

Tesina di
ARCHITETTURA DEI SISTEMI DIGITALI

Corso di Laurea
in
Ingegneria Informatica

By
Luca Antonio Scolletta
Erika Morelli
Andrea Bertolero
Nike Di Giacomo



UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

March, 2024

INDICE

INDICE	
CHAPTER 1 RETI COMBINATORIE ELEMENTARI	1
1.1 Esercizio 1.1 - Multiplexer 16:1	1
1.1.1 Progetto e architettura	1
1.1.2 Implementazione	3
1.1.3 Simulazione	6
1.2 Esercizio 1.2	10
1.2.1 Progetto e architettura	10
1.2.2 Implementazione	11
1.2.3 Simulazione	14
1.3 Esercizio 1.3	16
1.3.1 Progetto e architettura	16
1.3.2 Implementazione	17
1.3.3 Simulazione	20
1.4 Esercizio 2.1 - Sistema ROM+M	22
1.4.1 Progetto e architettura	22
1.4.2 Implementazione	23
1.4.3 Simulazione	27
1.5 Esercizio 2.2	29
1.5.1 Progetto e architettura	29
1.5.2 Implementazione	29
CHAPTER 2 RETI SEQUENZIALI ELEMENTARI	1
2.1 Esercizio 3.1 - Riconoscitore di sequenza	1

2.1.1	Progetto e architettura	1
2.1.2	Implementazione	3
2.1.3	Simulazione	6
2.2	Esercizio 3.2 - Riconoscitore di sequenza on board	11
2.2.1	Progetto e architettura	11
2.2.2	Implementazione	11
2.2.3	Timing Analysis	15
2.3	Esercizio 4.1 - Shift Register	17
2.3.1	Progetto e architettura	17
2.3.2	Implementazione	19
2.3.3	Simulazione	26
2.4	Esercizio 5.1 - Cronometro	32
2.4.1	Progetto e architettura	32
2.4.2	Implementazione	33
2.4.3	Simulazione	38
2.5	Esercizio 5.2 - Cronometro su board parte 1	40
2.5.1	Progetto e architettura	41
2.5.2	Implementazione	41
2.5.3	Timing Analysis	51
2.6	Esercizio 5.3 Cronometro su board parte 2	53
2.6.1	Progetto e architettura	53
2.6.2	Implementazione	54
2.7	Esercizio 6.1 - Sistema di lettura-elaborazione-scrittura PO PC	56
2.7.1	Progetto e architettura	57
2.7.2	Implementazione	59
2.7.3	Simulazione	72
2.8	Esercizio 6.2 - Implementazione su board del PO_PC	75
2.8.1	Progetto e architettura	75

CHAPTER 3 MACCHINE ARITMETICHE	1
3.1 Esercizio 7.1 - Moltiplicatore di Booth	1
3.1.1 Progetto e architettura	1
3.1.2 Implementazione	2
3.1.3 Simulazione	15
3.2 Esercizio 7.2 - Implementazione su board	18
3.2.1 Progetto e architettura	18
3.2.2 Implementazione	18
3.2.3 Simulazione	24
3.2.4 Timing Analysis	27
CHAPTER 4 COMUNICAZIONE CON HANDSHAKING	1
4.1 Esercizio 8.1 - Comunicazione con handshaking	1
4.2 Progetto e architettura	1
4.2.1 Implementazione	5
4.2.2 (.	22
CHAPTER 5 PROCESSORE MIC-1	1
5.1 Esercizio 9 - Processore	1
5.1.1 Analisi dell'archittettura del MIC-1	1
5.1.2 Analisi e simulazione di due istruzioni a scelta	5
CHAPTER 6 INTERFACCIA SERIALE	1
6.1 Esercizio 10 - Comunicazione seriale	1
6.2 Progetto e architettura	1
6.2.1 Implementazione	4
6.2.2 Simulazione	15
CHAPTER 7 SWITCH MULTISTADIO	1
7.1 Esercizio 11.1 - switch multistadio	1
7.2 Progetto e architettura	1
7.2.1 Implementazione	3
7.2.2 Simulazione	9

CHAPTER 1

RETI COMBINATORIE ELEMENTARI

1.1 Esercizio 1.1 - Multiplexer 16:1

Progettare, implementare in VHDL e testare mediante simulazione un multiplexer indirizzabile 16:1, utilizzando un approccio di progettazione per composizione a partire da multiplexer 4:1.

1.1.1 Progetto e architettura

Per la realizzazione del mux 16:1 per prima cosa è stato definito il multiplexer 2:1 che rappresenta il punto di partenza del progetto, mediante la composizione di 3 di questi componenti è stato realizzato un multiplexer 4:1. Il risultato è riportato in figura.

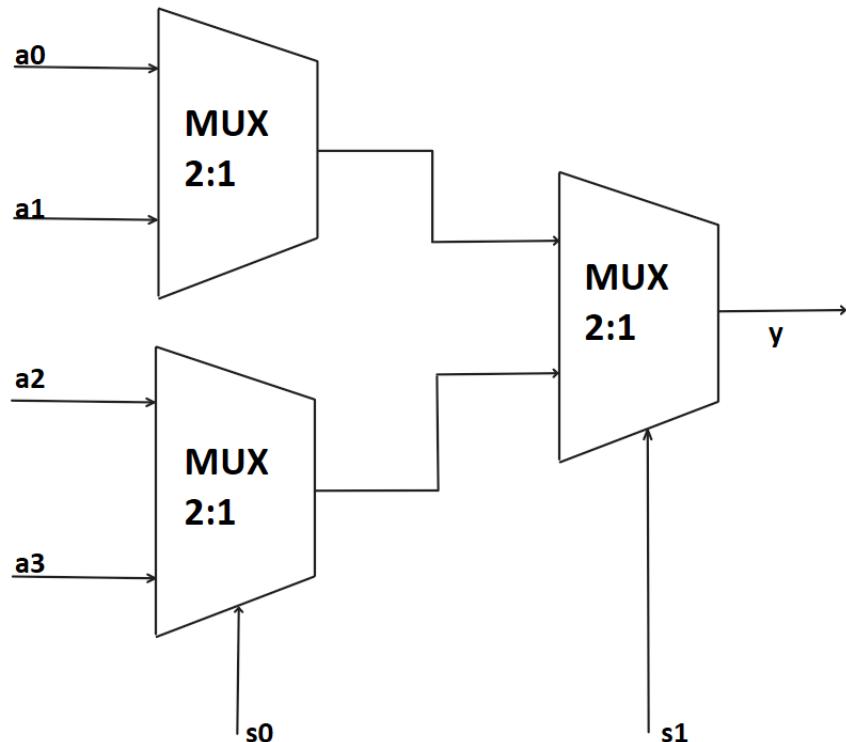


Figure 1.1: Schema del Multiplexer 4:1

Per descrivere il meccanismo operativo, si inizia dal primo livello, dove due multiplexer 2:1 determinano l'uscita basandosi sul valore del segnale di selezione

s_0 , indirizzando uno dei due ingressi verso l'uscita. Successivamente, tali uscite vengono poste in ingresso al multiplexer 2:1 del secondo livello, il quale, attraverso il segnale di selezione s_1 , determina l'uscita finale. In questo modo, il multiplexer 4:1 è caratterizzato da quattro ingressi, due segnali di selezione e una singola uscita. Implementando i multiplexer 4:1 attraverso un approccio strutturale, è stata successivamente realizzata la configurazione di un multiplexer 16:1. Tale dispositivo è stato ottenuto collegando insieme cinque multiplexer 4:1, come mostrato nella seguente figura:

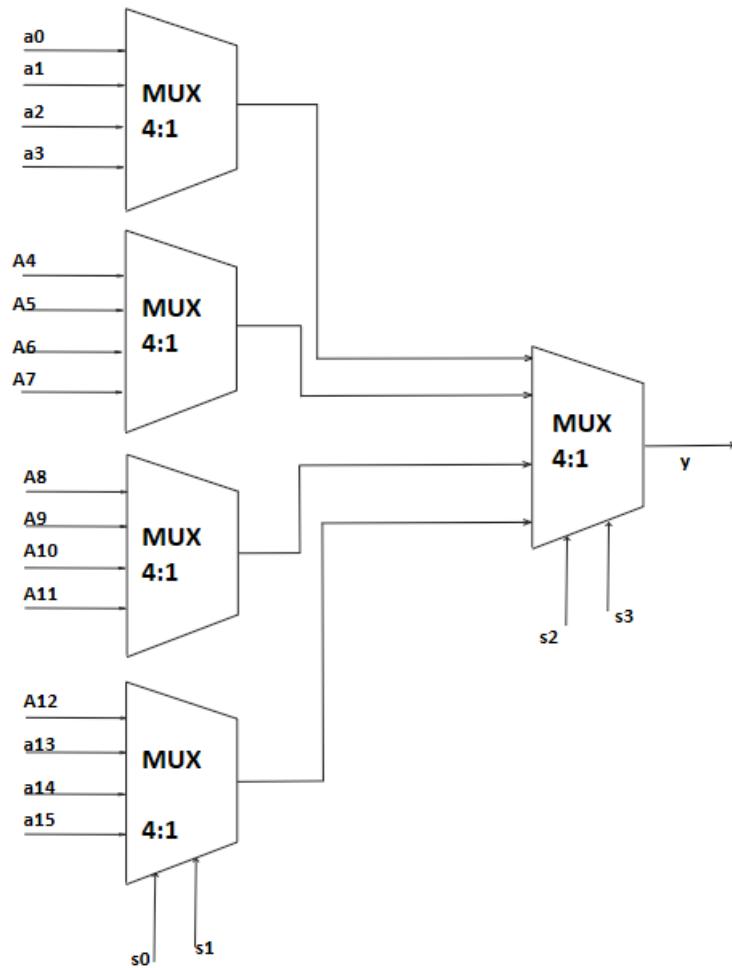


Figure 1.2: Schema del Multiplexer 16:1

Come si può notare il multiplexer 16:1 ha **16 ingressi, 4 ingressi di selezione e 1 uscita**.

1.1.2 Implementazione

In primo luogo, è stato implementato il multiplexer 2:1 definendo un'interfaccia entity chiamata mux21 nella quale sono stati definiti i segnali d'ingresso e d'uscita.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux21 is
5     port(    a0  :  in STD_LOGIC;
6             a1  :  in STD_LOGIC;
7             s   :  in STD_LOGIC;
8             y   :  out STD_LOGIC
9         );
10 end mux21;
11
12 architecture dataflow of mux21 is
13 begin
14     y <= ((a0 AND (NOT s)) OR (a1 AND s));
15 end dataflow;
```

Listing 1.1: Implementazione del Multiplexer 2:1

Per descrivere il funzionamento dell'oggetto è stata utilizzata un'architecture **dataflow** che, in VHDL, permette di descrivere il modo in cui un componente, mediante la combinazione degli ingressi, determina l'uscita.

Fatto ciò è stato implementato il multiplexer 4:1 definendo la sua interfaccia mediante un'entity mux41 che prevede 4 ingressi dato, 2 ingressi di selezione e un'uscita.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux41 is
5     port(    b0  :  in STD_LOGIC;
6             b1  :  in STD_LOGIC;
7             b2  :  in STD_LOGIC;
8             b3  :  in STD_LOGIC;
9             w0  :  in STD_LOGIC;
10            w1  :  in STD_LOGIC;
11            y0  :  out STD_LOGIC
12        );
```

```

13 end mux41;
14
15 architecture structural of mux41 is
16     signal u0, u1 : STD_LOGIC;
17     component mux21 is
18         port(    a0 : in STD_LOGIC;
19                     a1 : in STD_LOGIC;
20                     s : in STD_LOGIC;
21                     y : out STD_LOGIC
22             );
23     end component;
24
25 begin
26     mux0: mux21
27         port map( a0 => b0,
28                     a1 => b1,
29                     s => w0,
30                     y => u0
31             );
32     mux1: mux21
33         port map( a0 => b2,
34                     a1 => b3,
35                     s => w0,
36                     y => u1
37             );
38     mux2: mux21
39         port map( a0 => u0,
40                     a1 => u1,
41                     s => w1,
42                     y => y0
43             );
44 end structural;

```

Listing 1.2: Implementazione del Multiplexer 4:1

Visto che il multiplexer 4:1 è stato implementato mediante approccio strutturale non sarà descritta un'architecture dataflow ma uno **structural**. Per effettuare i collegamenti sono stati impiegati due segnali intermedi u0 e u1 al fine di collegare il primo ed il secondo livello. Successivamente è stato descritto il component mux21 che verrà stanziato per formare il multiplexer 4:1. Infine è stato effettuato il port map, permettendo il collegamento opportuno di tutti i segnali.

