <= '0'; i3 <= '0';
135  i4 <= '0'; i5 <= '0'; i6 <= '0'; i7 <= '0';
136  i8 <= '0'; i9 <= '0'; i10 <= '0'; i11 <= '0';
137  i12 <= '0'; i13 <= '0'; i14 <= '0'; i15 <= '0';
138  s0 <= '0'; s1 <= '0'; s2 <= '0'; s3 <= '0'; s4 <= '0'; s5 <=
139    ↪ '0';

```

```

134     wait for 10 ns;
135
136     -- Test Case 7: Selezione i5
137     i5 <= '1'; s0 <= '1'; s1 <= '0'; s2 <= '1'; s3 <= '0'; s4 <=
138         => '0'; s5 <= '0';
139     wait for 10 ns;
140
141     -- Test Case 8: Selezione i10
142     i10 <= '1'; s0 <= '0'; s1 <= '1'; s2 <= '0'; s3 <= '1'; s4 <=
143         => '0'; s5 <= '0';
144     wait for 10 ns;
145
146     -- Test Case 9: Selezione i6
147     i6 <= '1'; s0 <= '1'; s1 <= '0'; s2 <= '1'; s3 <= '1'; s4 <=
148         => '0'; s5 <= '0';
149     wait for 10 ns;
150
151     -- Test Case 10: Selezione i9
152     i9 <= '1'; s0 <= '0'; s1 <= '1'; s2 <= '1'; s3 <= '0'; s4 <=
153         => '0'; s5 <= '0';
154     wait for 10 ns;
155
156     -- Fine del test
157     wait;
158 end process;
159
160 end;

```

Code 1.9: Testbench: Rete di interconnessione 16:4

I risultati di tale simulazione sono osservabili nella seguente waveform realizzata dal tool di Vivado.

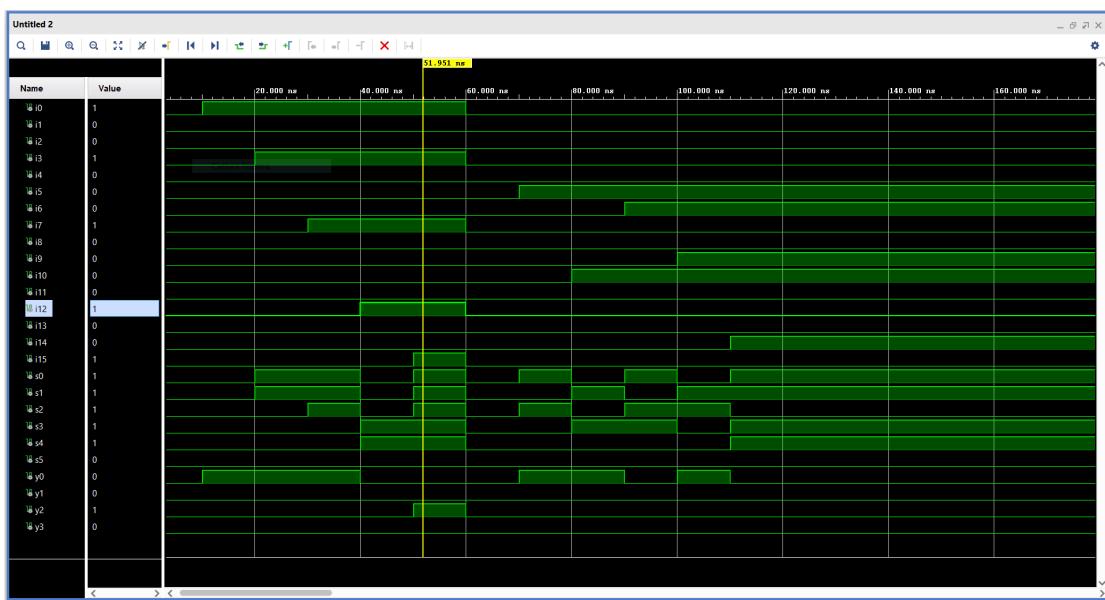


Figura 1.15: Rete di interconnessione: waveform

1.3 Implementazione su board del punto precedente

La board utilizzata è la **Nexys A7**, una scheda di sviluppo basata su FPGA progettata da Digilent.

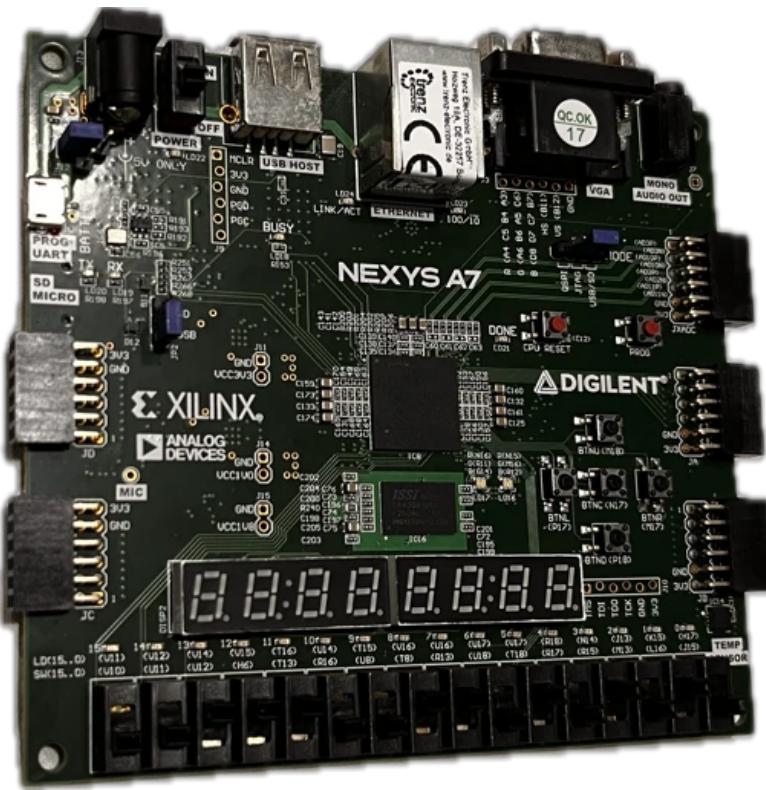


Figura 1.16: Board Nexys A7

1.3.1 Traccia

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 devono essere immessi mediante switch, 8 bit alla volta, sviluppando un'apposita rete di controllo per laacquisizione che utilizzi due bottoni della board per caricare rispettivamente la prima e la seconda metà del dato in ingresso.

1.3.2 Implementazione

Per permettere lo sviluppo sulla board, è stato necessario gestire gli input in modo appropriato; per fare ciò che viene richiesto, si è scelto di usare il bottone *BTNL* per il caricamento della prima metà degli ingressi, il bottone *BTNR* per il caricamento della seconda metà degli ingressi, e il bottone *BTNU* per il caricamento dei segnali di selezione; inoltre è stato previsto un bottone per il reset, *BTNC*. Gli ingressi sono stati gestiti con gli switch, e le uscite sono visualizzabili tramite i led. I primi 8 switch (da 0 a 7) sono stati utilizzati per gli ingressi, mentre i successivi 6 (da 8 a 13) per le selezioni. I led utilizzati per le uscite sono invece i primi 4 (da 0 a 3). Per permettere opportune connessioni tra i componenti hardware e i segnali utilizzati nella rete di interconnessione, è stata implementata una unità di controllo, che ha gestito gli ingressi in due fasi distinte, oltre che i segnali di selezione. Segue il codice dell'unità di controllo:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity control_unit is
5     Port (
6         clock : in STD_LOGIC;
7         reset : in STD_LOGIC;
8         load_first_part : in STD_LOGIC;
9         load_second_part : in STD_LOGIC;
10        load_selection: in STD_LOGIC;
11        value8_in : in STD_LOGIC_VECTOR(7 downto 0);
12        --valore acquisito dai primi 8 switch
13        value16_out: out STD_LOGIC_VECTOR(15 downto 0);
14        selection_in: in STD_LOGIC_VECTOR(5 downto 0);
```

```

14                     sel_out: out STD_LOGIC_VECTOR(5 downto 0)
15                         );
16 end control_unit;
17
18 architecture Behavioral of control_unit is
19
20 signal reg_value : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
21 signal selection_value: STD_LOGIC_VECTOR(5 downto 0);
22
23 begin
24 value16_out <= reg_value;
25 sel_out <= selection_value;
26
27 main: process(clock)
28 begin
29
30     if clock'event and clock = '1' then
31         if reset = '1' then
32             reg_value <= (others => '0');
33         else
34             if load_first_part = '1' then
35                 reg_value(7 downto 0) <= value8_in;
36             elsif load_second_part = '1' then
37                 reg_value(15 downto 8) <= value8_in;
38             elsif load_selection = '1' then
39                 selection_value <= selection_in;
40             end if;
41         end if;
42     end if;
43
44 end process;
45
46
47 end Behavioral;

```

Code 1.10: Control unit

Inoltre, per consentire il funzionamento del sistema sulla board, è stato implementato il seguente codice:


```
37          i1 : in STD_LOGIC;
38          i2 : in STD_LOGIC;
39          i3 : in STD_LOGIC;
40          i4 : in STD_LOGIC;
41          i5 : in STD_LOGIC;
42          i6 : in STD_LOGIC;
43          i7 : in STD_LOGIC;
44          i8 : in STD_LOGIC;
45          i9 : in STD_LOGIC;
46          i10 : in STD_LOGIC;
47          i11 : in STD_LOGIC;
48          i12 : in STD_LOGIC;
49          i13 : in STD_LOGIC;
50          i14 : in STD_LOGIC;
51          i15 : in STD_LOGIC;
52          s0 : in STD_LOGIC;
53          s1 : in STD_LOGIC;
54          s2 : in STD_LOGIC;
55          s3 : in STD_LOGIC;
56          s4 : in STD_LOGIC;
57          s5 : in STD_LOGIC;
58          y0 : out STD_LOGIC;
59          y1 : out STD_LOGIC;
60          y2 : out STD_LOGIC;
61          y3 : out STD_LOGIC
62      );
63  end component;
64
65 signal cu_value: STD_LOGIC_VECTOR(15 downto 0);
66 signal cu_sel: STD_LOGIC_VECTOR( 5 downto 0);
67
68 begin
69   cu: control_unit
70     port map(
71       clock => clock,
72       reset => reset,
73       load_first_part => load_first_part,
74       load_second_part => load_second_part,
75       load_selection => load_selection,
76       value8_in => value8_in,
77       value16_out => cu_value,
78       selection_in => selection_in,
79       sel_out => cu_sel
80     );
81
```

```
82    ri: interc16_4
83    port map(
84        i0 => cu_value(0),
85        i1 => cu_value(1),
86        i2 => cu_value(2),
87        i3 => cu_value(3),
88        i4 => cu_value(4),
89        i5 => cu_value(5),
90        i6 => cu_value(6),
91        i7 => cu_value(7),
92        i8 => cu_value(8),
93        i9 => cu_value(9),
94        i10 => cu_value(10),
95        i11 => cu_value(11),
96        i12 => cu_value(12),
97        i13 => cu_value(13),
98        i14 => cu_value(14),
99        i15 => cu_value(15),
100       s0 => cu_sel(0),
101       s1 => cu_sel(1),
102       s2 => cu_sel(2),
103       s3 => cu_sel(3),
104       s4 => cu_sel(4),
105       s5 => cu_sel(5),
106       y0 => y0,
107       y1 => y1,
108       y2 => y2,
109       y3 => y3
110    );
111
112 end structural;
```

Code 1.11: Implementazione: Rete di interconnessione on Board

1.3.3 Funzionamento

Di seguito si mostra l'esecuzione su board di uno dei casi di test visti in precedenza nella fase di simulazione. In particolare è stato testato ciò che avveniva a 51 ns, e si può vedere che il led acceso corrisponde

con l'uscita attesa $y2$.