In utilissima istanza, è stato effettuato il collegamento dei mux 4:1, per realizzare

il mux 16:1. Si nota che l'implementazione del multiplexer 16:1 è molto vicina a quella del multiplexer 4:1. L'entity che fa riferimento al componente è chiamata **mux161** e definisce i 16 ingressi dato, i 4 ingressi di selezione e l'uscita.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux161 is
5     port( c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11,
6             c12, c13,
7             c14, c15, s0, s1, s2, s3 : in STD_LOGIC;
8             y1 : out STD_LOGIC
9 );
10
11 end mux161;
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

```

```

37          b2 => c6,
38          b3 => c7,
39          w0 => s0,
40          w1 => s1,
41          y0 => u1
42      );
43
44 mux2: mux41
45     port map( b0 => c8,
46                 b1 => c9,
47                 b2 => c10,
48                 b3 => c11,
49                 w0 => s0,
50                 w1 => s1,
51                 y0 => u2
52             );
53
54 mux3: mux41
55     port map( b0 => c12,
56                 b1 => c13,
57                 b2 => c14,
58                 b3 => c15,
59                 w0 => s0,
60                 w1 => s1,
61                 y0 => u3
62             );
63
64 mux4: mux41
65     port map( b0 => u0,
66                 b1 => u1,
67                 b2 => u2,
68                 b3 => u3,
69                 w0 => s2,
70                 w1 => s3,
71                 y0 => y1
72             );
73
74 end structural;

```

Listing 1.3: Implementazione del Multiplexer 16:1

1.1.3 Simulazione

Per simulare il componente implementato, è stato definito un **testbench**, una design unit utilizzata per fornire gli stimoli al sistema ed ottenere degli output con il fine di testare il componente realizzato.

Di seguito è riportato il codice del testbench:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux161_tb is
5 end mux161_tb;
6
7 architecture behavioural of mux161_tb is
8 component mux161 is
9     port( c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10,
10           c11, c12, c13,
11           c14, c15, s0, s1, s2, s3 : in STD_LOGIC;
12           y1 : out STD_LOGIC
13 );
14
15 signal datain : STD_LOGIC_VECTOR(15 downto 0) := (others
16   => 'U');
17 signal selection : STD_LOGIC_VECTOR(3 downto 0) := (others
18   => 'U');
19 signal dataout : STD_LOGIC := 'U';
20
21 begin
22     dut : mux161
23         port map(
24             c0 => datain(0),
25             c1 => datain(1),
26             c2 => datain(2),
27             c3 => datain(3),
28             c4 => datain(4),
29             c5 => datain(5),
30             c6 => datain(6),
31             c7 => datain(7),
32             c8 => datain(8),
33             c9 => datain(9),
34             c10 => datain(10),
35             c11 => datain(11),
36             c12 => datain(12),
             c13 => datain(13),
             c14 => datain(14),
```

```

37      c15 => datain(15),
38      s0 => selection(0),
39      s1 => selection(1),
40      s2 => selection(2),
41      s3 => selection(3),
42      y1 => dataout
43  );
44
45  stim_proc : process
46  begin
47    wait for 10 ns;
48    datain <= "1000110110110010";
49    wait for 10 ns;
50    selection <= "0000";
51    wait for 5 ns;
52    selection <= "0011";
53    wait for 5 ns;
54    selection <= "0100";
55    wait for 5 ns;
56    selection <= "0111";
57    --assert dataout = '1'
58    --report "errore"
59    --severity failure;
60    wait;
61  end process;
62
63
64 end behavioural;

```

Listing 1.4: Testbench del Multiplexer 16:1

È stata definita un'entità vuota, necessaria esclusivamente per scopi di test. Successivamente, è stato introdotto il componente effettivo da testare, denominato mux161, a cui sono seguite le definizioni dei segnali richiesti per la mappatura del componente, allo scopo di consentire l'esecuzione del test. La mappatura è stata realizzata sotto la label Design Under Test (DUT), che, mediante l'uso di 'port map', ha facilitato la connessione delle porte del componente ai segnali pertinenti. In seguito, è stato definito il processo stim_proc, il quale ha permesso la gestione degli ingressi, impostando inizialmente l'ingresso su 1000110110110010 e i segnali di selezione su 0000 per il primo test, successivamente su 0100 per il secondo test, e infine su 0111 per l'ultimo test.

La realizzazione dei test sulla piattaforma **VIVADO** ha portato ai risultati indicati

di seguito:

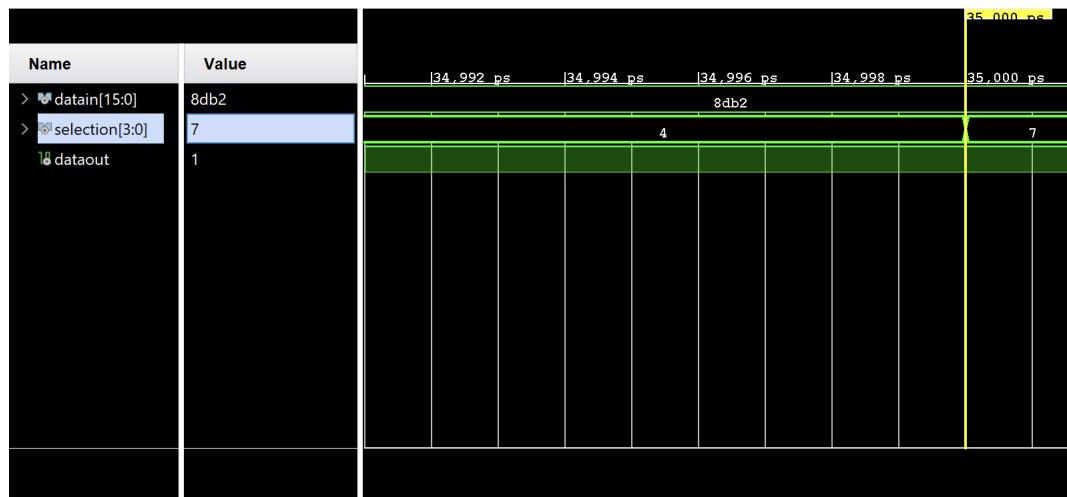


Figure 1.3

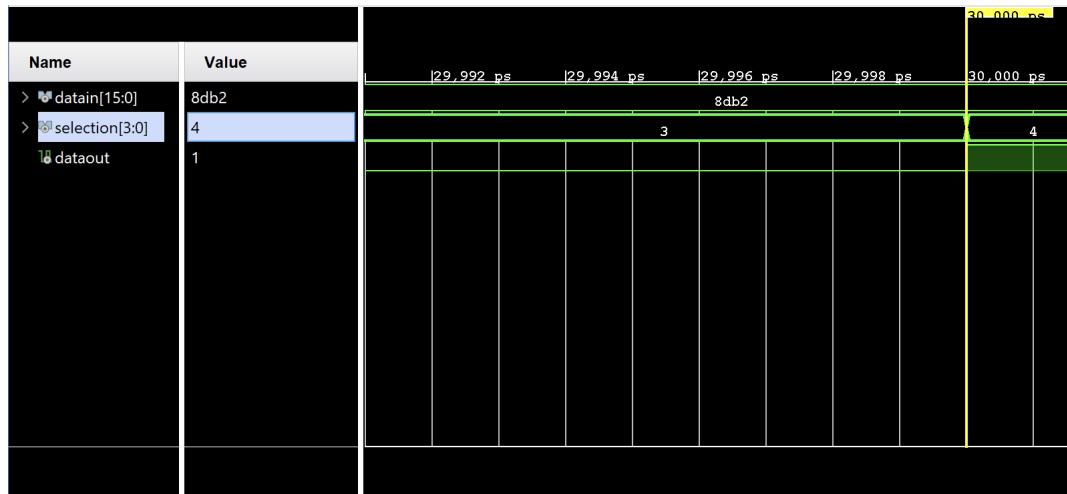


Figure 1.4

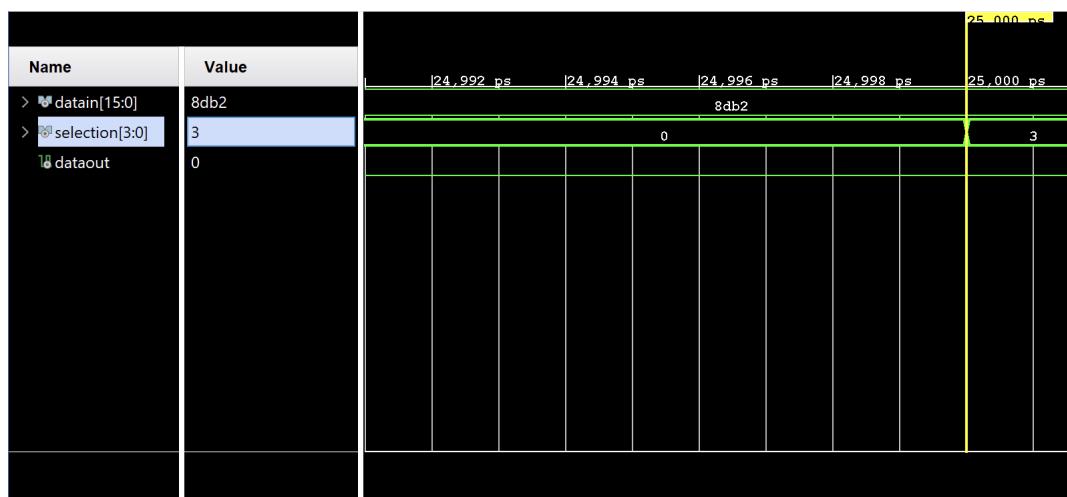


Figure 1.5

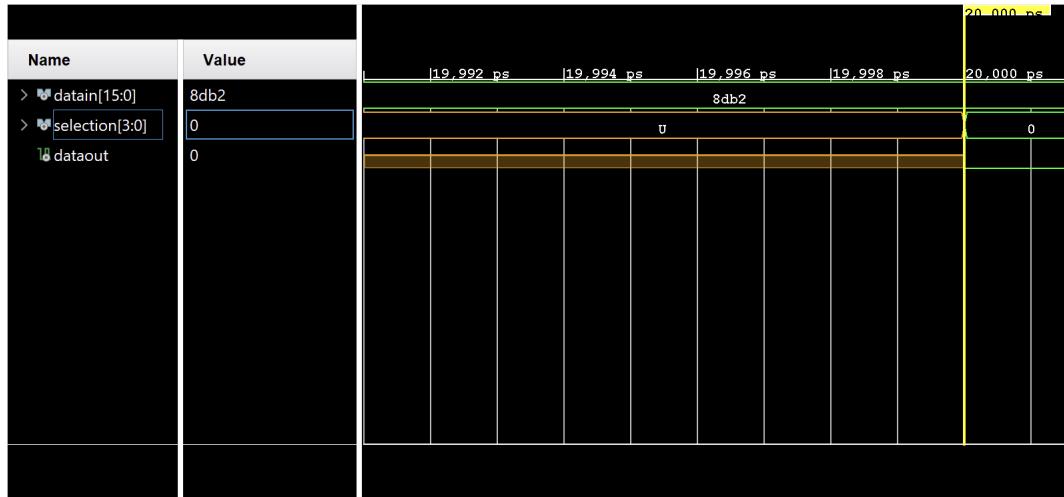


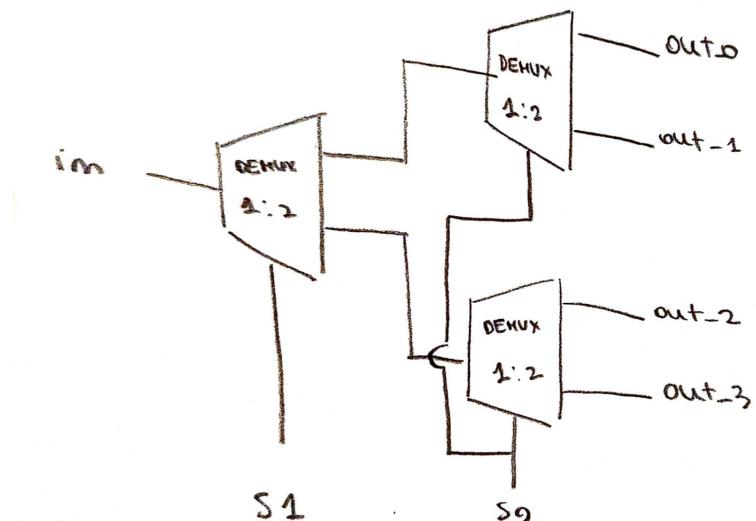
Figure 1.6

1.2 Esercizio 1.2

Utilizzando il componente sviluppato al punto precedente, progettare, implementare in VHDL e testare mediante simulazione una rete di interconnessione a 16 sorgenti e 4 destinazioni.

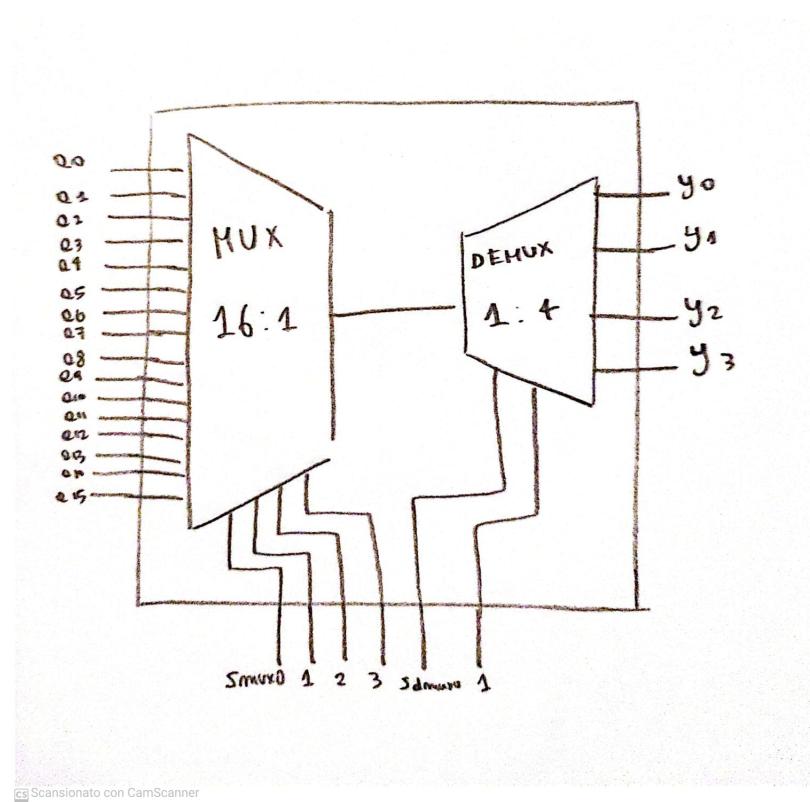
1.2.1 Progetto e architettura

Per realizzare una rete di interconnessione a 16 sorgenti e 4 destinazioni sono stati utilizzati due componenti principali: il multiplexer 16:1 realizzato nell'esercizio precedente per composizione e un demultiplexer 1:4 realizzato mediante un approccio strutturale. Il multiplexer 16:1 è realizzato mediante cinque multiplexer 4:1 combinati tra di loro, quattro al primo livello con gli stessi due ingressi di selezione ed uno al secondo livello con altri due di selezione. Ciascun multiplexer 4:1 è realizzato a sua volta mediante tre multiplexer 2:1, di cui i primi due al primo livello condividono l'ingresso di selezione mentre l'ulteriore mux al secondo livello vede in ingresso un segnale di selezione diverso. Il demultiplexer 1:4 è composto da 3 demultiplexer 1:2 opportunamente collegati tra loro, di cui uno al primo livello con un ingresso di selezione, e due al secondo livello con un altro ingresso di selezione condiviso.



Scansionato con CamScanner

Figure 1.7: Schema del Demultiplexer 1:4



Scansionato con CamScanner

Figure 1.8: Schema della rete

1.2.2 Implementazione

Il primo componente che verrà analizzato sarà il demultiplexer 1:2, segue la sua implementazione:

```

1
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4
5 entity demux12 is
6     port (    demux_12_in   : in STD_LOGIC;
7                 demux_12_sel   : in STD_LOGIC;
8                 demux_12_out  : out STD_LOGIC_VECTOR(1 downto 0)
9             );
10 end demux12;
11
12 architecture dataflow of demux12 is
13 begin
14     demux_12_out(0) <= ((NOT demux_12_sel) AND demux_12_in);
15     demux_12_out(1) <= (demux_12_sel AND demux_12_in);
16 end dataflow;

```

Listing 1.5: Demultiplexer 1:2

L'entity presenta rispettivamente un input, un ingresso di selezione ed un output. Per quanto riguarda il suo funzionamento, descritto dall'architettura dataflow, facendo riferimento alla teoria, quando l'ingresso di selezione assume il valore '0' l'input verrà collegato al primo filo d'uscita, invece quando l'ingresso di selezione assume il valore '1' l'input verrà collegato al secondo filo d'uscita.

Come già detto in precedenza, verranno stanziati tre demultiplexer 1:2 come componenti della architettura strutturale del demultiplexer 1:4, il quale presenta un input, due ingressi di selezione e quattro output.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity demux14 is
5     port (    demux_14_in   : in STD_LOGIC;
6                 demux_14_sel  : in STD_LOGIC_VECTOR(1 downto 0);
7                 demux_14_out  : out STD_LOGIC_VECTOR(3 downto 0)
8             );
9 end demux14;
10
11 architecture structural of demux14 is
12     signal u : STD_LOGIC_VECTOR(1 downto 0);

```

```

13
14 component demux12 is
15     port(    demux_12_in   : in STD_LOGIC;
16                 demux_12_sel  : in STD_LOGIC;
17                 demux_12_out : out STD_LOGIC_VECTOR(1 downto 0)
18 );
19 end component;
20
21 begin
22     mux0: demux12
23         port map( demux_12_in => demux_14_in,
24                     demux_12_sel => demux_14_sel(1),
25                     demux_12_out => u
26 );
27     mux1: demux12
28         port map( demux_12_in => u(0),
29                     demux_12_sel => demux_14_sel(0),
30                     demux_12_out => demux_14_out(1 downto 0)
31 );
32     mux2: demux12
33         port map( demux_12_in => u(1),
34                     demux_12_sel => demux_14_sel(0),
35                     demux_12_out => demux_14_out(3 downto 2)
36 );
37 end structural;

```

Listing 1.6: Demultiplexer 1:4

Il demultiplexer 1:4 e il multiplexer 16:1 vengono utilizzati come componenti dell’architettura strutturale della rete di interconnessione 16:4. L’entity net16_4 rappresenta la **rete di interconnessione**, essa è caratterizzata da 16 input (net16_4_a), 4 ingressi di selezione per il mux (net16_4_s_mux), 2 ingressi di selezione per il demux (net16_4_s_demux), e 4 uscite (net16_4_y)

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity net16_4 is
5     port(    net16_4_a : in STD_LOGIC_VECTOR(15 downto 0);
6                 net16_4_s_mux : in STD_LOGIC_VECTOR(3 downto 0);
7                 net16_4_s_demux : in STD_LOGIC_VECTOR(1 downto 0);
8                 net16_4_y : out STD_LOGIC_VECTOR(3 downto 0)

```

```

9      );
10 end net16_4;

11
12 architecture structural of net16_4 is
13     signal net16_4_u : STD_LOGIC;
14
15     component mux161 is
16         port( mux161_c : in STD_LOGIC_VECTOR(15 downto 0);
17             mux161_s : in STD_LOGIC_VECTOR(3 downto 0);
18             mux161_y1 : out STD_LOGIC
19         );
20     end component;
21
22     component demux14 is
23         port( demux_14_in : in STD_LOGIC;
24             demux_14_sel : in STD_LOGIC_VECTOR(1 downto 0);
25             demux_14_out : out STD_LOGIC_VECTOR(3 downto 0)
26         );
27     end component;
28
29 begin
30     net16_4_mux: mux161
31         port map ( mux161_c => net16_4_a,
32                     mux161_s => net16_4_s_mux,
33                     mux161_y1 => net16_4_u
34                 );
35     net16_4_demux: demux14
36         port map ( demux_14_in => net16_4_u,
37                     demux_14_sel => net16_4_s_demux,
38                     demux_14_out => net16_4_y
39                 );
40 end structural;

```

Listing 1.7: Rete di interconnessione 16:4

I 16 ingressi del mux corrisponderanno agli ingressi della rete, mentre le 4 uscite del demux alle uscite della rete. L'uscita del mux e l'ingresso del demux verranno collegate mediante il segnale intermedio net_16_4_u.

1.2.3 Simulazione

Per la simulazione è stata dichiarata una entity con un corpo vuoto, privo di segnali, in quanto essa non rappresenta un oggetto fisico da realizzare, ma è

necessaria solo per i test. Nell'architettura è stato dichiarato il componente da testare: la net16_4, sono stati mappati i suoi porti mediante dei segnali di input e output, sotto la label 'dut' (design under test) seguita da una port map. In seguito è stato definito un processo (stim_proc) nel quale, rispettando eventuali ritardi delle porte logiche, sono stati inizializzati il segnale di ingresso a "1001000100001001", il segnale del selezione del mux a "0011" e il segnale di selezione del demux a "01".

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity net16_4_tb is
5 end net16_4_tb;
6
7 architecture behavioural of net16_4_tb is
8 component net16_4 is
9     port(    net16_4_a : in STD_LOGIC_VECTOR(15 downto 0);
10             net16_4_s_mux : in STD_LOGIC_VECTOR(3 downto
11                                         0);
12             net16_4_s_demux : in STD_LOGIC_VECTOR(1 downto
13                                         0);
14             net16_4_y : out STD_LOGIC_VECTOR(3 downto 0)
15         );
16 end component;
17
18 signal datain : STD_LOGIC_VECTOR(15 downto 0) := (others
19             => 'U');
20 signal selection_mux : STD_LOGIC_VECTOR(3 downto 0) := (others
21             => 'U');
22 signal selection_demux : STD_LOGIC_VECTOR(1 downto 0) := (others
23             => 'U');
24 signal dataout : STD_LOGIC_VECTOR(3 downto 0) := (others
25             => 'U');
26
27 begin
28     dut : net16_4
29     port map(
30         net16_4_a => datain,
31         net16_4_s_mux => selection_mux,
32         net16_4_s_demux => selection_demux,
33         net16_4_y => dataout
34     );
35
36     process is
37         variable stim : STD_LOGIC_VECTOR(15 downto 0);
38         variable sel_mux : STD_LOGIC_VECTOR(3 downto 0);
39         variable sel_demux : STD_LOGIC_VECTOR(1 downto 0);
40         variable dataout : STD_LOGIC_VECTOR(3 downto 0);
41     begin
42         for i in 0 to 15 loop
43             stim(i) := '0';
44         end loop;
45         sel_mux(3) := '1';
46         sel_mux(2) := '0';
47         sel_mux(1) := '1';
48         sel_mux(0) := '0';
49         sel_demux(1) := '0';
50         sel_demux(0) := '1';
51         dataout := (others => 'U');
52         report "Initial values set" & integer'image(i);
53         report "Datain = " & integer'image(to_integer(unsigned(datain)));
54         report "Sel_mux = " & integer'image(to_integer(unsigned(sel_mux)));
55         report "Sel_demux = " & integer'image(to_integer(unsigned(sel_demux)));
56         report "Dataout = " & integer'image(to_integer(unsigned(dataout)));
57         report "-----";
58         if i = 15 then
59             report "Test completed" & integer'image(i);
60             exit;
61         else
62             report "Datain = " & integer'image(to_integer(unsigned(datain)));
63             report "Sel_mux = " & integer'image(to_integer(unsigned(sel_mux)));
64             report "Sel_demux = " & integer'image(to_integer(unsigned(sel_demux)));
65             report "Dataout = " & integer'image(to_integer(unsigned(dataout)));
66             report "-----";
67         end if;
68         wait on datain, sel_mux, sel_demux;
69     end process;
70 end;

```

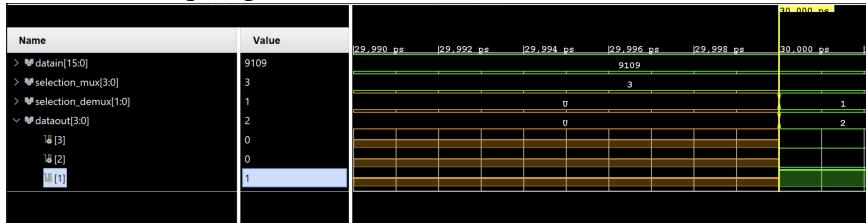
```