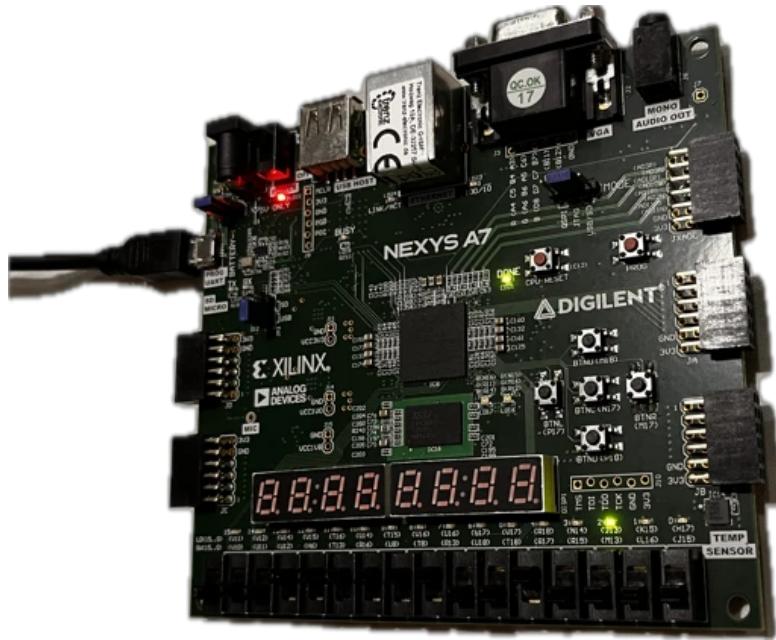


Figura 1.17: Uscita $y2$ attiva

Capitolo 2

Esercizio 2 - Sistema ROM+M

Il sistema che si vuole costruire consiste in due elementi principali: una ROM (Read-Only-Memory) puramente combinatoria e una macchina combinatoria M, che esegue una trasformazione sui dati letti da M e li pone in uscita. La ROM si compone di 16 locazioni di memoria, ciascuna contenente una stringa di 8 bit. Il sistema prende in ingresso un indirizzo di 4 bit, che permetterà di accedere a una delle locazioni della ROM; il dato in tale locazione viene posto in uscita alla ROM, e quindi in ingresso alla macchina M. La macchina M deve effettuare una trasformazione sulla stringa di 8 bit, in modo da restituire in uscita una stringa di 4 bit. La trasformazione scelta consiste nel sommare i 4 bit più significativi della stringa con i 4 bit meno significativi, la stringa di 4 bit risultante sarà restituita come

uscita all'intero sistema.

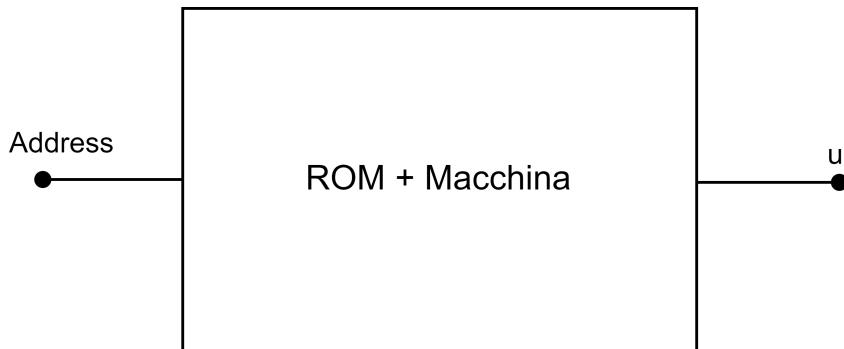


Figura 2.1: ROM + M

2.1 Progettazione

La progettazione consiste nella realizzazione dei due componenti fondamentali del sistema: ROM e M.

2.2 Implementazione

Dapprima si implementa la ROM, in cui sono memorizzati 16 elementi, ciascuno da 8 bit. Il codice sottostante crea l'entità ROM, al cui ingresso è presente un vettore da 4 bit di `std_logic` che rappresenta l'indirizzo, e in uscita restituisce un vettore di 8 bit. Vengono poi definite le stringhe di bit contenute nella ROM. Nel processo `main`, si pone in uscita l'elemento corrispondente alla locazione `address`.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5
6  entity ROM is
7      port(
8          address: in STD_LOGIC_VECTOR(3 downto 0);
9          dout: out STD_LOGIC_VECTOR(7 downto 0)
10         );
11 end entity ROM;
12
13
14 architecture RTL of ROM is
15
16 type MEMORY is array(15 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
17    --memoria da N locazioni che contengono 8 bit
18 constant ROM_N: MEMORY := (
19     "01000000", -- in locazione 15
20     "01000001",
21     "01000010",
22     "01000011",
23     "00010100",
24     "01000101",
25     "00000110",
26     "01000111",
27     "00001000",
28     "00001001",
29     "01001010",
30     "00001011",
31     "00001100",
32     "00001101",
33     "10001010",
34     "00001001" --in locazione 0
35 );
36
37 begin
38 main: process(address)
39 begin
40 dout<= ROM_N(TO_INTEGER(unsigned(address))); --lettura dalla rom
41 end process main;
42 end architecture RTL;

```

Code 2.1: Implementazione ROM in VHDL

Si procede poi con l'implementazione del componente M, che effettua la trasformazione.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity M is
6   port(
7     ingresso: in std_logic_vector(7 downto 0);
8     uscita: out std_logic_vector(3 downto 0)
9   );
10
11 end entity M;
12
13 architecture Behavioral of M is
14 begin
15   process(ingresso)
16   begin
17     -- Somma dei 4 bit pi significativi e dei 4 meno
18     -- significativi
19     uscita <= std_logic_vector(unsigned(ingresso(7 downto 4)) +
20       unsigned(ingresso(3 downto 0)));
21   end process;
22 end Behavioral;

```

Code 2.2: Macchina M

Nel processo si pone come uscita della macchina la somma tra i bit più significativi dell'ingresso (dal bit 7 al 4) e dei bit meno significativi (dal bit 3 allo 0).

Le due componenti sono parte del sistema S che è così implementato:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
```

```

5  entity ROMplusM is
6      Port (
7          A: in std_logic_vector(3 downto 0); --indirizzo in ingresso
8              -- al sistema
9          bout: out std_logic_vector(3 downto 0) --uscita complessiva
10             -- del sistema
11      );
12 end ROMplusM;
13
14
15 component ROM
16     port (
17         address: in STD_LOGIC_VECTOR(3 downto 0);
18         dout: out STD_LOGIC_VECTOR(7 downto 0)
19     );
20 end component;
21 component M
22     port (
23         ingresso: in std_logic_vector(7 downto 0);
24         uscita: out std_logic_vector(3 downto 0)
25     );
26 end component;
27
28 begin
29     -- Istanza della ROM
30     rom_instance: ROM
31         port map(
32             address => A,
33             dout => u0
34         );
35     -- Istanza della macchina combinatoria M
36     transform: M
37         port map(
38             ingresso => u0,
39             uscita => bout
40         );
41 end structural;

```

Code 2.3: Sistema S

Tale sistema è stato costruito come structural: sono stati dichiarati i componenti, e ne sono state definite le istanze. Si è utilizzato

un segnale di supporto u_0 , che funge da segnale intermedio tra l'uscita della ROM e l'ingresso della macchina.

Si osserva lo schematic fornito dall'ambiente di sviluppo Vivado:



Figura 2.2: Schematic di S

2.3 Simulazione

Per procedere alla simulazione si realizza un testbench, con diversi casi di test, che permettano di osservare il comportamento del sistema.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity ROMplusM_tb is
6     -- Un testbench non ha porte, un'entità vuota.
7 end ROMplusM_tb;
8
9 architecture behavior of ROMplusM_tb is
10
11     -- Component declaration for the unit under test (UUT)
12     component ROMplusM
13         Port (
14             A: in std_logic_vector(3 downto 0);
15             bout: out std_logic_vector(3 downto 0)
16         );
    
```

```

17      end component;
18
19      -- Signals to connect to the UUT
20      signal A_tb: std_logic_vector(3 downto 0) := (others => '0'); -->
21          -- Ingresso inizializzato a 0
22      signal bout_tb: std_logic_vector(3 downto 0); -- Uscita
23
24
25      -- Instantiation of the UUT (Unit Under Test)
26      uut: ROMplusM
27      Port map (
28          A => A_tb,
29          bout => bout_tb
30      );
31
32      -- Stimulus process to provide inputs and check outputs
33      stimulus_process: process
34      begin
35          -- Test 1: Indirizzo A = 0
36          A_tb <= "0000";
37          wait for 10 ns;
38
39          -- Test 2: Indirizzo A = 1
40          A_tb <= "0001";
41          wait for 10 ns;
42
43          -- Test 3: Indirizzo A = 2
44          A_tb <= "0010";
45          wait for 10 ns;
46
47          -- Test 4: Indirizzo A = 3
48          A_tb <= "0011";
49          wait for 10 ns;
50
51          -- Test 5: Indirizzo A = 5
52          A_tb <= "0101";
53          wait for 10 ns;
54
55          -- Test 6: Indirizzo A = 7
56          A_tb <= "0111";
57          wait for 10 ns;
58
59          -- Test 7: Indirizzo A = 10
60          A_tb <= "1010";

```

```

61      wait for 10 ns;
62
63      -- Test 8: Indirizzo A = 255
64      A_tb <= "1111";
65      wait for 10 ns;
66
67      -- Fine simulazione
68      wait;
69      end process;
70
71 end behavior;
```

Code 2.4: Testbench

La seguente figura permette la visualizzazione delle waveform.

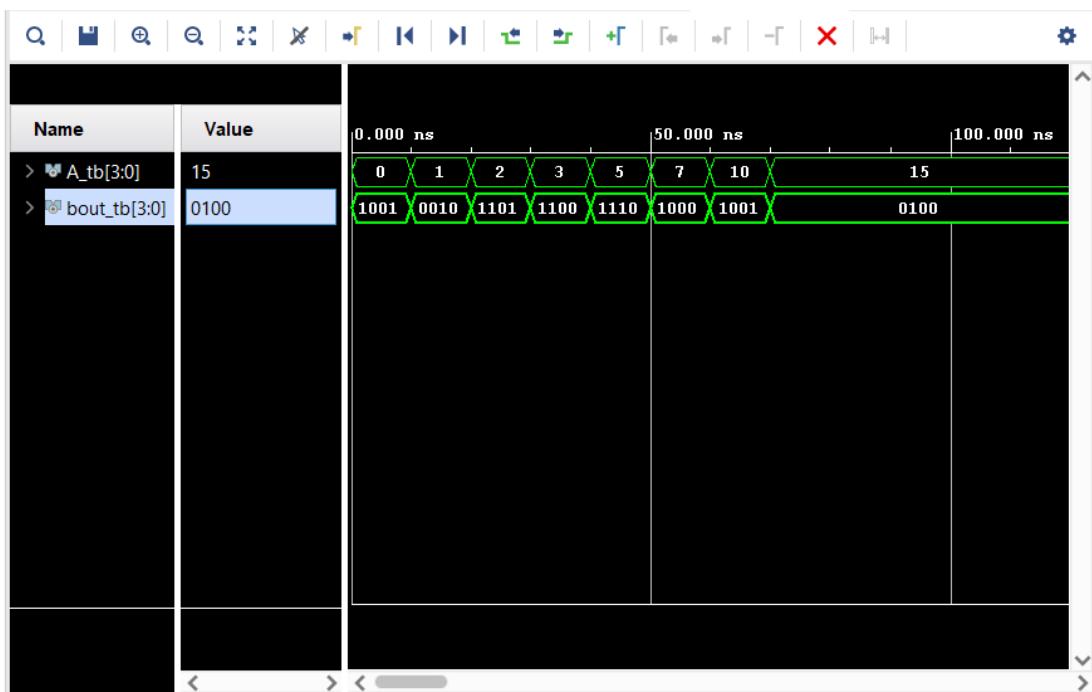


Figura 2.3: Waveform della simulazione di S

Si procede con dei test effettuati manualmente per mostrare la correttezza nel funzionamento del sistema S. Per consentire una maggiore leggibilità, si è scelto di visualizzare gli indirizzi come Unsigned Decimal.

Nel caso $A = 0$, si accede alla stringa 00001001, sommando i bit meno significativi con quelli più significativi si ottiene $0000 + 1001 = 1001$; nel caso $A = 5$, si accede alla stringa 01001010, e procedendo come sopra si ottiene $0100 + 1010 = 1110$.

Come si può vedere, i risultati di questi test coincidono con il comportamento atteso dal sistema e che sono mostrati nella waveform relativa alla simulazione.

2.4 Implementazione su board

2.4.1 Traccia

Sintetizzare ed implementare su board il progetto del sistema ROM+M sviluppato al punto 2.1, utilizzando gli switch per fornire l'indirizzo della ROM da cui leggere i valori da trasformare e i led per visualizzare i 4 bit di uscita.

2.4.2 Implementazione

In questo caso, per implementare il sistema sulla board, è stato sufficiente modificare il file `Nexys-A7-50T-Master.xdc`, collegando i primi 4 switch (da 0 a 3) all'indirizzo A in ingresso, e i led da 0 a 3 alle uscite bout della macchina.

In particolare, il file xdc è composto dalle seguenti righe utili:

```
#GESTIONE SWITCH
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports{ A[0]}];
# IO_L24N_T3_RS0_15 Sch=sw[0]
```

CAPITOLO 2. ESERCIZIO 2 - SISTEMA ROM+M

```
set_property -dict {PACKAGE_PIN L16 IO_STANDARD LVCMOS33} [get_ports{ A[1]}];
# IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
set_property -dict {PACKAGE_PIN M13 IO_STANDARD LVCMOS33} [get_ports{ A[2]}];
# IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict {PACKAGE_PIN R15 IO_STANDARD LVCMOS33} [get_ports{ A[3]}];
# IO_L13N_T2_MRCC_14 Sch=sw[3]
#GESTIONE LED
set_property -dict {PACKAGE_PIN H17 IO_STANDARD LVCMOS33} [get_ports {bout[0]}];
#IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict {PACKAGE_PIN K15 IO_STANDARD LVCMOS33} [get_ports {bout[1]}];
#IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict {PACKAGE_PIN J13 IO_STANDARD LVCMOS33} [get_ports {bout[2]}];
#IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict {PACKAGE_PIN N14 IO_STANDARD LVCMOS33} [get_ports {bout[3]}];
#IO_L8P_T1_D11_14 Sch=led[3]
```

Si mostrano in seguito alcuni test eseguiti sulla board, che hanno confermato i risultati ottenuti dalla simulazione.

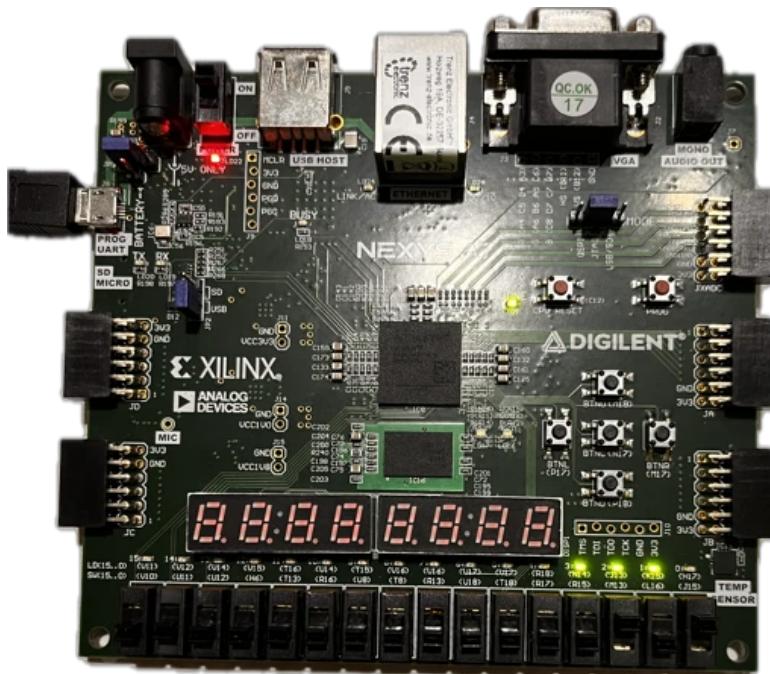


Figura 2.4: A = "0101", bout = "1110"

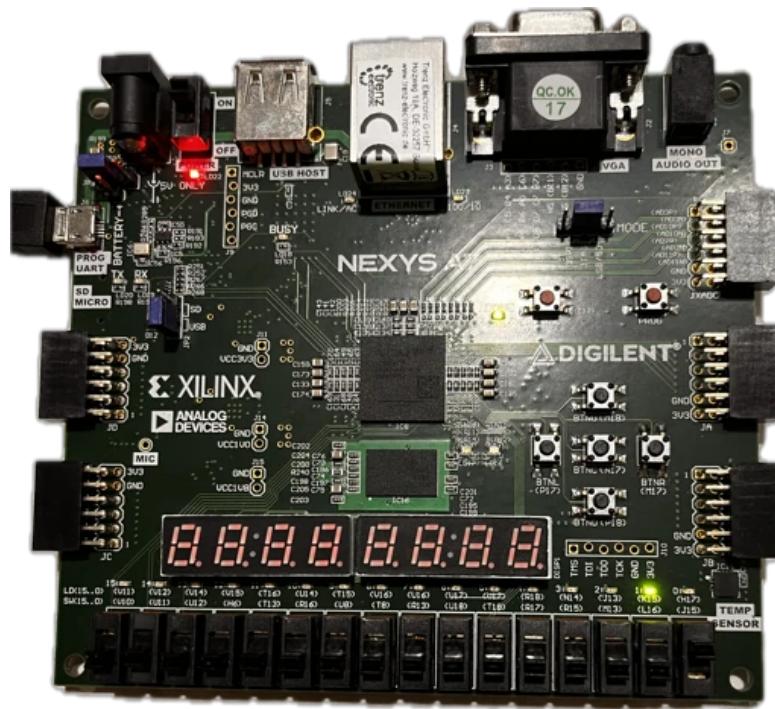


Figura 2.5: $A = "0001"$, $bout = "0010"$

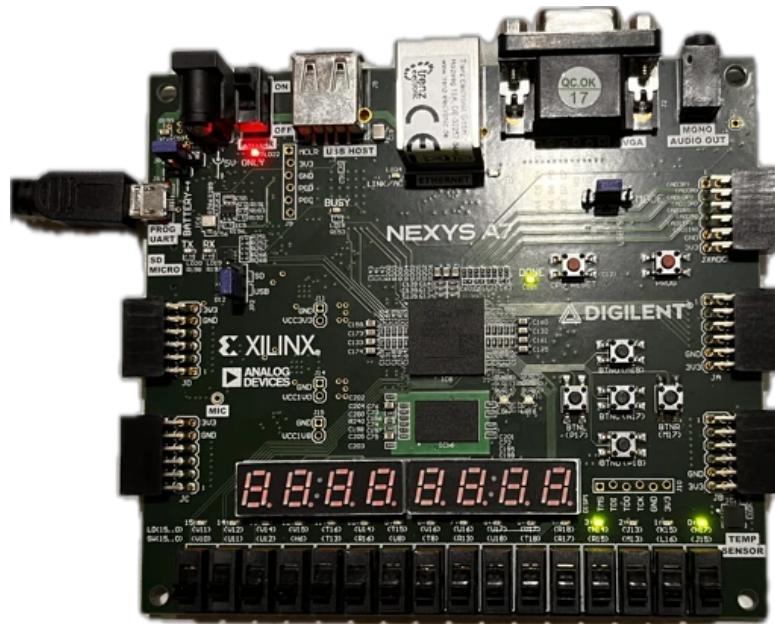


Figura 2.6: $A = "1001"$, $bout = "1001"$

Confrontando i risultati ottenuti con la waveform generata dalla simulazione si confermano le conclusioni precedenti.

Capitolo 3

Esercizio 3

3.1 Riconoscitore di sequenze

Un **riconoscitore di sequenze**, è una macchina sequenziale impulsiva¹ che riceve una sequenza di bit in ingresso e che, a seconda se tale sequenza sia uguale o non ad una data, ritorni i valori 1 e 0, rispettivamente.

In particolare si possono avere due tipi di riconoscitori:

1. **riconoscitori di sequenze non sovrapposte**: valuta i bit in ingresso a gruppi di n elementi alla volta;
2. **riconoscitori di sequenze parzialmente sovrapposte**: valuta i bit in ingresso a uno alla volta, tornando allo stato iniziale ogni qual volta la sequenza viene riconosciuta.

¹Macchina in cui l'uscita è vera solo per un determinato stato e per un determinato ingresso, e poi torna ad essere falsa.

Nel caso in esame si vuole implementare un riconoscitore della sequenza **101**.

Oltre al dato, tale macchina ha in ingresso la tempificazione A e il valore M , che nel caso in cui $M = 0$, la macchina lavora come riconoscitore di sequenze non sovrapposte, mentre se $M = 1$ lavora come riconoscitore di sequenze parzialmente sovrapposte.

3.1.1 Progettazione e architettura

Per progettare una macchina sequenziale, vi è bisogno dell'automa a stati finiti.