28     );
29
30     stim_proc : process
31 begin
32     wait for 10 ns;
33     datain <= "1001000100001001";
34     wait for 10 ns;
35     selection_mux <= "0011"; -- il quarto bit che vale 1.
36     wait for 10 ns;
37     selection_demux <= "01"; -- esce 1 sul secondo filo.
38     wait for 10 ns;
39     wait;
40 end process;
41
42 end behavioural;

```

Listing 1.8: Rete di interconnessione 16:4 testbench

Selezionando "0011" mediante il mux viene preso in considerazione il quarto ingresso, su cui abbiamo il bit '1', e poi inizializzando l'ingresso di selezione del demux a "01" andiamo a collegare l'uscita del mux con il secondo filo d'uscita del demux, che, dunque, porterà il valore 1.



1.3 Esercizio 1.3

Sintetizzare ed implementare su board il progetto della rete di interconnessione sviluppato al punto 1.2, utilizzando gli switch per fornire gli input di selezione e i led per visualizzare i 4 bit di uscita. Per quanto riguarda i 16 bit dato in input, essi devono essere immessi mediante switch, 8 bit alla volta, sviluppando un'apposita "rete di controllo" per l'acquisizione che utilizzi due bottoni della board per caricare rispettivamente la prima e la seconda metà del dato in ingresso.

1.3.1 Progetto e architettura

Per l'implementazione su board è stato riutilizzare il componente net16_4 dell'esercizio precedente, aggiungendo un'unità di controllo che permetta il corretto instradamento, vista la particolare richiesta di immettere i dati in input 8 bit alla volta.

L'unità di controllo prende in input 8 bit mediante gli switch della board, due bit di selezione (rispettivamente per instradare nell'ingresso della rete 16:4 la prima metà degli switch e poi la seconda metà), e ha come uscita 16 bit che andranno poi in ingresso alla rete.

Il sistema è stato costruito seguendo un approccio strutturale. Esso è composto dalla rete e dall'unità di controllo e prende in ingresso, oltre agli 8 bit forniti dagli switch, anche gli ingressi di selezione per multiplexer e demultiplexer. Questi vengono mappati nei restanti switch della board. Lo schema del sistema è riportato di seguito:

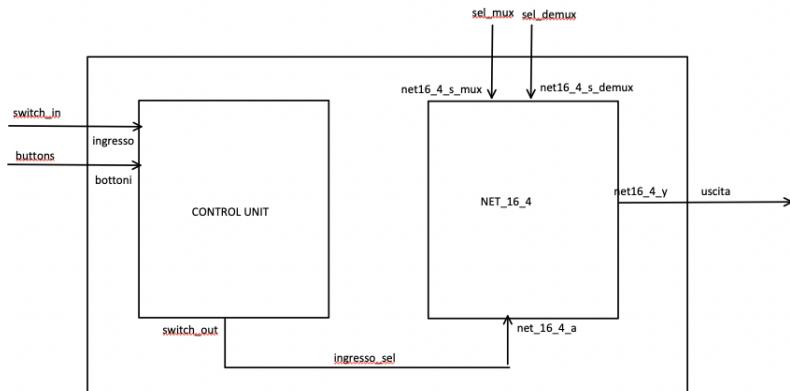


Figure 1.9: Schema implementazione su board.

1.3.2 Implementazione

Il componente relativo alla rete è identico a quello dell'esercizio precedente: procediamo ad analizzare l'**unità di controllo**.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity unita_di_controllo is
5     port(    switch_in : in std_logic_vector(7 downto 0);
6             buttons : in std_logic_vector(1 downto 0);
7             switch_out : out std_logic_vector(15 downto 0)
8         );
9 end unita_di_controllo;
10
11 architecture behavioral of unita_di_controllo is
12     signal temp : std_logic_vector(15 downto 0) := (others =>
13         '0');

```

```

14 begin
15     process(buttons, switch_in)
16         begin
17             if(buttons(0) = '1') then
18                 temp(7 downto 0) <= switch_in;
19             elsif(buttons(1) = '1') then
20                 temp(15 downto 8) <= switch_in;
21             end if;
22         end process;
23
24     switch_out <= temp;
25 end behavioral;

```

Listing 1.9: Unità di controllo

L’unità di controllo si occupa di instradare correttamente l’input proveniente dagli switch della board. Sono stati definiti due vettori di input e uno di output:

- `switch_in` è un vettore di ingresso di 8 bit;
- `buttons` è un vettore di 2 bit il cui valore determina quale porzione del vettore di uscita sarà sovrascritta con il valore del vettore di input.

Il comportamento del componente descritto mediante un *process*, è il seguente: viene istanziato un segnale `temp` da 16 bit e lo si inizializza a 0. Successivamente viene letto il valore degli elementi del vettore `buttons`: se il primo bit del vettore è alto si sovrascrive la prima metà di `temp` con il valore del vettore di ingresso, mentre se è alto il secondo bit viene sovrascritta la seconda metà. Infine, il valore attuale di `temp` viene salvato nell’uscita.

I componenti, sono collegati opportunamente per realizzare il componente finale. Per l’architettura si è scelto ancora una volta un approccio strutturale, in cui si individuano due componenti: la `net16_4` e l’unità di controllo: l’implementazione del sistema finale consiste esclusivamente nel mappare correttamente i segnali di ingresso, uscita e interconnessione. Il segnale `ingresso_sel` rappresenta l’uscita dell’unità di controllo che rappresenterà l’ingresso della rete.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity sistema is
5     port(    ingresso : in std_logic_vector(7 downto 0);
6                  bottoni : in std_logic_vector(1 downto 0);
7                  sel_mux : in std_logic_vector(3 downto 0);

```

```

8         sel_demux : in std_logic_vector(1 downto 0);
9         uscita : out std_logic_vector(3 downto 0)
10        );
11 end sistema;
12
13 architecture structural of sistema is
14     signal ingresso_sel : std_logic_vector(15 downto 0);
15
16     component unita_di_controllo is
17         port(    switch_in : in std_logic_vector(7 downto 0);
18                     buttons : in std_logic_vector(1 downto 0);
19                     switch_out : out std_logic_vector(15 downto 0)
20                 );
21     end component;
22
23     component net16_4 is
24         port(    net16_4_a : in STD_LOGIC_VECTOR(15 downto 0);
25                     net16_4_s_mux : in STD_LOGIC_VECTOR(3 downto
26                                     0);
27                     net16_4_s_demux : in STD_LOGIC_VECTOR(1 downto
28                                     0);
29                     net16_4_y : out STD_LOGIC_VECTOR(3 downto 0)
30                 );
31     end component;
32
33 begin
34     unita_di_controllo : unita
35         port map (    switch_in => ingresso,
36                         buttons => bottoni,
37                         switch_out => ingresso_sel
38                     );
39     net16_4 : rete
40         port map (    net16_4_a => ingresso_sel,
41                         net16_4_s_mux => sel_mux,
42                         net16_4_s_demux => sel_demux,
43                         net16_4_y => uscita
44                     );
45 end structural;

```

I dati in ingresso del sistema sono quelli mappati sulla board fisica, essi vengono poi mappati successivamente all'interno dei componenti istanziati.

1.3.3 Simulazione

Per poter utilizzare la board è stato necessario effettuare alcune modifiche al file di configurazione `Nexys-A7-100T-Master.xdc`: questo file permette di instradare i segnali di input e di output sui pin fisici sulla board. In particolare, si è scelto di mappare il vettore `ingresso` del sistema con gli ultimi 8 switch della board, mentre i vettori di selezione `sel_mux` e `sel_demux` sono mappati dal terzo all'ottavo switch:

```
1 ##Switches
2 set_property -dict { PACKAGE_PIN J15    IOSTANDARD LVCMOS33 }
      [get_ports { ingresso[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
3 set_property -dict { PACKAGE_PIN L16    IOSTANDARD LVCMOS33 }
      [get_ports { ingresso[1] }]; #IO_L3N_T0_DQS_EMCCCLK_14
      Sch=sw[1]
4 set_property -dict { PACKAGE_PIN M13    IOSTANDARD LVCMOS33 }
      [get_ports { ingresso[2] }]; #IO_L6N_T0_D08_VREF_14
      Sch=sw[2]
5 set_property -dict { PACKAGE_PIN R15    IOSTANDARD LVCMOS33 }
      [get_ports { ingresso[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
6 set_property -dict { PACKAGE_PIN R17    IOSTANDARD LVCMOS33 }
      [get_ports { ingresso[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
7 set_property -dict { PACKAGE_PIN T18    IOSTANDARD LVCMOS33 }
      [get_ports { ingresso[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
8 set_property -dict { PACKAGE_PIN U18    IOSTANDARD LVCMOS33 }
      [get_ports { ingresso[6] }]; #IO_L17N_T2_A13_D29_14
      Sch=sw[6]
9 set_property -dict { PACKAGE_PIN R13    IOSTANDARD LVCMOS33 }
      [get_ports { ingresso[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
10 set_property -dict { PACKAGE_PIN T8     IOSTANDARD LVCMOS18 }
      [get_ports { sel_mux[0] }]; #IO_L24N_T3_34 Sch=sw[8]
11 set_property -dict { PACKAGE_PIN U8     IOSTANDARD LVCMOS18 }
      [get_ports { sel_mux[1] }]; #IO_25_34 Sch=sw[9]
12 set_property -dict { PACKAGE_PIN R16    IOSTANDARD LVCMOS33 }
      [get_ports { sel_mux[2] }]; #IO_L15P_T2_DQS_RDWR_B_14
      Sch=sw[10]
13 set_property -dict { PACKAGE_PIN T13    IOSTANDARD LVCMOS33 }
      [get_ports { sel_mux[3] }]; #IO_L23P_T3_A03_D19_14
      Sch=sw[11]
14 set_property -dict { PACKAGE_PIN H6     IOSTANDARD LVCMOS33 }
      [get_ports { sel_demux[0] }]; #IO_L24P_T3_35 Sch=sw[12]
```

```

15 set_property -dict { PACKAGE_PIN U12    IOSTANDARD LVCMOS33 }
      [get_ports { sel_demux[1] }]; #IO_L20P_T3_A08_D24_14
      Sch=sw[13]
16 #set_property -dict { PACKAGE_PIN U11    IOSTANDARD LVCMOS33 }
      [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14
      Sch=sw[14]
17 #set_property -dict { PACKAGE_PIN V10    IOSTANDARD LVCMOS33 }
      [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

```

Invece, il vettore di uscita uscita del sistema è stato mappato sugli ultimi 4 led della board:

```

1 ## LEDs
2 set_property -dict { PACKAGE_PIN H17    IOSTANDARD LVCMOS33 }
      [get_ports { uscita[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
3 set_property -dict { PACKAGE_PIN K15    IOSTANDARD LVCMOS33 }
      [get_ports { uscita[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
4 set_property -dict { PACKAGE_PIN J13    IOSTANDARD LVCMOS33 }
      [get_ports { uscita[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
5 set_property -dict { PACKAGE_PIN N14    IOSTANDARD LVCMOS33 }
      [get_ports { uscita[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]

```

Infine, i due bottoni di selezione sono stati mappati sui bottoni **P17** ed **M17**