Nel caso in questione, vi è il seguente risultato

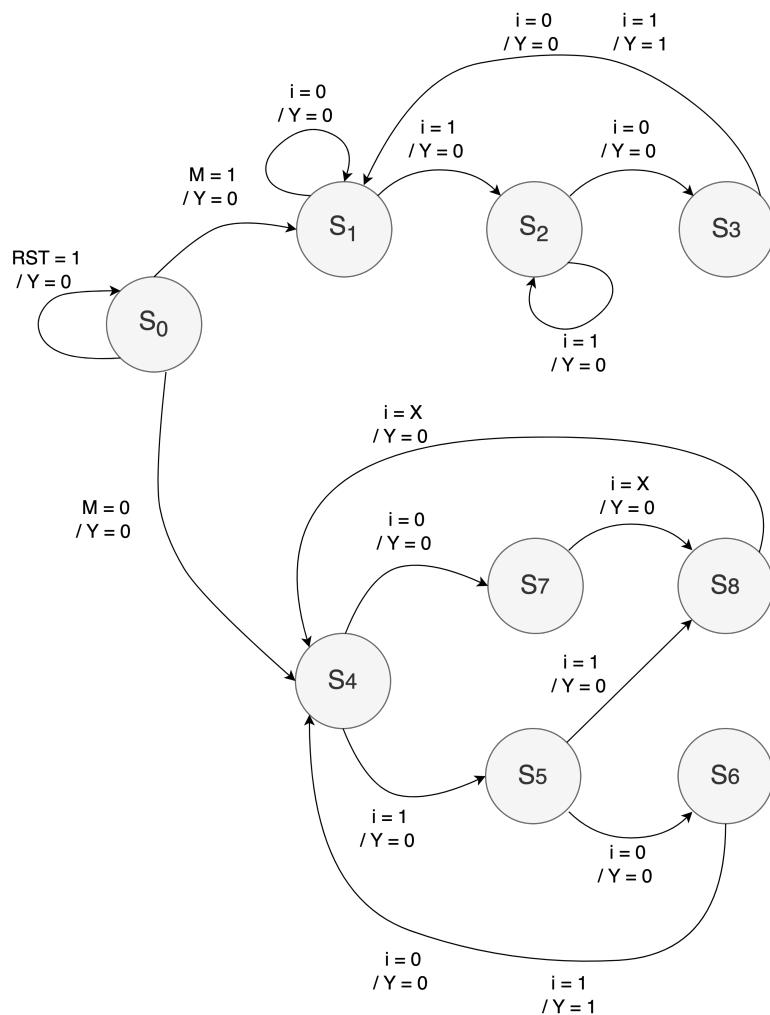


Figura 3.1: Automa riconoscitore di sequenza

3.1.2 Implementazione

Per l'implementazione VHDL dell'automa, si dichiarano dapprima gli ingressi

- RST: permette il reset della macchina, portandola allo stato S_0 ;
- A: rappresenta l'abilitazione, ovvero il clock;
- i: è l'ingresso;

- M: permette di selezionare con quale modalità far lavorare la macchina: se $M = 0$ effettua il riconoscimento a gruppi di tre bit per volta; se $M = 1$ effettua il riconoscimento un bit alla volta

L'uscita è rappresentata dal segnale Y.

L'architettura è costruita con un approccio comportamentale e vi è una variazione di stato ad ogni fronte di salita del clock (A).

Si vuole notare che il segnale RST è sincrono.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity riconoscitore is
5      port
6      (
7          RST:    in  std_logic;
8          i:      in  std_logic;
9          A:      in  std_logic;
10         M:      in  std_logic;
11         Y:      out std_logic
12     );
13 end riconoscitore;
14
15 architecture Behavioral of riconoscitore is
16     type state_type is (S0, S1, S2, S3, S4, S5 , S6, S7, S8);
17     signal current_state, next_state: state_type;
18     signal temp_y: std_logic;
19
20 begin
21     process(A, RST)
22     begin
23         if rising_edge(A) then                      -- (A'event and A='1')
24             if RST = '1' then
25                 current_state  <=  S0;
26                 Y           <=  '0';
27             else
28                 current_state  <=  next_state;

```

```
29          Y           <= temp_y;
30      end if;
31  end if;
32 end process;
33
34 process (current_state, i, M)
35 begin
36     next_state           <= current_state;
37     --Y                 <= '0';
38
39     case current_state is
40     when S0 =>
41         if M = '1' then
42             next_state       <= S1;
43             temp_y          <= '0';
44         elsif M = '0' then
45             next_state       <= S4;
46             temp_y          <= '0';
47         end if;
48     when S1 =>
49         if i = '0' then
50             next_state       <= current_state;
51             temp_y          <= '0';
52         elsif i = '1' then
53             next_state       <= S2;
54             temp_y          <= '0';
55         end if;
56     when S2 =>
57         if i = '1' then
58             next_state       <= current_state;
59             temp_y          <= '0';
60         elsif i = '0' then
61             next_state       <= S3;
62             temp_y          <= '0';
63         end if;
64     when S3 =>
65         next_state       <= S1;
66         if i = '0' then
67             temp_y          <= '0';
68         elsif i = '1' then
69             temp_y          <= '1';
70         end if;
71
72     when S4 =>
73         if i = '0' then
```

```
74          next_state      <=  S7;
75          temp_y        <=  '0';
76      elsif i = '1' then
77          next_state      <=  S5;
78          temp_y        <=  '0';
79      end if;
80
81      when S5 =>
82          if i = '1' then
83              next_state      <=  S8;
84              temp_y        <=  '0';
85          elsif i = '0' then
86              next_state      <=  S6;
87              temp_y        <=  '0';
88          end if;
89      when S6 =>
90          next_state      <=  S4;
91          if i = '1' then
92              temp_y        <=  '1';
93          end if;
94      when S7 =>
95          next_state      <=  S8;
96          temp_y        <=  '0';
97      when S8 =>
98          next_state      <=  S4;
99          temp_y        <=  '0';
100     end case;
101 end process;
102 end Behavioral;
```

Code 3.1: riconoscitore.vhdl

3.1.3 Simulazione

Per effettuare la simulazione, è stato necessario il seguente testbench.

```

1  -- Testbench for riconoscitore (sequence 101 detection)
2  library IEEE;
3  use IEEE.Std_logic_1164.all;
4  use IEEE.Numeric_Signed.all;
5
6  entity riconoscitore_tb is
7  end;
8
9  architecture bench of riconoscitore_tb is
10
11  component riconoscitore
12    port
13    (
14      RST:     in  std_logic;
15      i:       in  std_logic;    -- Input signal
16      A:       in  std_logic;    -- Clock signal
17      M:       in  std_logic;    -- Mode or another input (adjust
18      --> as needed)
19      Y:       out std_logic   -- Output signal (detects "101")
20    );
21  end component;
22
23  signal RST: std_logic := '0';
24  signal i: std_logic := '0';
25  signal A: std_logic := '0';  -- Clock
26  signal M: std_logic := '0';
27  signal Y: std_logic;
28
29  constant clock_period: time := 10 ns;
30  signal stop_the_clock: boolean := false;
31
32 begin
33
34  uut: riconoscitore port map (
35    RST => RST,
36    i    => i,
37    A    => A,
38    M    => M,
39    Y    => Y
40  );
41
42  -- Clock generation
43  clocking: process
44  begin
45    while not stop_the_clock loop

```

```
45      A <= '0';
46      wait for clock_period/2;
47      A <= '1';
48      wait for clock_period/2;
49      end loop;
50      wait;
51  end process;
52
53  -- Stimulus process
54  stimulus: process
55  begin
56      -- Initialization
57      RST <= '1';
58      wait for 2 * clock_period; -- Hold reset for 2 clock cycles
59      RST <= '0';
60      wait for clock_period;
61
62      M <= '1';
63      wait for 2 * clock_period;
64      i <= '1';
65      wait for clock_period;
66      i <= '1';
67      wait for clock_period;
68      i <= '0';
69      wait for clock_period;
70      i <= '1';
71      wait for clock_period;
72      i <= '0';
73      wait for clock_period;
74      i <= '0';
75      wait for clock_period;
76      i <= '1';
77      wait for clock_period;
78      i <= '0';
79      wait for clock_period;
80      i <= '1';
81      wait for clock_period;
82
83      RST <= '1';
84      wait for 2 * clock_period;
85      RST <= '0';
86      wait for clock_period;
87
88      M <= '0';
89      wait for 2 * clock_period;
```

```
90      i <= '1';
91      wait for clock_period;
92      i <= '1';
93      wait for clock_period;
94      i <= '0';
95      wait for clock_period;
96      i <= '1';
97      wait for clock_period;
98      i <= '0';
99      wait for clock_period;
100     i <= '1';
101     wait for clock_period;
102     i <= '1';
103     wait for clock_period;
104     i <= '0';
105     wait for clock_period;
106     i <= '1';
107     wait for clock_period;

108

109
-- End simulation
110 stop_the_clock <= true;
111 wait;
112 end process;
113
114
115 end bench;
```

Code 3.2: riconoscitore_tb.vhdl

Gli ingressi sono i seguenti:

- $M = 1$:
 - 1, 1, 0, 1, 0, 0, 1, 0, 1
- $M = 0$:
 - 1, 1, 0, 1, 0, 1, 1, 0, 1

Il risultato è il seguente:

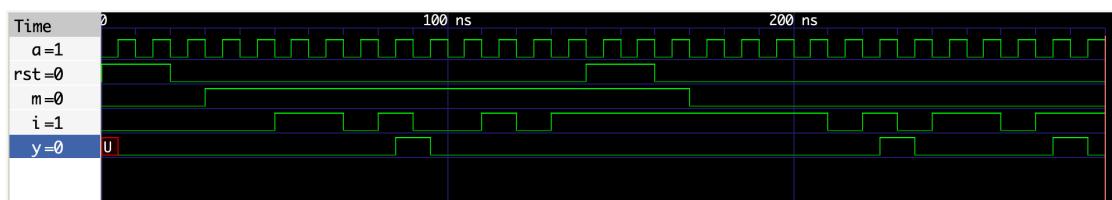


Figura 3.2: Simulazione Riconoscitore

3.2 Implementazione su board del punto precedente

DA SCRIVERE

Capitolo 4

Esercizio 4

4.1 Shift Register - Approccio comportamentale

Si vuole implementare uno Shift Register con approccio comportamentale, la cui dimensione è N .

Tale macchina ha come ingresso un valore y con il quale si può scegliere di fare shift verso destra o sinistra e di fare shift di uno o due bit.

4.1.1 Progetto e architettura

La macchina da implementare è la seguente:

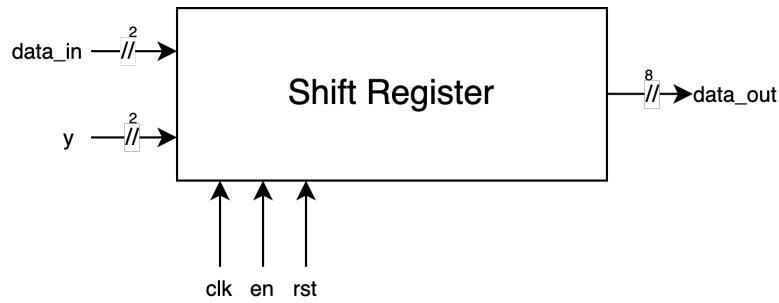


Figura 4.1: Shift Register

Dall’immagine si può notare che è stato scelto $N = 8$.

Gli ingressi sono i seguenti:

- `clk`: il clock, necessario per la temporizzazione. La macchina lavorerà sul fronte di salita;
- `en`: l’abilitazione, la quale permette di abilitare o disabilitare la macchina;
- `rst`: reset sincrono della macchina;
- `y`: vettore di 2 elementi che sceglie la modalità di funzionamento della macchina; in particolare:
 - $y = 00$: shift a sinistra di 1;
 - $y = 01$: shift a sinistra di 2;
 - $y = 10$: shift a destra di 1;
 - $y = 11$: shift a destra di 2;

- `data_in`: rappresenta i dati in ingresso; esso è un vettore di due elementi poiché quando vi è la necessità di fare uno shift di 2, si ha bisogno di due bit

L'uscita della macchina è `data_out`, vettore di 8 bit.

4.1.2 Implementazione

L'implementazione è la seguente

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity shift_register_beh is
6     generic ( N: integer := 8 );      --lunghezza registro
7
8     port
9     (
10         clk      : in std_logic;
11         rst      : in std_logic;
12         en       : in std_logic;
13         y        : in std_logic_vector(1 downto 0);
14         data_in : in std_logic_vector(1 downto 0);
15         data_out: out std_logic_vector(N-1 downto 0)
16     );
17 end entity;
18
19 architecture behavioral of shift_register_beh is
20     signal reg  : std_logic_vector(N-1 downto 0);
21
22 begin
23     process(clk)
24     begin
25         if (rising_edge(clk)) then
26             if (rst = '1') then
27                 reg <= (others => '0');
28             end if;
29     end process;
30 end architecture;
```

```

30      if (en = '1') then
31          case y is
32              when "00" =>           --shift a sinistra di 1
33                  reg(7 downto 1) <= reg(6 downto 0);
34                  reg(0)           <= data_in(0);
35              when "01" =>           --shift a sinistra di 2
36                  reg(7 downto 2) <= reg(5 downto 0);
37                  reg(1)           <= data_in(1);
38                  reg(0)           <= data_in(0);
39              when "10" =>           --shift a destra di 1
40                  reg(6 downto 0) <= reg(7 downto 1);
41                  reg(7)           <= data_in(0);
42              when "11" =>           --shift a destra di 2
43                  reg(5 downto 0) <= reg(6 downto 1);
44                  reg(6)           <= data_in(0);
45                  reg(7)           <= data_in(1);
46              when others =>
47                  null;
48          end case;
49      end if;
50  end if;
51 end process;
52
53 data_out <= reg;
54 end behavioral;

```

Code 4.1: shift_register_beh.vhdl

4.1.3 Simulazione

Per effettuare la simulazione, si utilizza il seguente testbench:

```

1 library IEEE;
2 use IEEE.Std_logic_1164.all;
3 use IEEE.Numeric_Signed.all;
4
5 entity shift_register_beh_tb is
6 end;
7
8 architecture bench of shift_register_beh_tb is
9

```

```

10   component shift_register_beh
11     generic ( N: integer := 8 );
12     port
13     (
14       clk      : in std_logic;
15       rst      : in std_logic;
16       en       : in std_logic;
17       y        : in std_logic_vector(1 downto 0);
18       data_in : in std_logic_vector(1 downto 0);
19       data_out: out std_logic_vector(N-1 downto 0)
20     );
21   end component;
22
23   -- Signal declarations
24   signal clk      : std_logic := '0';
25   signal rst      : std_logic := '0';
26   signal en       : std_logic := '0';
27   signal y        : std_logic_vector(1 downto 0) := "00";
28   signal data_in : std_logic_vector(1 downto 0) := "00";
29   signal data_out: std_logic_vector(7 downto 0);
30
31   constant clock_period: time := 10 ns;
32   signal stop_the_clock: boolean := false;
33
34 begin
35
36   -- Instanziazione del componente sotto test
37   uut: shift_register_beh
38     generic map ( N => 8 ) -- Dimensione del registro
39     port map (
40       clk      => clk,
41       rst      => rst,
42       en       => en,
43       y        => y,
44       data_in  => data_in,
45       data_out => data_out
46     );
47
48   -- Stimoli
49   stimulus: process
50   begin
51     -- Reset iniziale
52     rst <= '1';
53     wait for clock_period;
54     rst <= '0';

```

```
55      wait for clock_period;
56
57      -- Abilitazione e shift a sinistra di 1
58      en <= '1';
59      y <= "00";
60      data_in <= "01";  -- Input dati
61      wait for clock_period;
62
63      -- Shift a sinistra di 2
64      y <= "01";
65      data_in <= "10";
66      wait for clock_period;
67
68      -- Shift a destra di 1
69      y <= "10";
70      data_in <= "11";
71      wait for clock_period;
72
73      -- Shift a destra di 2
74      y <= "11";
75      data_in <= "01";
76      wait for clock_period;
77
78      -- Disabilitazione
79      en <= '0';
80      data_in <= "00";
81      wait for clock_period;
82
83      -- Concludi la simulazione
84      stop_the_clock <= true;
85      wait;
86  end process;
87
88  -- Processo di generazione del clock
89  clocking: process
90  begin
91      while not stop_the_clock loop
92          clk <= '0', '1' after clock_period / 2;
93          wait for clock_period;
94      end loop;
95      wait;
96  end process;
97
98 end;
```

Code 4.2: shift_register_beh_tb.vhdl

Il risultato della simulazione è il seguente:

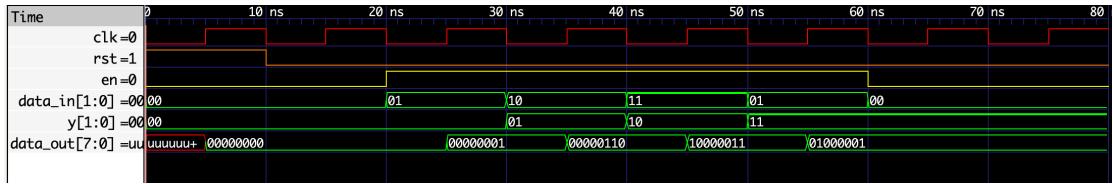


Figura 4.2: Simulazione Shift Register con approccio comportamentale

Si può facilmente notare dall'immagine che la macchina lavora come desiderato: ad ogni fronte di salita del clock e quando l'abilitazione è alta, in base alla modalità di lavoro, shifta a destra o a sinistra, di uno o due bit.

4.2 Shift Register - Approccio strutturale

Si vuole riprogettare la macchina precedente, figura 4.1, utilizzando un approccio strutturale.

Le componenti della macchina sono 8 registri da 1 bit e 8 mux 4:1. Si sono scelti 8 registri poiché in tale esempio si realizza un registro da 8 bit ($N = 8$).

4.2.1 Progetto e architettura

Registro da un bit

Il primo componente necessario è il registro da un bit.

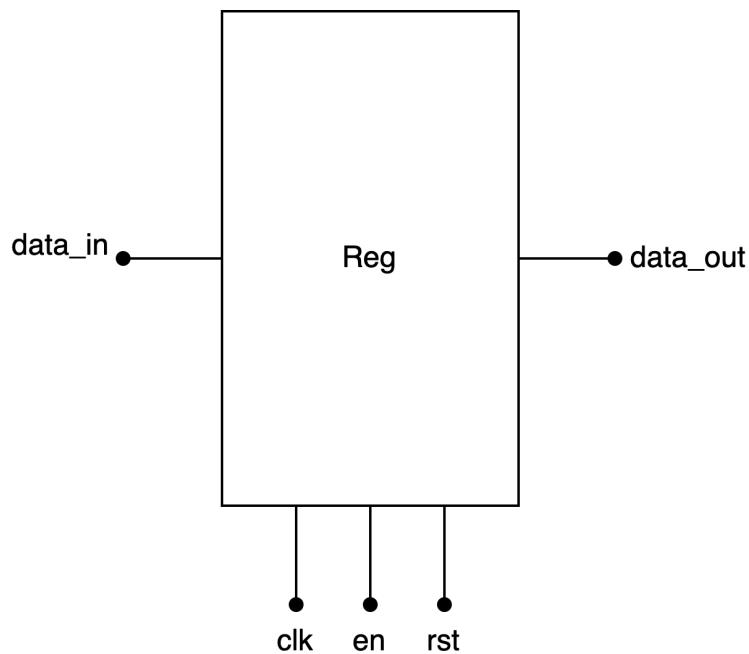


Figura 4.3: Registro da 1 bit

Gli ingressi di tale componente sono i seguenti:

- `data_in`: bit in ingresso, che verrà memorizzato nel registro;
- `clk`: il clock per la temporizzazione; il registro lavora sul fronte di salita di quest'ultimo;
- `en`: segnale di abilitazione; il registro memorizza il bit in ingresso solo quando tale segnale è alto;
- `rst`: segnale che quando è alto resetta il registro, portando il valore al suo interno a 0; il reset è sincrono.

La sua uscita è `data_out`, che altro non rappresenta il bit memorizzato nel registro.

Mux 4:1

Il secondo componente è il mux 4:1.

Tramite quest'ultimo si decide qual è l'ingresso di un registro, attraverso la selezione.

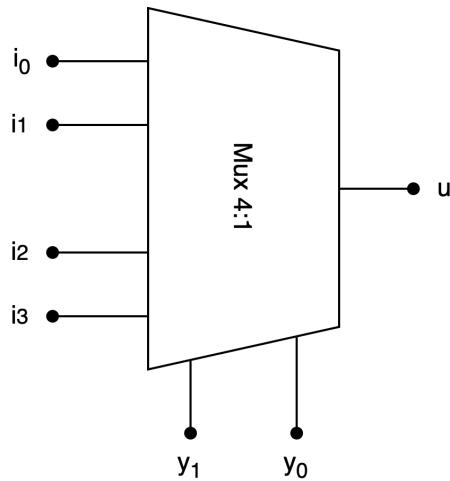


Figura 4.4: Mux 4:1

Tale multiplexer lavora seguendo la seguente tabella: Si può ban-

y₁	y₀	u
0	0	i ₀
0	1	i ₁
1	0	i ₂
1	1	i ₃

Tabella 4.1: Tabella di verità del multiplexer 4:1 per lo Shift Register

nalmente notare che l'uscita altro non è che uno dei 4 ingressi del multiplexer, scelto tramite la selezione.

Si può ora comporre lo Shift Register.

Facendo gli opportuni collegamenti, si ottiene il seguente schema

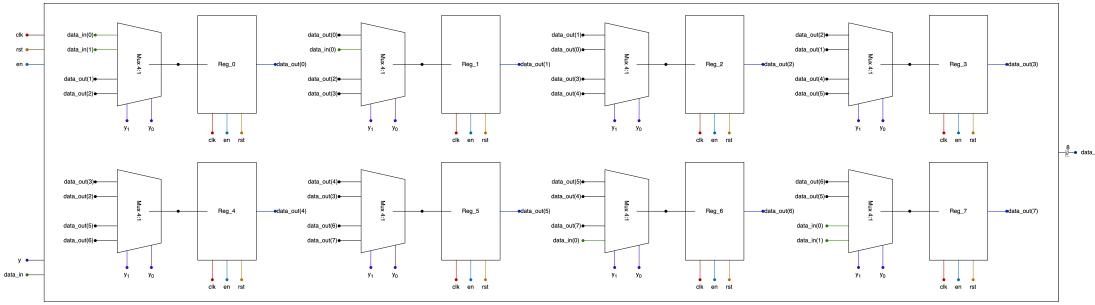


Figura 4.5: Shift Register con approccio strutturale

Gli ingressi e l'uscita dello Shift Register sono identici a quelli visti nell'esercizio precedente.