```

1 ##Buttons
2 #set_property -dict { PACKAGE_PIN C12    IOSTANDARD LVCMOS33 }
      [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15
      Sch=cpu_resetn
3 #set_property -dict { PACKAGE_PIN N17    IOSTANDARD LVCMOS33 }
      [get_ports { reset }]; #IO_L9P_T1_DQS_14 Sch=btnc
4 #set_property -dict { PACKAGE_PIN M18    IOSTANDARD LVCMOS33 }
      [get_ports { load_enable }]; #IO_L4N_T0_D05_14 Sch=btlu
5 set_property -dict { PACKAGE_PIN P17    IOSTANDARD LVCMOS33 }
      [get_ports { bottoni[0] }]; #IO_L12P_T1_MRCC_14 Sch=btlnl
6 set_property -dict { PACKAGE_PIN M17    IOSTANDARD LVCMOS33 }
      [get_ports { bottoni[1] }]; #IO_L10N_T1_D15_14 Sch=btlnr
7 #set_property -dict { PACKAGE_PIN P18    IOSTANDARD LVCMOS33 }
      [get_ports { BTND }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd

```

I risultati della simulazione sono i seguenti: come dati in input si è deciso di utilizzare gli stessi dell'esercizio precedente, settando quindi:

- vettore di ingresso pari a **1001000100001001** (9109);
- vettore di selezione del mux pari a **0011** (3);

- vettore di selezione del demux pari a **01** (1).

1.4 Esercizio 2.1 - Sistema ROM+M

Progettare, implementare in VHDL e testare mediante simulazione un sistema S composto da una ROM puramente combinatoria di 16 locazioni da 8 bit ciascuna e da una macchina combinatoria M che opera come segue: fornito al sistema un indirizzo A di 4 bit, il sistema restituisce il valore contenuto nella ROM all'indirizzo A opportunamente “trasformato” attraverso la macchina M. Il comportamento della macchina M è totalmente a scelta dello studente, l'unico vincolo è che essa prenda in ingresso 8 bit e ne fornisca in uscita 4.

1.4.1 Progetto e architettura

Il sistema S è composto da due componenti: la ROM da 16 locazioni da 8 bit ciascuna, ed una macchina M. L'input del sistema è dato da un indirizzo da 4 bit, che consente di ottenere il dato presente nella locazione corrispondente nella ROM. Questo sarà poi dato in ingresso alla macchina M. Si è scelto di costruire il sistema mediante un approccio strutturale:

ROM

Si è deciso di realizzare la ROM usando un approccio **comportamentale**. Il componente prende in ingresso l'indirizzo per poi restituire il dato corrispondente a quella locazione; come nota, i dati con cui si popola la ROM sono scelti in modo che ogni 4 bit sia presente un solo 1, per fare in modo che venga rispettato il vincolo scelto per la macchina M.

Macchina M

Si è scelto di implementare la macchina M come un **encoder 8:4**: in particolare, si è scelto di procedere per composizione a partire da due elementi più semplici, ovvero due **encoder 4:2**, che daranno entrambi in uscita due bit, ottenendo quindi un'uscita finale composta da 4 bit.

Sistema S

Infine, il sistema finale è stato costruito utilizzando i due componenti precedenti: il dato alla locazione *address* presente nella ROM sarà quindi l'ingresso della macchina M, da cui si otterrà quindi l'uscita finale.

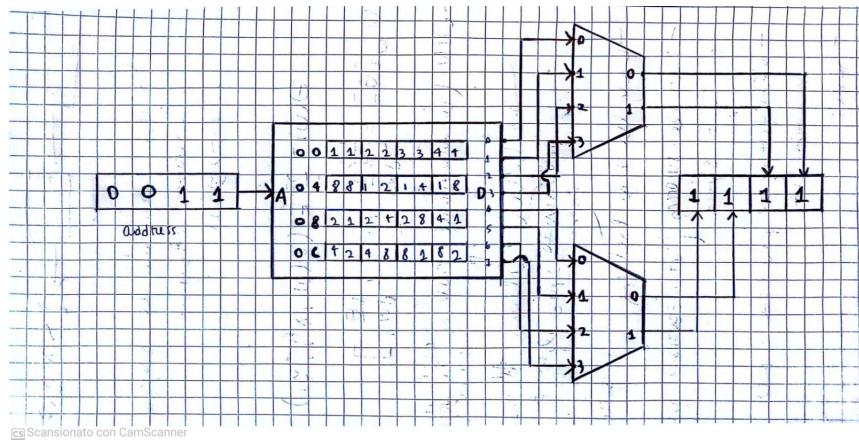


Figure 1.10

1.4.2 Implementazione

Procediamo ad analizzare i componenti, dai più semplici ai più complessi.

ROM

Procediamo con il codice della ROM:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use work.all;
5
6 entity rom is
7     port (
8         address : in std_logic_vector(3 downto 0);
9         out_value : out std_logic_vector(7 downto 0)
10    );
11 end entity rom;
12
13 architecture RTL of rom is
14     type MEMORY_16_8 is array (0 to 15) of std_logic_vector(7
15         downto 0);
16     constant ROM_16_8 : MEMORY_16_8 := (
17         "00010001",
18         "00100010",
19         "01000100",
20         "10001000",
21         "00010010",
22         "00010100",
23         "00011000",
24         "00011100",
25         "00100000",
26         "00100001",
27         "00100000",
28         "00100000",
29         "00100000",
30         "00100000",
31         "00100000",
32         "00100000");
33
34 begin
35     process (address) begin
36         if address = "0000" then
37             out_value <= ROM_16_8(0);
38         elsif address = "0001" then
39             out_value <= ROM_16_8(1);
40         elsif address = "0010" then
41             out_value <= ROM_16_8(2);
42         elsif address = "0011" then
43             out_value <= ROM_16_8(3);
44         elsif address = "0100" then
45             out_value <= ROM_16_8(4);
46         elsif address = "0101" then
47             out_value <= ROM_16_8(5);
48         elsif address = "0110" then
49             out_value <= ROM_16_8(6);
50         elsif address = "0111" then
51             out_value <= ROM_16_8(7);
52         elsif address = "1000" then
53             out_value <= ROM_16_8(8);
54         elsif address = "1001" then
55             out_value <= ROM_16_8(9);
56         elsif address = "1010" then
57             out_value <= ROM_16_8(10);
58         elsif address = "1011" then
59             out_value <= ROM_16_8(11);
60         elsif address = "1100" then
61             out_value <= ROM_16_8(12);
62         elsif address = "1101" then
63             out_value <= ROM_16_8(13);
64         elsif address = "1110" then
65             out_value <= ROM_16_8(14);
66         elsif address = "1111" then
67             out_value <= ROM_16_8(15);
68         else
69             out_value <= ROM_16_8(15);
70         end if;
71     end process;
72 end RTL;
```

```

23      "00100001",
24      "00100100",
25      "00101000",
26      "01000001",
27      "01000010",
28      "01001000",
29      "10000001",
30      "10000100",
31      "10000010"
32  );
33 begin
34     out_value <= ROM_16_8(to_integer(unsigned(address)));
35 end architecture RTL;

```

Listing 1.10: ROM 16:8

Mentre nell'*entity* si definiscono semplicemente l'indirizzo di ingresso e l'uscita corrispondente al dato, nell'*architecture* della ROM si dichiara un tipo rappresentante la ROM stessa. Questo è un array di 16 *std_logic_vector*; dichiarato il tipo, si istanzia la componente del tipo dichiarato e la si popola con dei dati, i quali rispettano la condizione relativa all'encoder(ogni 4 bit è presente un solo 1).

Successivamente si definisce il comportamento della ROM: il segnale di output si ottiene dando in ingresso alla *ROM_16_8* l'indirizzo, ottenendo il dato corrispondente.

Notiamo tuttavia che *address* è un *std_logic_vector*, quindi, prima lo si deve convertire (casting) in un *unsigned* e poi in un intero.

Macchina M

Vediamo ora il codice della macchina M. Essendo stata costruita per composizione, vediamo prima l'elemento base, ovvero l'*encoder 4:2*:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity encoder42 is
5   port(  a : in STD_LOGIC_VECTOR (3 downto 0);
6           y : out STD_LOGIC_VECTOR (1 downto 0)
7 );
8 end encoder42;
9
10 architecture dataflow of encoder42 is
11   begin

```

```

12      y      <=  "00" when a="0001" else
13                      "01" when a="0010" else
14                      "10" when a="0100" else
15                      "11" when a="1000" else
16                      "--";
17 end dataflow;

```

Listing 1.11: Encoder 4:2

Dalla definizione dell'entity notiamo che l'encoder prende in ingresso un std_logic_vector da 4 bit, e in uscita se ne ottengono 2; il comportamento della macchina è descritto con un'architettura *dataflow* definendo l'uscita per ogni combinazione di ingressi, la quale è stata ottenuta dalla teoria. Successivamente, vediamo la costruzione della macchina M per composizione: ,

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.all;
4
5 entity m_machine is
6     port( a_m : in STD_LOGIC_VECTOR(7 downto 0);
7             y_m : out STD_LOGIC_VECTOR(3 downto 0)
8         );
9 end m_machine;
10
11 architecture structural of m_machine is
12
13     component encoder42 is
14         port( a : in STD_LOGIC_VECTOR(3 downto 0);
15                 y : out STD_LOGIC_VECTOR(1 downto 0)
16         );
17     end component;
18
19     signal a_temp1, a_temp2 : STD_LOGIC_VECTOR(3 downto 0);
20
21 begin
22     a_temp1 <= a_m(3 downto 0);
23     a_temp2 <= a_m(7 downto 4);
24
25     encoder1 : encoder42
26     port map(
27         a => a_temp1,
28         y => y_m(1 downto 0)

```

```

29 );
30
31     encoder2 : encoder42
32     port map(
33         a => a_temp2,
34         y => y_m(3 downto 2)
35     );
36 end structural;

```

La macchina prende in ingresso un vettore da 8 bit e ne da in uscita 4: è descritta mediante uno *structural* definendo un *component* encoder4_2 e mappando le uscite su quella dell'entity.

Sistema S

Infine, vediamo la descrizione del sistema S. ,

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.all;
4
5 entity s_system is
6     port (
7         in_system : in STD_LOGIC_VECTOR(3 downto 0);
8         out_system : out STD_LOGIC_VECTOR(3 downto 0)
9     );
10 end s_system;
11
12 architecture structural of s_system is
13     signal u : STD_LOGIC_VECTOR(7 downto 0);
14
15     component rom is
16         port (
17             address : in std_logic_vector(3 downto 0);
18             out_value : out std_logic_vector(7 downto 0)
19         );
20     end component;
21
22     component m_machine is
23         port( a_m : in STD_LOGIC_VECTOR(7 downto 0);
24               y_m : out STD_LOGIC_VECTOR(3 downto 0)
25         );
26     end component;

```

```

27
28 begin
29     ROM_system : rom
30         port map(
31             address => in_system,
32             out_value => u
33         );
34
35     M_MACHINE_system : m_machine
36         port map(
37             a_m => u,
38             y_m => out_system
39         );
40 end structural;

```

Anche in questo caso si è proceduto per composizione, definendo due *component* rom e m_machine. Si è utilizzato un segnale *u* di interconnessione tra le due componenti per collegarle, in particolare questo segnale rappresenterà l'uscita della ROM e l'ingresso della macchina M.

1.4.3 Simulazione

Per la simulazione si è scritto un *testbench*: ,

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.all;
4
5 entity s_system_tb is
6 end s_system_tb;
7
8 architecture behavioural of s_system_tb is
9     component s_system is
10        port(    in_system : in STD_LOGIC_VECTOR(3 downto 0);
11                  out_system : out STD_LOGIC_VECTOR(3 downto 0)
12            );
13     end component;
14
15     signal datain : STD_LOGIC_VECTOR(3 downto 0) := (others =>
16                                         'U');
17     signal dataout : STD_LOGIC_VECTOR(3 downto 0) := (others
18                                         => 'U');

```

17

```

18      begin
19          dut : s_system
20          port map(
21              in_system => datain,
22              out_system => dataout
23          );
24
25          stim_proc : process
26          begin
27              wait for 10 ns;
28              datain <= "0011";
29              wait for 10 ns;
30              wait;
31          end process;
32
33 end behavioural;

```

Per la simulazione è stata dichiarata una entity con un corpo vuoto, privo di segnali, in quanto essa non rappresenta un oggetto fisico da realizzare, ma è necessaria solo per i test. Successivamente, nel architecture dell'entity, si è dichiarata una componente relativa a quella da testare, ovvero proprio il sistema s.

Si è settato l'ingresso a "0011": l'uscita della ROM è quindi quella relativa al quarto indirizzo, ovvero "10001000"; gli encoder vedranno in ingresso rispettivamente i vettori "1000" e "1000", dando quindi in uscita "1111". Dalle simulazioni notiamo che il risultato è infatti quello atteso.

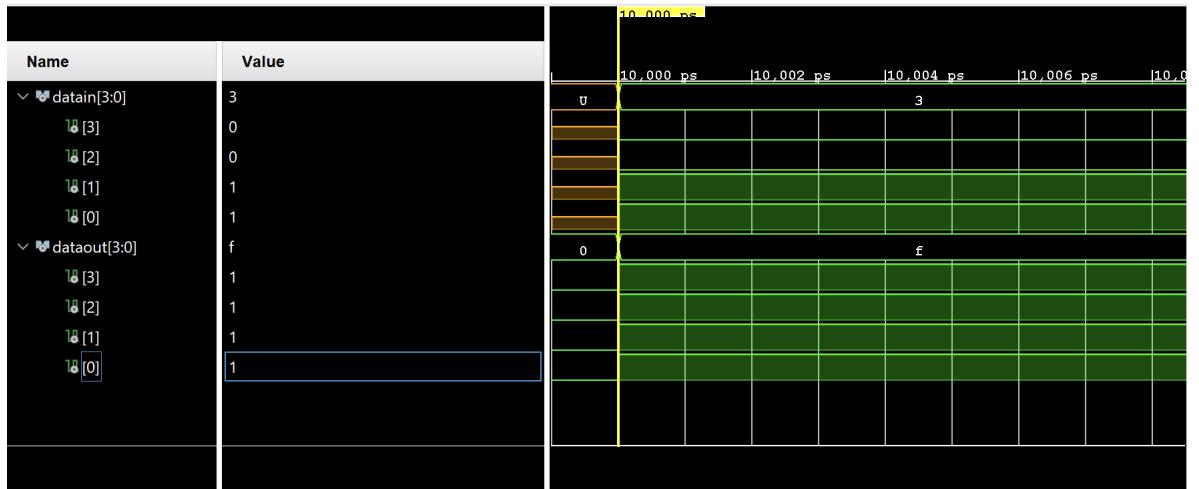


Figure 1.11: Risultato della simulazione del tb del Sistema S

1.5 Esercizio 2.2

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.

1.5.1 Progetto e architettura

Il problema richiedeva di andare ad utilizzare gli switch per fornire l'indirizzo della rom, per questo motivo si è deciso di collegare l'ingresso dell's-system (in_system) ai primi 4 switch. Per leggere l'uscita codificata dall'encoder sui led come ci è stato richiesto, è stato scelto di mappare l'out_system sui primi 4 led della board.

1.5.2 Implementazione

Vediamo le modifiche effettuate per permettere il funzionamento di questo esercizio:

```
,  
1 ##Switches  
2 set_property -dict { PACKAGE_PIN J15    IOSTANDARD LVCMOS33 }  
   [get_ports { in_system[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]  
3 set_property -dict { PACKAGE_PIN L16    IOSTANDARD LVCMOS33 }  
   [get_ports { in_system[1] }]; #IO_L3N_T0_DQS_EMCCCLK_14  
   Sch=sw[1]  
4 set_property -dict { PACKAGE_PIN M13    IOSTANDARD LVCMOS33 }  
   [get_ports { in_system[2] }]; #IO_L6N_T0_D08_VREF_14  
   Sch=sw[2]  
5 set_property -dict { PACKAGE_PIN R15    IOSTANDARD LVCMOS33 }  
   [get_ports { in_system[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]  
6 #set_property -dict { PACKAGE_PIN R17    IOSTANDARD LVCMOS33 }  
   [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]  
7 #set_property -dict { PACKAGE_PIN T18    IOSTANDARD LVCMOS33 }  
   [get_ports { SW[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]  
8 #set_property -dict { PACKAGE_PIN U18    IOSTANDARD LVCMOS33 }  
   [get_ports { SW[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]  
9 #set_property -dict { PACKAGE_PIN R13    IOSTANDARD LVCMOS33 }  
   [get_ports { SW[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]  
10 #set_property -dict { PACKAGE_PIN T8     IOSTANDARD LVCMOS18 }  
   [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]  
11 #set_property -dict { PACKAGE_PIN U8     IOSTANDARD LVCMOS18 }  
   [get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
```

```

12 #set_property -dict { PACKAGE_PIN R16    IOSTANDARD LVCMOS33 }
      [get_ports { SW[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
13 #set_property -dict { PACKAGE_PIN T13    IOSTANDARD LVCMOS33 }
      [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
14 #set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 }
      [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
15 #set_property -dict { PACKAGE_PIN U12    IOSTANDARD LVCMOS33 }
      [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
16 #set_property -dict { PACKAGE_PIN U11    IOSTANDARD LVCMOS33 }
      [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14
      Sch=sw[14]
17 #set_property -dict { PACKAGE_PIN V10    IOSTANDARD LVCMOS33 }
      [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]
18
19 ## LEDs
20 set_property -dict { PACKAGE_PIN H17    IOSTANDARD LVCMOS33 }
      [get_ports { out_system[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
21 set_property -dict { PACKAGE_PIN K15    IOSTANDARD LVCMOS33 }
      [get_ports { out_system[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
22 set_property -dict { PACKAGE_PIN J13    IOSTANDARD LVCMOS33 }
      [get_ports { out_system[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
23 set_property -dict { PACKAGE_PIN N14    IOSTANDARD LVCMOS33 }
      [get_ports { out_system[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
24 #set_property -dict { PACKAGE_PIN R18    IOSTANDARD LVCMOS33 }
      [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
25 #set_property -dict { PACKAGE_PIN V17    IOSTANDARD LVCMOS33 }
      [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
26 #set_property -dict { PACKAGE_PIN U17    IOSTANDARD LVCMOS33 }
      [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
27 #set_property -dict { PACKAGE_PIN U16    IOSTANDARD LVCMOS33 }
      [get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
28 #set_property -dict { PACKAGE_PIN V16    IOSTANDARD LVCMOS33 }
      [get_ports { LED[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
29 #set_property -dict { PACKAGE_PIN T15    IOSTANDARD LVCMOS33 }
      [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
30 #set_property -dict { PACKAGE_PIN U14    IOSTANDARD LVCMOS33 }
      [get_ports { LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
31 #set_property -dict { PACKAGE_PIN T16    IOSTANDARD LVCMOS33 }
      [get_ports { LED[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14
      Sch=led[11]
32 #set_property -dict { PACKAGE_PIN V15    IOSTANDARD LVCMOS33 }

```

```
[get_ports { LED[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
33 #set_property -dict { PACKAGE_PIN V14    IOSTANDARD LVCMOS33 }
    [get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
34 #set_property -dict { PACKAGE_PIN V12    IOSTANDARD LVCMOS33 }
    [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
35 #set_property -dict { PACKAGE_PIN V11    IOSTANDARD LVCMOS33 }
    [get_ports { LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14
    Sch=led[15]
```

CHAPTER 2

RETI SEQUENZIALI ELEMENTARI

2.1 Esercizio 3.1 - Riconoscitore di sequenza

Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza 101. La macchina prende in ingresso un segnale binario i che rappresenta il dato, un segnale A di temporizzazione e un segnale M di modo, che ne disciplina il funzionamento, e fornisce un'uscita Y alta quando la sequenza viene riconosciuta. In particolare,

- se **$M=0$** , la macchina valuta i bit seriali in ingresso a gruppi di 3 (sequenze non sovrapposte),
- se **$M=1$** , la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta (sequenze parzialmente sovrapposte).

2.1.1 Progetto e architettura

Per l'implementazione del seguente esercizio si sono, prima di tutto, progettati gli automi dei riconoscitori di sequenza **parzialmente sovrapposto e non sovrapposto**. Entrambi gli automi sono stati riportati nelle figure di seguito:

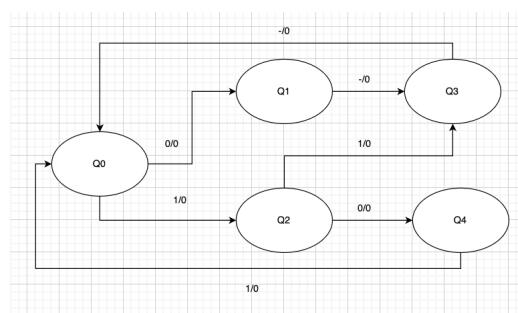


Figure 2.1: Riconoscitore di Sequenza Non Sovrapposto

Un riconoscitore di sequenza **non sovrapposto** è un tipo specifico di riconoscitore in cui **ogni stato ha un'unica transizione possibile per ciascun input**. In altre parole, il sistema non può trovarsi contemporaneamente in più di uno stato dopo aver letto l'input. In un automa a stati finiti tradizionale, contrariamente, uno stato può avere transizioni per diversi input e può muoversi in più di uno stato dopo la

lettura di un input. Tuttavia, in un riconoscitore non sovrapposto, questa sovrapposizione di stati non è consentita, come già anticipato precedentemente. Questo tipo di automa, pertanto, può semplificare l'analisi e la progettazione di sistemi, poiché riduce la complessità nella gestione degli stati. La nozione di "non sovrapposto" si riferisce al fatto che, in ogni momento, **l'automa si trova chiaramente in uno e un solo stato**.

Si è giunti alla formalizzazione dell'automa definitivo successivamente alla semplificazione degli stati equivalenti (ossia degli stati per i quali dati stessi input, si restituiscono stessi output). La sequenza che intendiamo riconoscere è "**101**", pertanto osservando lo schema in figura si nota che, inizialmente siamo nello stato **q0**, se in input si riceve **0** ci si muove verso un nuovo stato: **q1** mentre invece se in input si riceve **1** ci si muove verso un altro stato: **q2**. Dallo stato **q1** a prescindere da quale possa essere l'ingresso ci si porta nello stato **q5** (è possibile notare che non c'è la possibilità di trovare la sequenza desiderata dato che il primo bit in ingresso, era 1) e successivamente ci si riporta a **q0** non avendo riconosciuto la sequenza. Dallo stato **q2** invece, è possibile individuare due alternative: se venisse dato in ingresso **1**, ci si riporterebbe allo stato **q5** e successivamente, a prescindere dal valore dell'ingresso ci si riporterebbe in **q0** non essendo possibile individuare la sequenza. Nel secondo caso invece, se venisse dato in ingresso **0** si transiterebbe nello stato **q4** e da quest'ultimo a prescindere dal valore di ingresso, si transiterebbe nello stato **q0**, con la differenza però che se l'ingresso fosse **1**, allora si sarebbe individuata la sequenza e l'uscita sarà alta. In nessun altro caso, come è possibile riscontrare anche in figura, l'uscita sarà alta!

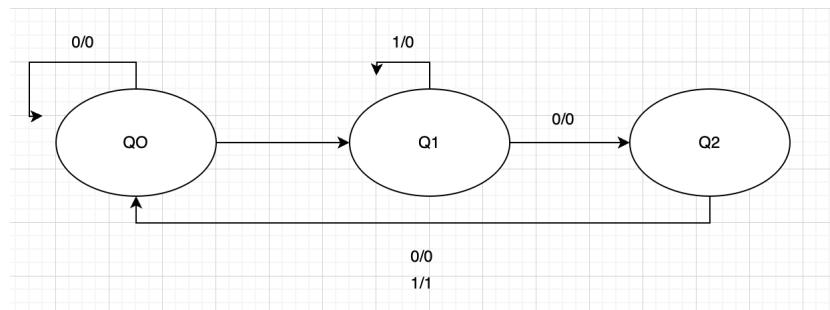


Figure 2.2: Riconoscitore di Sequenza Parzialmente sovrapposto

In un riconoscitore di sequenza **parzialmente sovrapposto**, alcuni stati **possono avere più di una transizione possibile** per uno o più input. Tuttavia, non tutti gli stati permettono questa sovrapposizione. Questo concetto implica che mentre alcuni stati possono avere transizioni multiple, altri stati continuano ad avere una transizione univoca. In altre parole, la sovrapposizione degli stati è permessa solo in modo parziale nell'automa. L'uso di riconoscitori parzialmente sovrapposti **può**

essere utile in alcune situazioni in cui la complessità della modellazione richiede la possibilità di rappresentare più stati contemporaneamente in determinate condizioni. Questa flessibilità aggiuntiva può rendere più semplice descrivere comportamenti complessi che potrebbero essere difficili da modellare con automi a stati finiti non sovrapposti. Tuttavia, va notato che l'introduzione della sovrapposizione può anche aumentare la complessità dell'analisi e della progettazione del sistema. Com'è possibile notare, l'automa è più semplice di quello visto in precedenza. La sequenza che intendiamo riconoscere è ancora **"101"**, pertanto osservando lo schema in figura si nota che, inizialmente siamo nello stato **q0** e vi permaniamo se in ingresso viene dato **0**, se in input si riceve **1** ci si muove verso un nuovo stato: **q1**. Si permane in quest'ultimo stato fintanto che l'ingresso è **1**, mentre invece si transita nello stato **q2** nel momento in cui l'ingresso sarà pari a **0**. Infine, a prescindere da quale sia l'ingresso si transiterà dallo stato **q2 a q0**, con la differenza che se l'ingresso sottoposto sarà **1** avremmo riconosciuto la sequenza, e pertanto l'uscita dell'automa sarà alta. In caso contrario invece, l'uscita sarà bassa.