Quello che si vuole mettere in evidenza in questo caso è come tali ingressi siano collegati con le strutture interni: in particolare si vede che y , che sceglie la modalità di lavoro della macchina, è collegato alla selezione dei multiplexer, mentre `data_in`, è collegato solo ai primi due e agli ultimi due multiplexer.

I restanti sono collegati direttamente ai registri.

4.2.2 Implementazione

Si vuole ora procedere con l'implementazione in VHDL.

Partendo dal registro, si ha:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity reg is
6     port
7     (
8         data_in:      in  std_logic;
9         en:          in  std_logic;

```

```
10      clk:      in  std_logic;
11      rst:      in  std_logic;
12      data_out:  out std_logic
13  );
14 end entity;
15
16 architecture Behavioral of reg is
17 begin
18     process(clk)
19     begin
20         if(rising_edge(clk))then
21             if (rst = '1') then
22                 data_out <= '0';
23             end if;
24
25             if (en = '1') then
26                 data_out <= data_in;
27             end if;
28         end if;
29     end process;
30 end architecture;
```

Code 4.3: register.vhdl

Si prosegue con il multiplexer 4:1

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux_41 is
5     port
6     (
7         i: in std_logic_vector(0 to 3);
8         y: in std_logic_vector(1 downto 0);
9         output: out std_logic
10    );
11 end entity;
12
13 architecture Behavioral of mux_41 is
14 begin
15     process(y, i)
16     begin
17         case y is
```

```

18      when "00" =>
19          output <= i(0);
20      when "01" =>
21          output <= i(1);
22      when "10" =>
23          output <= i(2);
24      when "11" =>
25          output <= i(3);
26      when others =>
27          output <= '-';
28  end case;
29 end process;
30 end architecture;

```

Code 4.4: mux_4_1.vhdl

Implementate le componenti base per il progetto, si prosegue con lo Shift Register:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity shift_register is
5 port
6 (
7     clk:      in  std_logic;
8     rst:      in  std_logic;
9     en:       in  std_logic;
10    y:        in  std_logic_vector(1 downto 0);
11    data_in:   in  std_logic_vector(1 downto 0);
12    data_out:  out std_logic_vector(7 downto 0)
13 );
14 end entity;
15
16 architecture structural of shift_register is
17 signal in_mux:      std_logic_vector(0 to 31);
18 signal out_mux:     std_logic_vector(0 to 7);
19 signal temp_out:    std_logic_vector(0 to 7);
20
21 component reg is
22 port
23 (

```

```

24      data_in:    in  std_logic;
25      en:        in  std_logic;
26      clk:       in  std_logic;
27      rst:       in  std_logic;
28      data_out:   out std_logic
29  );
30 end component;
31
32 component mux_41 is
33     port
34     (
35         i:        in std_logic_vector(0 to 3);
36         y:        in std_logic_vector(1 downto 0);
37         output:   out std_logic
38     );
39 end component;
40
41 begin
42     data_out      <=  temp_out;
43
44     mux0to7: for k in 0 to 7 generate
45         m: mux_41
46         port map
47         (
48             in_mux(4*k to (k*4 + 3)),
49             y,
50             out_mux(k)
51         );
52     end generate;
53
54     reg0_to7: for k in 0 to 7 generate
55         r: reg
56         port map
57         (
58             out_mux(k),
59             en,
60             clk,
61             rst,
62             temp_out(k)
63         );
64     end generate;
65
66     --assignmet input signals for mux
67     --mux_0
68     in_mux(0)    <=  data_in(0);

```

```
69      in_mux(1)    <=  data_in(1);
70      in_mux(2)    <=  temp_out(1);
71      in_mux(3)    <=  temp_out(2);
72
73      --mux_1
74      in_mux(4)    <=  temp_out(0);
75      in_mux(5)    <=  data_in(0);
76      in_mux(6)    <=  temp_out(2);
77      in_mux(7)    <=  temp_out(3);
78
79      --mux_2 to mux_5
80
81      conn_mux2to5: for k in 2 to 5 generate
82          in_mux(4*k)      <=  temp_out(k-1);
83          in_mux(4*k + 1)  <=  temp_out(k-2);
84          in_mux(4*k + 2)  <=  temp_out(k+1);
85          in_mux(4*k + 3)  <=  temp_out(k+2);
86      end generate;
87
88      --mux_6
89      in_mux(24)    <=  temp_out(5);
90      in_mux(25)    <=  temp_out(4);
91      in_mux(26)    <=  temp_out(7);
92      in_mux(27)    <=  data_in(0);
93
94      --mux_7
95      in_mux(28)    <=  temp_out(6);
96      in_mux(29)    <=  temp_out(5);
97      in_mux(30)    <=  data_in(0);
98      in_mux(31)    <=  data_in(1);
99
100 end structural;
```

Code 4.5: shift_register.vhdl

Si può notare come nella architettura, sono state prima generate le componenti e solo dopo sono stati effettuati i vari collegamenti, utilizzando variabili ausiliarie.

4.2.3 Simulazione

Per effettuare la simulazione, si implementa il seguente testbench

```

1  library IEEE;
2  use IEEE.Std_logic_1164.all;
3  use IEEE.Numeric_Std.all;
4
5  entity shift_register_tb is
6  end;
7
8  architecture bench of shift_register_tb is
9
10 component shift_register
11 port
12 (
13     clk:      in  std_logic;
14     rst:      in  std_logic;
15     en:       in  std_logic;
16     Y:        in  std_logic_vector(1 downto 0);
17     data_in:   in  std_logic_vector(1 downto 0);
18     data_out:  out std_logic_vector(7 downto 0)
19 );
20 end component;
21
22 signal clk: std_logic := '0';
23 signal rst: std_logic := '0';
24 signal en: std_logic := '0';
25 signal y: std_logic_vector(1 downto 0) := "00";
26 signal data_in: std_logic_vector(1 downto 0) := "00";
27 signal data_out: std_logic_vector(7 downto 0);
28
29 begin
30
31     uut: shift_register port map ( clk      => clk,
32                                     rst      => rst,
33                                     en       => en,
34                                     Y        => y,
35                                     data_in  => data_in,
36                                     data_out => data_out );
37
38     clk_process: process
39     begin
40         -- Clock generation

```

```

41      clk <= not clk after 10 ns;
42      wait for 10 ns;
43  end process;
44
45  stimulus: process
46  begin
47      -- Test Case 1: Apply reset
48      rst <= '1';                      -- Assert reset
49      wait for 20 ns;                  -- Wait for reset to propagate
50      rst <= '0';                      -- Deassert reset
51      wait for 20 ns;
52
53      -- Test Case 2: Enable shift register with y = "00"
54      en <= '1';                      -- Enable shift register
55      y <= "00";                      -- Set y to "00"
56      data_in <= "01";                -- Apply data "01"
57      wait for 40 ns;
58
59      -- Test Case 3: Apply data_in while changing y to "01"
60      y <= "01";                      -- Change y to "01"
61      data_in <= "10";                -- Apply data "10"
62      wait for 40 ns;
63
64      -- Test Case 4: Apply data_in while changing y to "10"
65      y <= "10";                      -- Change y to "10"
66      data_in <= "11";                -- Apply data "11"
67      wait for 40 ns;
68
69      -- Test Case 5: Disable shift register with y = "11"
70      en <= '0';                      -- Disable shift register
71      y <= "11";                      -- Set y to "11"
72      data_in <= "00";                -- Change data input
73      wait for 40 ns;
74
75      -- Test Case 6: Apply data_in with y = "00" and reset active
76      rst <= '1';                      -- Assert reset
77      y <= "00";                      -- Set y to "00"
78      data_in <= "10";                -- Apply data "10" while reset is active
79      wait for 20 ns;
80      rst <= '0';                      -- Deassert reset
81      wait for 40 ns;
82
83      -- Test Case 7: Apply data_in while y = "01" and enable shifting
84      y <= "01";                      -- Set y to "01"
85      en <= '1';                      -- Enable shift register

```

```

86      data_in <= "11";      -- Apply data "11"
87      wait for 40 ns;
88      data_in <= "00";      -- Change data to "00"
89      wait for 40 ns;
90
91      -- Test Case 8: Apply reset while shifting and changing y
92      rst <= '1';          -- Assert reset
93      y <= "10";           -- Change y to "10"
94      data_in <= "11";      -- Apply data "11"
95      wait for 20 ns;
96      rst <= '0';          -- Deassert reset
97      wait for 40 ns;
98
99      -- Test Case 9: Apply y = "11" and shift continuously with
100     → data_in changes
101     y <= "11";           -- Set y to "11"
102     en <= '1';            -- Enable shift register
103     data_in <= "00";      -- Apply data "00"
104     wait for 40 ns;
105     data_in <= "01";      -- Apply data "01"
106     wait for 40 ns;
107     data_in <= "10";      -- Apply data "10"
108     wait for 40 ns;
109     data_in <= "11";      -- Apply data "11"
110     wait for 40 ns;
111
112     -- Test Case 10: Reset during shifting with y = "00"
113     rst <= '1';          -- Assert reset during shift
114     y <= "00";           -- Set y to "00"
115     data_in <= "01";      -- Apply data "01"
116     wait for 20 ns;
117     rst <= '0';          -- Deassert reset
118     wait for 40 ns;
119
120     -- Test Case 11: Changing y while shifting with enabled register
121     y <= "01";           -- Set y to "01"
122     en <= '1';            -- Enable shift register
123     data_in <= "01";      -- Apply data "01"
124     wait for 40 ns;
125     y <= "10";           -- Change y to "10"
126     data_in <= "11";      -- Change data to "11"
127     wait for 40 ns;
128
129     -- Test Case 12: Apply y = "11" and shift with reset active
130     rst <= '1';          -- Assert reset

```

```

130      y <= "11";           -- Set y to "11"
131      data_in <= "10";    -- Apply data "10"
132      wait for 20 ns;
133      rst <= '0';        -- Deassert reset
134      wait for 40 ns;

135
136      -- Test Case 13: Disable shifting during changes in y
137      en <= '0';          -- Disable shift register
138      y <= "00";          -- Set y to "00"
139      data_in <= "11";    -- Change data to "11"
140      wait for 40 ns;
141      y <= "01";          -- Change y to "01"
142      data_in <= "00";    -- Change data to "00"
143      wait for 40 ns;