2.1.2 Implementazione

In primo luogo, abbiamo implementato il **riconoscitore di sequenza** definendo un'interfaccia chiamata **riconoscitore di sequenza** per la quale sono stati definiti segnali di ingresso e di uscita (rispettivamente **i** e **y**), tra questi segnali, **m** consente di discriminare la modalità nella quale lavoreremo: **parzialmente sovrapposta** o **non sovrapposta**, determinando dunque il **modo di esecuzione**, ed infine il segnale **a** che rappresenta il **clock** della nostra macchina sequenziale.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.all;
4
5 entity riconoscitore_sequenza is
6   port(  i : in STD_LOGIC;
7          m : in STD_LOGIC;
8          a : in STD_LOGIC;
9          y : out STD_LOGIC
10         );
11 end riconoscitore_sequenza;
```

Listing 2.1: Implementazione del riconoscitore di sequenza

Per la descrizione del componente abbiamo utilizzato il costrutto architecture di tipo **behavioral** che permette di descrivere il comportamento del componente senza guardare ai componenti più piccoli che lo realizzano. In primo luogo, è stato definito un **tipo enumerativo**, che consentisse di tener traccia degli stati individuati all'atto

della progettazione dell'automa e sono stati definiti dei segnali d'appoggio al fine di tener traccia dello **stato attuale**, dello **stato prossimo** e dell'**uscita ottenuta durante una specifica transizione di stato**. Per la descrizione del comportamento, abbiamo utilizzato il costrutto **"case...when"**, in particolare, con l'obiettivo di discriminare la modalità di funzionamento dell'automa: si lavorerà con un riconoscitore non sovrapposto quando **m = 0** mentre al contrario si lavorerà con un riconoscitore parzialmente sovrapposto quando **m = 1**. Internamente al **"case...when"** ne è stato innestato un secondo al fine di specificare il comportamento dell'automa a seconda dello stato nel quale attualmente si trova, proprio come spiegato nella sezione precedente nella quale è stato dettagliata ciascuna transizione di stato a seguito di ingressi, riportando lo stato prossimo e l'uscita corrente. Di seguito dunque, si ha semplicemente una traduzione VHDL di quanto descritto nel paragrafo **"Progetto e architettura"**.

```

1 architecture behavioural of riconoscitore_sequenza is
2     type stato is (q0, q1, q2, q3, q4);
3     signal stato_corrente : stato := q0;
4     signal stato_prossimo : stato;
5     signal uscita_corrente : STD_LOGIC := 'U';
6
7 begin
8     calcolo_transizione : process(m, i, stato_corrente)
9     begin
10        case m is
11            when '0' =>
12                case stato_corrente is
13                    when q0 =>
14                        if(i = '1') then
15                            stato_prossimo <= q2;
16                            uscita_corrente <= '0';
17                        else
18                            stato_prossimo <= q1;
19                            uscita_corrente <= '0';
20                        end if;
21                    when q1 =>
22                        stato_prossimo <= q3;
23                        uscita_corrente <= '0';
24                    when q2 =>
25                        if(i = '1') then
26                            stato_prossimo <= q3;
```

```

27                         uscita_corrente <= '0';
28
29                         stato_prossimo <= q4;
30                         uscita_corrente <= '0';
31
32         end if;
33
34     when q3 =>
35             stato_prossimo <= q0;
36             uscita_corrente <= '0';
37
38     when q4 =>
39             stato_prossimo <= q0;
40             if(i = '1') then
41                 uscita_corrente <= '1';
42             else
43                 uscita_corrente <= '0';
44
45             end if;
46
47     when others =>
48         null;
49
50     end case;
51
52 when '1' =>
53
54     case stato_corrente is
55
56         when q0 =>
57             if(i = '1') then
58                 stato_prossimo <= q1;
59                 uscita_corrente <= '0';
60             else
61                 stato_prossimo <= q0;
62                 uscita_corrente <= '0';
63
64             end if;
65
66         when q1 =>
67             if(i = '1') then
68                 stato_prossimo <= q1;
69                 uscita_corrente <= '0';
70             else
71                 stato_prossimo <= q2;
72                 uscita_corrente <= '0';
73
74             end if;
75
76         when q2 =>
77             stato_prossimo <= q0;
78             if(i = '1') then
79                 uscita_corrente <= '1';
80             else

```

```

68                      uscita_corrente <= '0';
69      end if;
70      when others =>
71          null;
72      end case;
73      when others =>
74          null;
75      end case;
76  end process;

```

Listing 2.2: Implementazione del comportamento del riconoscitore di sequenza

Infine, l'architecture termina con il seguente process che consente di assegnare lo stato prossimo allo stato corrente ed infine consente di assegnare l'uscita correntemente calcolata al segnale **y** d'uscita al riconoscitore, il tutto verrà fatto all'avvenire di un fronte di salita del segnale di clock, rappresentato dal segnale **a**.

```

1
2      tempificazione : process(a)
3      begin
4          if rising_edge(a) then
5              stato_corrente <= stato_prossimo;
6              y <= uscita_corrente;
7          end if;
8      end process;
9
10 end behavioural;

```

Listing 2.3: Implementazione del comportamento del riconoscitore di sequenza

2.1.3 Simulazione

Per simulare il componente è stato realizzato un **testbench**.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.all;
4
5 entity riconoscitore_sequenza_tb is
6 end riconoscitore_sequenza_tb;
7
8 architecture rtl of riconoscitore_sequenza_tb is
9     component riconoscitore_sequenza
10    port(    i : in STD_LOGIC;

```

```

11          m : in STD_LOGIC;
12          a : in STD_LOGIC;
13          y : out STD_LOGIC
14      );
15  end component;
16
17  signal datain : STD_LOGIC := 'U';
18  signal mode : STD_LOGIC := 'U';
19  signal timing : STD_LOGIC := 'U';
20  signal dataout : STD_LOGIC := 'U';
21
22  begin
23      dut : riconoscitore_sequenza
24      port map(
25          i => datain,
26          m => mode,
27          a => timing,
28          y => dataout
29      );
30
31      stim_proc : process
32      begin
33          -- m = 1 parzialmente sovrapposta.
34          mode <= '1';
35          timing <= '0';
36          wait for 5 ns;
37          timing <= '1';
38          wait for 5 ns;
39          timing <= '0';
40
41          wait for 5 ns;
42          datain <= '0';
43          wait for 5 ns;
44          timing <= '1';
45          wait for 5 ns;
46          timing <= '0';
47          wait for 5 ns;
48          datain <= '1';
49          wait for 5 ns;
50          timing <= '1';
51          wait for 5 ns;

```

```

52      timing <= '0';
53      wait for 5 ns;
54      datain <= '1';
55      wait for 5 ns;
56      timing <= '1';
57      wait for 5 ns;
58      timing <= '0';
59      wait for 5 ns;
60      datain <= '0';
61      wait for 5 ns;
62      timing <= '1';
63      wait for 5 ns;
64      timing <= '0';
65      wait for 5 ns;
66      datain <= '1';
67      wait for 5 ns;
68      timing <= '1';
69      wait for 5 ns;
70      timing <= '0';
71      wait for 10 ns;--01101

72
73      -- m = 0 non sovrapposto.
74      mode <= '0';
75      timing <= '0';
76      --wait for 5 ns;
77      --timing <= '1';
78      --wait for 5 ns;
79      --timing <= '0';

80
81      wait for 5 ns;
82      datain <= '0';
83      wait for 5 ns;
84      timing <= '1';
85      wait for 5 ns;
86      timing <= '0';
87      wait for 5 ns;
88      datain <= '1';
89      wait for 5 ns;
90      timing <= '1';
91      wait for 5 ns;
92      timing <= '0';

```

```
93      wait for 5 ns;
94      datain <= '0';
95      wait for 5 ns;
96      timing <= '1';
97      wait for 5 ns;
98      timing <= '0';
99      wait for 5 ns;
100     datain <= '1';
101     wait for 5 ns;
102     timing <= '1';
103     wait for 5 ns;
104     timing <= '0';
105     wait for 5 ns;
106     datain <= '0';
107     wait for 5 ns;
108     timing <= '1';
109     wait for 5 ns;
110     timing <= '0';
111     wait for 5 ns;
112     datain <= '1';
113     wait for 5 ns;
114     timing <= '1';
115     wait for 5 ns;
116     timing <= '0';
117     wait for 5 ns;
118     datain <= '0';
119     wait for 5 ns;
120     timing <= '1';
121     wait for 5 ns;
122     timing <= '0';
123     wait for 5 ns;
124     datain <= '1';
125     wait for 5 ns;
126     timing <= '1';
127     wait for 5 ns;
128     timing <= '0';
129     wait for 5 ns;
130     datain <= '0';
131     wait for 5 ns;
132     timing <= '1';
133     wait for 5 ns;
```

```

134     timing <= '0';
135     wait for 10 ns;--010101010
136
137
138     wait;
139
140   end process;
141
142 end rtl;

```

Listing 2.4: Testbench del riconoscitore di sequenza

In primo luogo, è stata dichiarata una entity, **riconoscitore sequenza tb**, vuota dato che non rappresenta un oggetto fisico da realizzare, ma anzi è necessaria solo al fine di condurre i test. Successivamente nell'architecture è stato dichiarato un componente che rappresenta proprio il riconoscitore di sequenza che si intende testare e i segnali di datain, mode, timing e dataout tutti inizializzati al valore "U" (che rappresenta undefined). È stato successivamente effettuato il **port map**, al fine di permettere il mapping tra i segnali di input e output del componente definito in questo file e il componente definito nel file "riconoscitore di sequenza.vhd" analizzato precedentemente. In seguito, è stato definito un processo identificato dalla label **stim proc** nel quale abbiamo provveduto a dare in ingresso determinati stimoli al fine di testare il componente implementato. Ad esempio: è stato impostato mode $\text{:= } '1'$ impostando la modalità di esecuzione parzialmente sovrapposta successivamente sono stati dati in ingresso una serie di "datain" al fine di determinare a simulazione eseguita cosa succedesse ai segnali di output. È stato condotto un secondo test, del tutto analogo, impostando la modalità di utilizzo a non sovrapposta, dando poi una serie di stimoli in ingresso al fine di determinare l'uscita del componente. I risultati sono riportati nelle figure sottostanti:



Figure 2.3: Test in modalità parzialmente sovrapposta

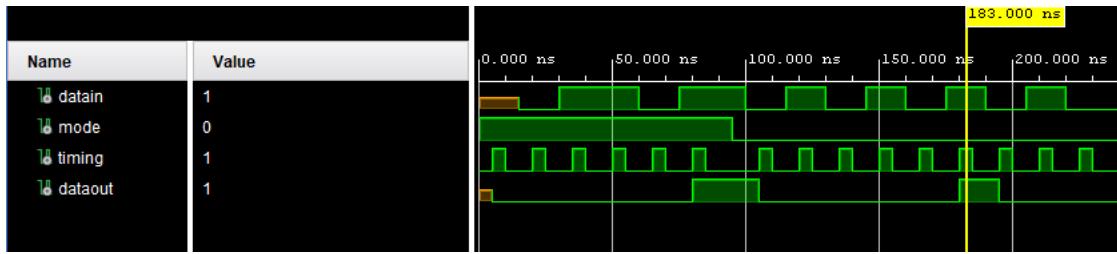


Figure 2.4: Test in modalità non sovrapposta

2.2 Esercizio 3.2 - Riconoscitore di sequenza on board

Sintetizzare e implementare su board la rete sviluppata al punto precedente, utilizzando uno switch S1 per codificare l'input i e uno switch S2 per codificare il modo M, in combinazione con due pulsanti B1 e B2 utilizzati rispettivamente per acquisire l'input da S1 e S2 in sincronismo con il segnale di temporizzazione A, che deve essere ottenuto a partire dal clock della board. Infine, l'uscita Y può essere codificata utilizzando un led.

2.2.1 Progetto e architettura

La componente del riconoscitore è la stessa utilizzata per l'esercizio precedente; in aggiunta sono state introdotte alcune componenti:

- Un **sistema** esterno direttamente connesso alla board;
- Un'**unità di controllo** che permette l'abilitazione dei valori di input;
- Un **Button Debouncer**, la cui implementazione è data, che permette di pulire i segnali di ingresso alla pressione del pulsante.

Il progetto prevede che con gli switch della board si mappino il segnale di ingresso al riconoscitore e la modalità con cui questo agisce, mentre i pulsanti permettono al valore di ingresso di essere letto e di abilitarlo. L'output del riconoscitore verrà visualizzato su uno dei led della board.