144
145      -- End
146  end process;
147
148 end;

```

Code 4.6: tb_shift_register.vhd

Lanciando la simulazione, il risultato è il seguente:

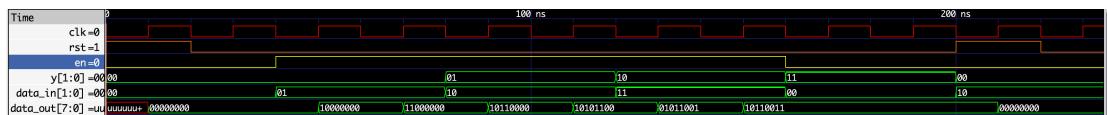


Figura 4.6: Simulazione dello Shift Register con approccio strutturale

Si nota chiaramente un corretto funzionamento della macchina.

Capitolo 5

Esercizio 5

5.1 Cronometro

5.2 Implementazione su board del punto precedente

Capitolo 6

Esercizio 6

6.1 Sistema di lettura - elaborazione - scrittura PO__PC

6.1.1 Traccia

Progettare, implementare in VHDL e verificare mediante simulazione un sistema dotato di una memoria ROM di N locazioni da 8 bit ciascuna, una macchina combinatoria M in grado di trasformare (secondo una funzione a scelta dello studente) la stringa di 8 bit letta dalla ROM in una stringa di 4 bit, e una memoria MEM di N locazioni che memorizza la stringa in output da M. Il sistema si avvia in corrispondenza di un segnale di START che viene fornito esternamente. Una volta avviato, tramite un'apposita unità di controllo che gestisce la temporizzazione del sistema, viene scandita una locazione

alla volta della ROM e viene scritta la corrispondente locazione di MEM. Gli indirizzi di memoria sono forniti da un contatore. Le memorie ROM e MEM hanno rispettivamente un read e un write sincrono.

6.1.2 Progettazione

Per il progetto di questo sistema si riprende l'esercizio nel capitolo 2: **Sistema ROM + M**. Viene usato anche un contatore, per scandire una alla volta tutte le locazioni della ROM da cui prelevare la stringa contenente 8 bit. Come nell'esercizio precedente, l'uscita della ROM viene posta in ingresso alla macchina M, che somma i 4 bit più significativi dell'ingresso con i 4 bit meno significativi. L'uscita della macchina M viene posto in ingresso a una memoria MEM e poi caricato nella locazione corrispondente all'uscita del contatore; il funzionamento del sistema viene gestito da un'unità di controllo. La struttura del sistema sarà fatta in questo modo:

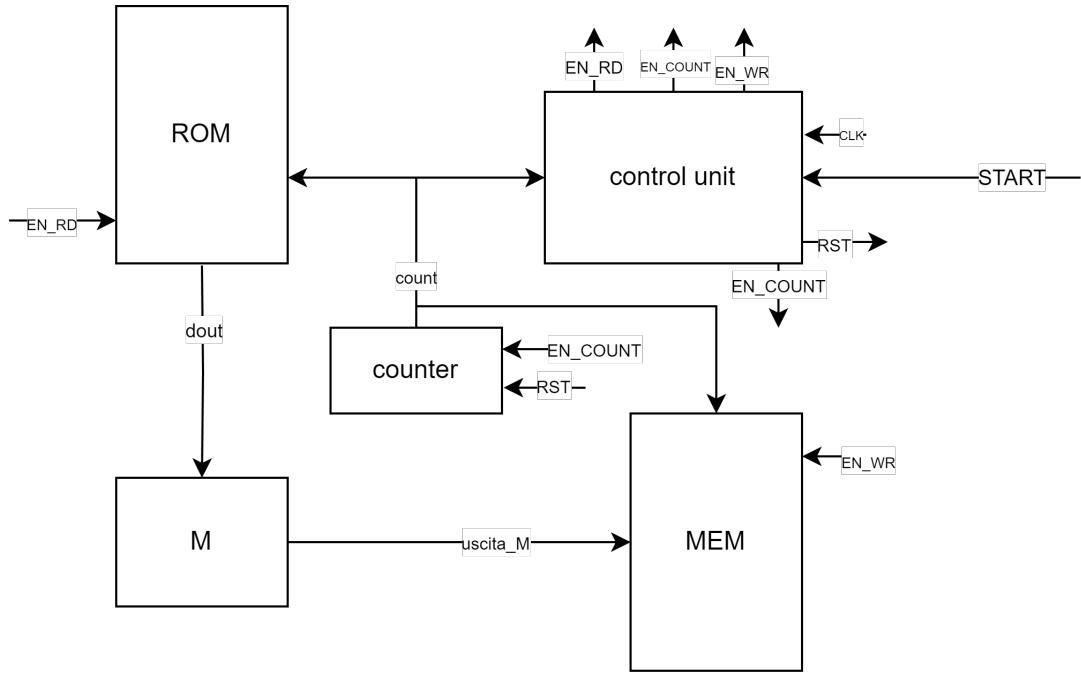


Figura 6.1: Schema a blocchi del sistema ROM + M + MEM

L'unità di controllo (CU) può essere efficacemente modellata come una macchina a stati finiti (FSM). Nel caso particolare avremo i seguenti stati:

- **idle**
- **read**
- **m_work**
- **write**

L'unità di controllo può essere quindi rappresentata da un automa come si mostra in figura:

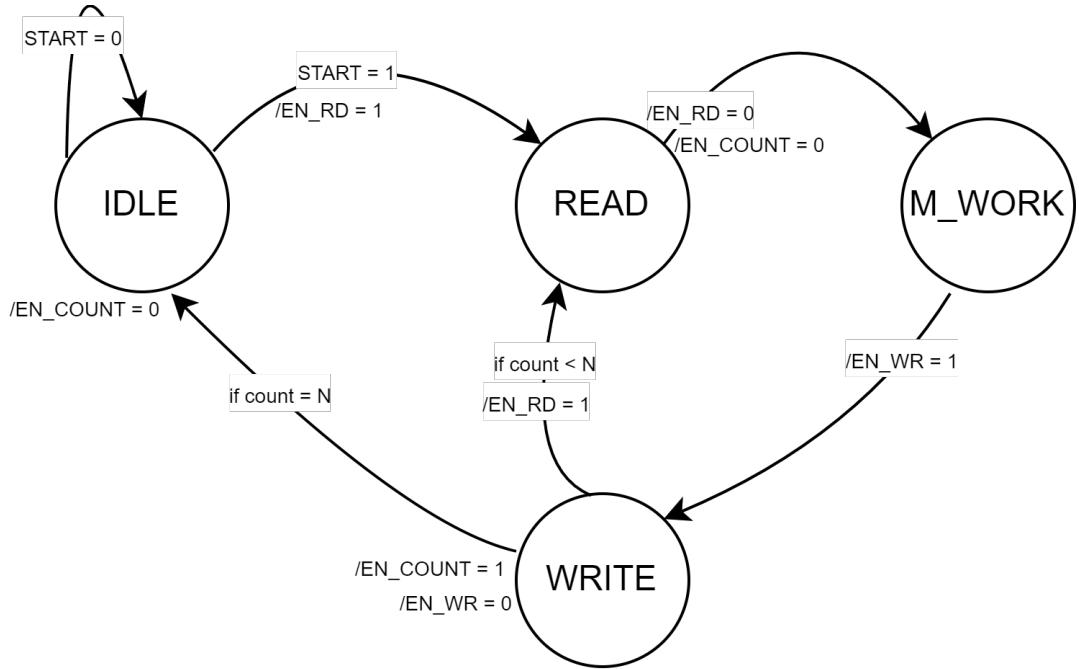


Figura 6.2: Macchina a stati della control unit di ROM + M + MEM

6.1.3 Implementazione

Il codice implementativo di M resta invariato, in quanto macchina puramente combinatoria. Si nota che viene richiesto che le operazioni di Read dalla ROM e di Write sulla memoria MEM siano svolte in modo sincrono. Quindi, a differenza della ROM usata nell'esercizio precedente, che era puramente combinatoria, le operazioni di lettura di questa ROM avvengono in sincronia con un segnale di clock. Questo segnale fornisce un riferimento temporale preciso per tutte le operazioni interne della ROM, garantendo così un funzionamento coerente e affidabile. Inoltre, sono stati utilizzati dei segnali di abilitazione alla lettura e alla scrittura, in modo da evitare conflitti e da permettere che i dati vengano letti al momento giusto. Si mostra

innanzitutto il nuovo codice di ROM:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity ROM is
6     generic(N: integer range 0 to 32:= 16);
7     port(
8         CLK: in STD_LOGIC; --read sincrona
9         address: in STD_LOGIC_VECTOR(3 downto 0); --l'indirizzo in
10        → ingresso viene dal contatore
11        EN_RD: in STD_LOGIC;
12        dout: out STD_LOGIC_VECTOR(7 downto 0)
13    );
14 end entity ROM;
15
16
17 architecture Behavioral of ROM is
18
19 type MEMORY is array(N-1 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
20        → --memoria da N locazioni che contengono 8 bit
21 constant ROM_N: MEMORY := (
22     "01000000", -- in locazione 15
23     "01000001",
24     "01000010",
25     "01000011",
26     "00010100",
27     "01000101",
28     "00000110",
29     "01000111",
30     "00001000",
31     "00001001",
32     "01001010",
33     "00001011",
34     "00001100",
35     "00001101",
36     "10001010",
37     "00001001" --in locazione 0
38 );
39
40 begin
41
42 lettura: process(EN_RD, address, CLK)

```

```
41 begin
42     if (CLK'event AND CLK = '1') then
43         if (EN_RD = '1') then
44             dout<= ROM_N(TO_INTEGER(unsigned(address))); --lettura dalla
45             --rom
46         end if;
47     end if;
48 end process;
49
50
51 end architecture Behavioral;
```

Code 6.1: ROM.vhdl

Si è scelto di utilizzare le stesse stringhe dell'esercizio 2 per "popolare" la ROM, in modo da poter confrontare i risultati. Si mostra ora l'implementazione dell'unità di controllo del sistema; tale codice gestisce i cambiamenti di stato, e si può considerare il "cervello" del sistema in esame.

Si noti che si è scelto di porre in uscita gli stati, in modo da visualizzare in simulazione anche le variazioni di stato, è una scelta ovviamente facoltativa, ma per ragione di debugging è stato scelto di visualizzare anche la variazione di stato, come sarà visibile dalla waveform nella prossima sezione.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity control_unit is
5     Port (
6         START, RST: in STD_LOGIC;
7         CLK: in STD_LOGIC;
8         count: in STD_LOGIC_VECTOR(3 downto 0);
9         stato: out STD_LOGIC_VECTOR(1 downto 0);
```

```

10          EN_RD, EN_WR, EN_COUNT: out STD_LOGIC
11      );
12  end control_unit;
13
14 architecture Behavioral of control_unit is
15 type stati is (idle, read, m_work, write);
16 signal current_state: stati;
17 signal next_state: stati;
18
19 begin
20
21 reg_stato: process(CLK, RST)
22 begin
23     if (CLK'event AND CLK = '1') then
24         if RST = '1' then
25             current_state <= idle;
26         else
27             current_state <= next_state;
28         end if;
29         end if;
30     end process;
31
32 change: process(CLK, START, count)
33 begin
34     CASE current_state is
35         WHEN idle =>
36             EN_COUNT <= '0';
37             if (START = '0') then
38                 next_state <= idle;
39             else
40                 EN_RD <= '1';
41                 next_state <= read;
42             end if;
43             WHEN read =>
44                 EN_RD <= '0';
45                 EN_COUNT <= '0';
46                 next_state <= M_work;
47             WHEN M_work =>
48                 EN_WR <= '1';
49                 next_state <= write;
50             WHEN write =>
51                 EN_WR <= '0';
52                 EN_COUNT <= '1';
53                 if (count = "1111") then
54                     next_state <= idle;

```

```

55         else
56             EN_RD <= '1';
57             next_state <= read;
58         end if;
59     end CASE;
60
61 end process;
62
63 stato <= "00" when current_state = idle else
64     "01" when current_state = read else
65     "10" when current_state = m_work else
66     "11" when current_state = write; -- Associa a ogni stato
       ↳ un codice binario
67
68 end Behavioral;

```

Code 6.2: control unit.vhdl

Si mostra infine il codice sistema nel suo complesso, composto dall'unità di controllo e da tutte le altre componenti utilizzate; è stato utilizzato un approccio di tipo strutturale:

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Rom_M_MEM is
5  generic(N: integer range 0 to 32:=16);
6  Port (
7      START, RST: in std_logic;
8      CLK: in std_logic;
9      stato: out STD_LOGIC_VECTOR(1 downto 0);
10     count: out STD_LOGIC_VECTOR(3 downto 0);
11     Y: out std_logic_vector(3 downto 0)
12   );
13 end Rom_M_MEM;
14
15 architecture structural of Rom_M_MEM is
16  --segnali intermedi:
17  signal Yc: std_logic_vector(3 downto 0);
18  signal yROM : std_logic_vector(7 downto 0); --uscita della ROM di 8
       ↳ bit

```

```

19  signal YM: std_logic_vector(3 downto 0); --uscita dalla macchina di
   ↵ trasform di 4 bit
20  signal ENABLE, WRITE, READ: std_logic;
21
22 component ROM is
23     port(
24         CLK: in STD_LOGIC;
25         address: in STD_LOGIC_VECTOR(3 downto 0);
26         EN_RD: in STD_LOGIC;
27         dout: out STD_LOGIC_VECTOR(7 downto 0)
28     );
29 end component;
30
31 component M is
32     port(
33         ingresso: in std_logic_vector(7 downto 0);
34         uscita: out std_logic_vector(3 downto 0)
35     );
36 end component;
37
38 component MEM
39     port(
40         CLK: in std_logic;
41         EN_WR: in std_logic;
42         ADD: in std_logic_vector(3 downto 0);
43         DATA_IN: in std_logic_vector(3 downto 0)
44     );
45 end component;
46
47 component cont_mod16 is
48     Port (
49         CLK: in std_logic;
50         RST: in std_logic;
51         EN_COUNT: in std_logic;
52         count: out std_logic_vector(3 downto 0)
53     );
54 end component;
55
56 component control_unit
57     Port (
58         START, RST: in STD_LOGIC;
59         CLK: in STD_LOGIC;
60         count: in STD_LOGIC_VECTOR(3 downto 0);
61         stato: out STD_LOGIC_VECTOR(1 downto 0);
62         EN_RD, EN_WR, EN_COUNT: out STD_LOGIC

```

```

63      );
64  end component;
65
66 begin
67
68 --collegamenti tra le componenti
69 ROM0: ROM
70     Port map(
71         CLK => CLK,
72         address => Yc,
73         EN_RD => READ,
74         dout => yROM
75     );
76 M0: M
77     Port map(
78         ingresso => yROM,
79         uscita => yM
80     );
81
82 MEM0: MEM
83     Port map(
84         CLK => CLK,
85         EN_WR => WRITE,
86         ADD => Yc,
87         DATA_IN => YM
88     );
89
90 cont: cont_mod16
91     port map(
92         CLK => CLK,
93         RST => RST,
94         EN_COUNT => ENABLE,
95         count => Yc --l'uscita del contatore deve andare in ingresso ad
96         ← address della ROM e della mem
97     );
98
99 cu: control_unit
100    port map(
101        START => START,
102        RST => RST,
103        CLK => CLK,
104        count => Yc,
105        stato => stato,
106        EN_RD => READ,
107        EN_WR => WRITE,

```

```

107         EN_COUNT => ENABLE
108     );
109
110     Y <= YM;
111     count <= YC;
112
113 end structural;

```

Code 6.3: ROM + M + MEM.vhdl

Si vuole porre l'attenzione allo Schematic generato dall'ambiente Vivado, che mostra chiaramente i collegamenti e le dipendenze tra tutte le componenti del sistema.

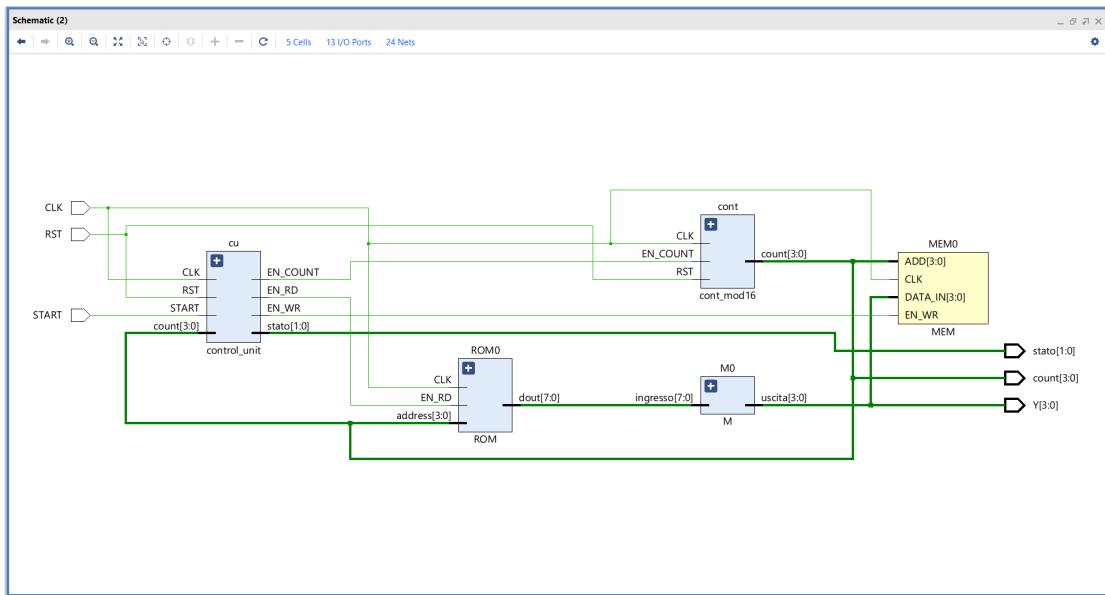


Figura 6.3: Schematic di ROM + M + MEM

6.1.4 Simulazione

Per procedere con la simulazione è stato necessario generare un testbench:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 -- Testbench for Rom_M_MEMORY entity
6
7 entity tb_Rom_M_MEMORY is
8 end entity;
9
10 architecture Behavioral of tb_Rom_M_MEMORY is
11
12 -- Signals for the testbench
13 signal CLK: std_logic := '0';
14 signal RST: std_logic := '1';
15 signal START: std_logic := '0';
16 signal stat0: std_logic_vector(1 downto 0);
17 signal count: STD_LOGIC_VECTOR(3 downto 0);
18 signal Y: std_logic_vector(3 downto 0);
19
20 begin
21
22 -- Clock generation
23 process
24 begin
25 wait for 5 ns;
26 CLK <= not CLK;
27 end process;
28
29 -- Test stimulus
30 process
31 begin
32 wait for 10 ns; -- Wait
33
34 -- Reset the system
35 RST <= '0';
36 wait for 10 ns;
37 RST <= '1';
38 wait for 10 ns;
39 RST <= '0';
40 wait for 10 ns;
41 -- Start the operation
42 START <= '1';
43 wait for 10 ns;
44 START <= '0';
45 wait for 10 ns;
```

```

46
47     wait for 100 ns;
48
49     -- End of simulation
50     wait;
51 end process;
52
53 uut: entity work.Rom_M_MEMORY
54 generic map (N => 16)
55 port map (
56     CLK => CLK,
57     RST => RST,
58     START => START,
59     stato => stato,
60     count => count,
61     Y => Y
62 );
63
64 end Behavioral;

```

Code 6.4: Testbench di ROM + M + MEM.vhd

Eseguendo la simulazione si avrà la seguente forma d'onda:

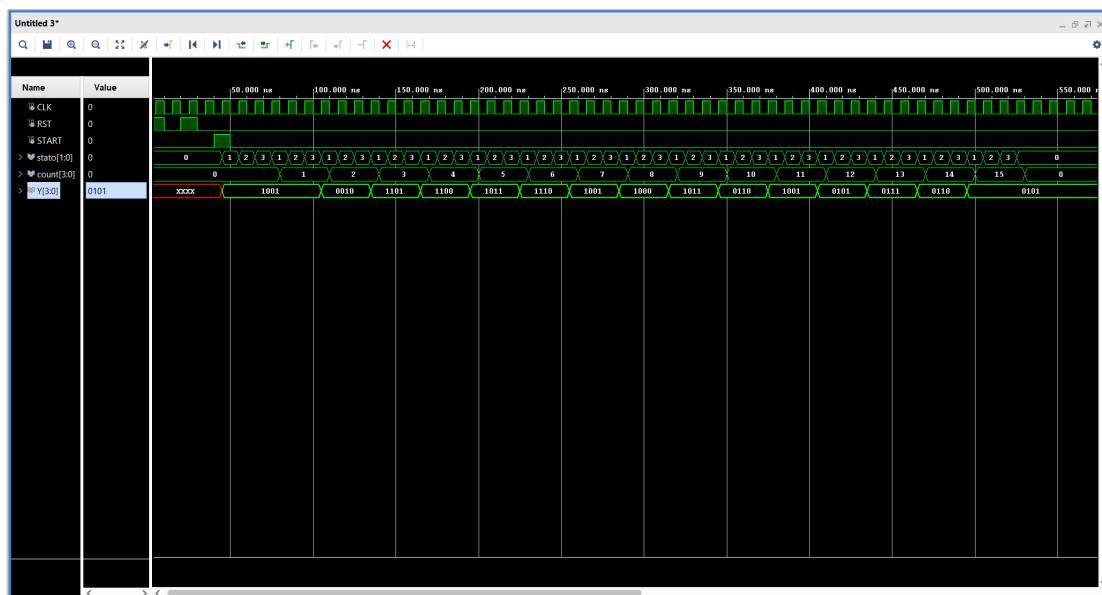


Figura 6.4: Waveform di ROM + M + MEM

Si analizzano alcuni casi per dimostrare la correttezza del sistema.

ma.

- **istante 0:** si accede alla locazione di memoria 0, in cui si trova la stringa "00001001", sommando i 4 bit più significativi con i 4 meno significativi si ottiene 1001;
- **istante 1:** si accede alla locazione di memoria 1, in cui si trova la stringa "10001010", all'uscita della macchina M si ottiene 0010;
- **istante 2:** si accede alla locazione di memoria 2, in cui si trova la stringa "00001101", procedendo come in precedenza si ottiene 1101.

Si può procedere in questo modo per tutte le locazioni di memoria scandite dal contatore confermando così il risultato della simulazione. Come detto, si è scelto di mostrare anche le variazioni dello stato e del contatore, in modo da avere la possibilità di osservare in ogni istante il comportamento del sistema, tale scelta è del tutto opzionale.

6.2 Implementazione su board del punto precedente

6.2.1 Traccia

Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando due bottoni per i segnali di read e

CAPITOLO 6. ESERCIZIO 6

reset rispettivamente e i led per la visualizzazione delle uscite della macchina istante per istante.

Bibliografia

- [1] Carlo Brandoles. *Introduzione al linguaggio VHDL - Aspetti teorici ed esempi di progettazione.* Politecnico di Milano.
- [2] Rocco di Torrepidula Franca e Somma Alessadra. *Architettura dei Sistemi di Elaborazione - Appunti tratti dalle lezioni del prof N. Mazzocca.*