2.2.2 Implementazione

Analizziamo le componenti nuove:

Unità di controllo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;

```

```

3
4 entity unita_di_controllo is
5     port(    switch_in_1 : in std_logic;
6             switch_in_2 : in std_logic;
7             button_1 : in std_logic;
8             button_2 : in std_logic;
9             clk_cu : in std_logic;
10            switch_out_data : out std_logic;
11            switch_out_mode : out std_logic
12        );
13 end unita_di_controllo;
14
15 architecture behavioral of unita_di_controllo is
16     signal temp_data : std_logic;
17     signal temp_mode : std_logic;
18
19 begin
20     process(clk_cu, button_1, button_2, switch_in_1,
21             switch_in_2)
22         begin
23             if(rising_edge(clk_cu)) then
24                 if(button_1 = '1') then
25                     temp_data <= switch_in_1;
26                 end if;
27                 if(button_2 = '1') then
28                     temp_mode <= switch_in_2;
29                 end if;
30             end if;
31         end process;
32
33     switch_out_data <= temp_data;
34     switch_out_mode <= temp_mode;
35 end behavioral;

```

Quello che questa componente fa è semplicemente far passare i valori in ingresso ed ottenerli in output se i valori di abilitazione, in questo caso button_1 e button_2, sono alti.

Sistema

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.all;
4
5 entity system is
6     port ( s1 : in std_logic;
7             s2 : in std_logic;
8             b1 : in std_logic;
9             b2 : in std_logic;
10            clock100MHZ : in std_logic;
11            uscita : out std_logic
12        );
13 end system;
14
15 architecture structural of system is
16     signal data_in : std_logic;
17     signal mode : std_logic;
18     signal cleared_b1 : std_logic;
19     signal cleared_b2 : std_logic;
20
21     component ButtonDebouncer is
22         generic (
23             CLK_period: integer := 10; -- periodo del clock
24                                     (della board) in nanosecondi
25             btn_noise_time: integer := 10000000 -- durata
26                                     stimata dell'oscillazione del bottone in
27                                     nanosecondi
28                                     -- il valore
29                                     di default
30                                     10
31                                     millisecondi
32         );
33         Port ( RST : in STD_LOGIC;
34                 CLK : in STD_LOGIC;
35                 BTN : in STD_LOGIC;
36                 CLEARED_BTN : out STD_LOGIC);
37     end component;
38
39     component unita_di_controllo is
40         port( switch_in_1 : in std_logic;
41               switch_in_2 : in std_logic;

```

```

36         button_1 : in std_logic;
37         button_2 : in std_logic;
38         clk_cu : in std_logic;
39         switch_out_data : out std_logic;
40         switch_out_mode : out std_logic
41     );
42 end component;
43
44 component riconoscitore_sequenza is
45     port(    i : in STD_LOGIC;
46             m : in STD_LOGIC;
47             a : in STD_LOGIC;
48             y : out STD_LOGIC
49 );
50 end component;
51
52 begin
53     debouncer_1 : ButtonDebouncer
54     port map(    RST => '0',
55                 CLK => clock100MHZ,
56                 BTN => b1,
57                 CLEARED_BTN => cleared_b1
58 );
59
60     debouncer_2 : ButtonDebouncer
61     port map(    RST => '0',
62                 CLK => clock100MHZ,
63                 BTN => b2,
64                 CLEARED_BTN => cleared_b2
65 );
66
67     unita : unita_di_controllo
68     port map(    switch_in_1 => s1,
69                 switch_in_2 => s2,
70                 button_1 => cleared_b1,
71                 button_2 => cleared_b2,
72                 clk_cu => clock100MHZ,
73                 switch_out_data => data_in,
74                 switch_out_mode => mode
75 );
76

```

```

77      rico : riconoscitore_sequenza
78      port map(    i => data_in,
79                  m => mode,
80                  a => clock100MHZ,
81                  y => uscita
82      );
83
84 end structural;

```

2.2.3 Timing Analysis

Per effettuare la timing analysis in Vivado è necessario prima di tutto effettuare la sintesi, che consentirebbe di eseguire già l'analisi ma con delle somme approssimate, quindi abbiamo effettuato anche l'implementazione. Nella schermata “Report Timing Summary” è possibile configurare i parametri della timing analysis, che specificano il tipo di report da generare e il contenuto da mostrare una volta eseguita l'analisi. In particolare, abbiamo selezionato min e max, come “Path delay type”, per misurare il Worst Negative Slack (WNS), cioè il tempo che impiega un segnale di input a stabilizzarsi prima del fronte successivo del clock, tale che le uscite raggiungano il valore desiderato e il Worst Hold Slack (WHS), cioè il tempo per cui un segnale di input deve restare stabile dopo il fronte del clock per consentire all'output di raggiungere il valore desiderato. Lo slack indica la differenza tra require time e arrival time. In questa analisi viene misurato anche il Worst Pulse Width Slack (WPWS) che indica il peggiore tra tutti i controlli considerando i ritardi sia minimi che massimi. Il parametro maximum number of path per clock indica quanti percorsi possono essere analizzati simultaneamente in un singolo ciclo di clock, mentre il maximum number of worst paths per endpoint consente di limitare il numero massimo di percorsi peggiori analizzati per ciascun endpoint.

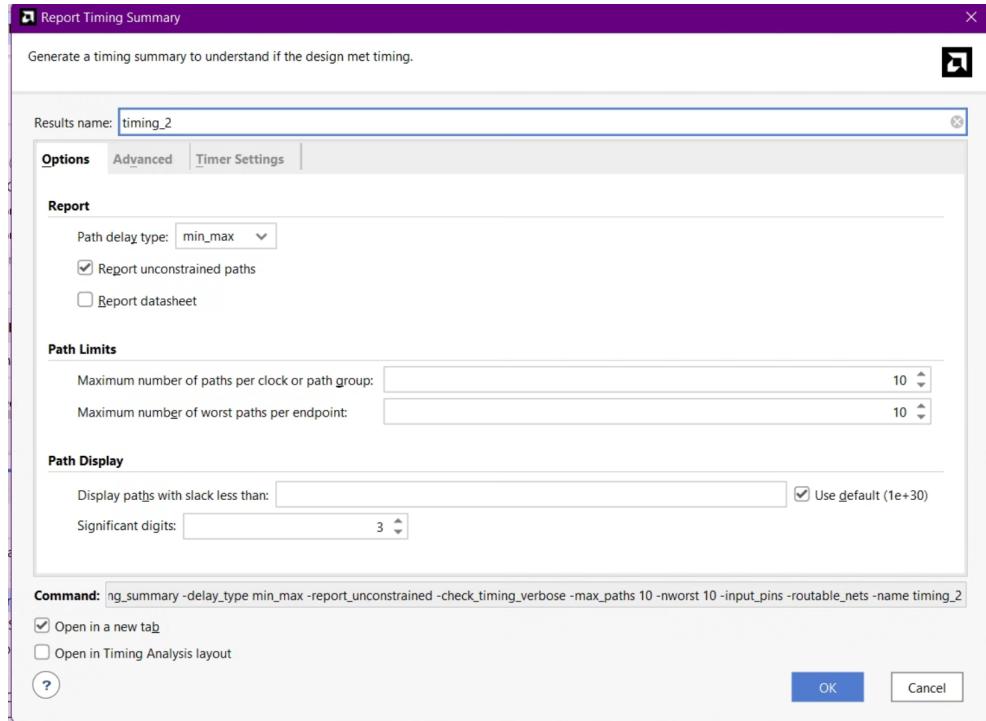


Figure 2.5: Report Timing Summary

È importante che il WNS sia positivo, perché altrimenti vuol dire che il percorso sarà fallito. Di seguito sono riportati i risultati per quanto riguarda il clock creato:

Design Timing Summary			
	Setup	Hold	Pulse Width
	Worst Negative Slack (WNS): 4.938 ns	Worst Hold Slack (WHS): 0.256 ns	Worst Pulse Width Slack (WPWS): 4.500 ns
	Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
	Total Number of Endpoints: 196	Total Number of Endpoints: 196	Total Number of Endpoints: 77
All user specified timing constraints are met.			

Figure 2.6: U

na volta controllato questo risultato, abbiamo calcolato anche la FMAX, cioè la frequenza massima di funzionamento. Essa non è fornita esplicitamente nei report ma l'abbiamo stimata con la seguente formula:

$$\frac{1}{T-WNS}$$

dove T è il periodo del clock target. Per trovare questo valore è possibile diminuire progressivamente il periodo del clock di design, finché non si ottiene un WNS negativo. In particolare, abbiamo diminuito il periodo fino a 3.5 ns con una forma d'onda con fronte di salita in 0 ns e fronte di discesa a 1.75 ns period 3.50 waveform 1.75 Il valore approssimato è 281 MHZ. Oltre questa frequenza i vincoli temporali non sono più rispettati.

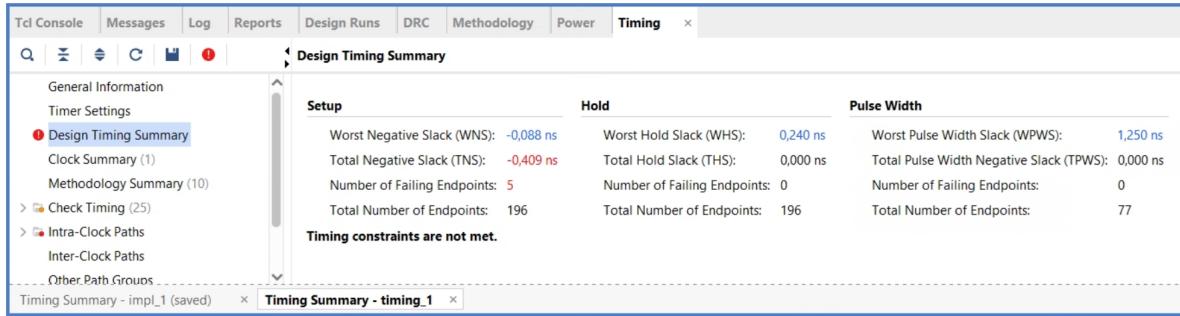


Figure 2.7

2.3 Esercizio 4.1 - Shift Register

Progettare, implementare in VHDL e testare mediante simulazione un registro a scorrimento di N bit in grado di shiftare a destra o a sinistra di un numero Y variabile di posizioni a seconda di una opportuna selezione. In particolare, i valori possibili di Y sono 1 e 2. L'utente tramite selezione deve scegliere di quante posizioni shiftare. Il componente deve essere realizzato utilizzando sia a) approccio comportamentale sia un b) approccio strutturale. Nota: il numero di bit del registro deve essere implementato come un generic, e dall'esterno deve poter essere scelta la modalità di funzionamento mediante opportuni segnali di selezione.

2.3.1 Progetto e architettura

Al fine di realizzare lo shift register si è proceduto mediante due progetti diversi per l'implementazione comportamentale e strutturale.

Dal punto di vista **comportamentale**, il registro a scorrimento progettato è caratterizzato dai seguenti ingressi:

- `EI`: rappresenta il bit in input.
- `ck`: associato al clock del sistema, determina il momento in cui il bit in input viene acquisito.
- `rst`: segnale di reset che riporta il registro a uno stato predefinito quando attivo.
- `choice`: permette di selezionare uno shift a destra o a sinistra.
- `y`: indica il numero di posizioni per lo shift.

L'uscita del registro è unica:

- **memo:** rappresenta il contenuto corrente dei registri. Questa uscita fornisce il valore attuale dei bit nel registro dopo ogni ciclo di clock e operazione di shift.

Invece per quanto riguarda lo **strutturale** sono stati utilizzati N **flip-flop** di tipo **D** come componenti elementari dell'intero sistema, assieme ad N **mux 4:1** collegati opportunamente ai flip-flop per permettere di selezionare la modalità di shift tra quattro opzioni: shift a sinistra di una posizione, shift a sinistra di due posizioni, shift a destra di una posizione e shift a destra di due posizioni. Il registro a scorrimento è caratterizzato dai seguenti ingressi:

- **ei:** rappresenta il bit in input che viene inserito nel registro.
- **clock:** associato al clock del sistema, determina il momento in cui il bit in input viene acquisito.
- **reset:** segnale di reset che riporta il registro a uno stato predefinito quando attivo.

Il registro presenta un'unica uscita:

- **eo:** Indica il contenuto corrente dei registri. L'uscita **eo** fornisce il valore attuale dei bit nel registro dopo ogni ciclo di clock.

La struttura vede N multiplexer 4:1 i cui ingressi sono collegati nel seguente modo:

- il primo ingresso del multiplexer, che rappresenta lo shift a sinistra di una posizione, è collegato all'uscita del flip-flop successivo, dunque, ad $y_temp(i+1)$.
- il secondo ingresso del multiplexer, che rappresenta lo shift a sinistra di due posizioni, è collegato all'uscita del flip-flop di due posizioni in avanti, dunque, ad $y_temp(i+2)$.
- il terzo ingresso del multiplexer, che rappresenta lo shift a destra di una posizione, è collegato all'uscita del flip-flop precedente, dunque, $y_temp(i-1)$.
- il quarto ingresso del multiplexer, che rappresenta lo shift a destra di due posizioni, è collegato all'uscita del flip-flop di due posizioni indietro, dunque, $y_temp(i-2)$.

Fanno **eccezione** i primi due mux e gli ultimi due, in quanto prevedono, rispettivamente, il primo e il secondo ingresso pari ad **ei**, il secondo ingresso pari ad **ei**, il quarto ingresso pari ad **ei**, il terzo e il quarto ingresso pari ad **ei**.

Inoltre, le uscite dei multiplexer sono collegate agli ingressi dei flip-flop.

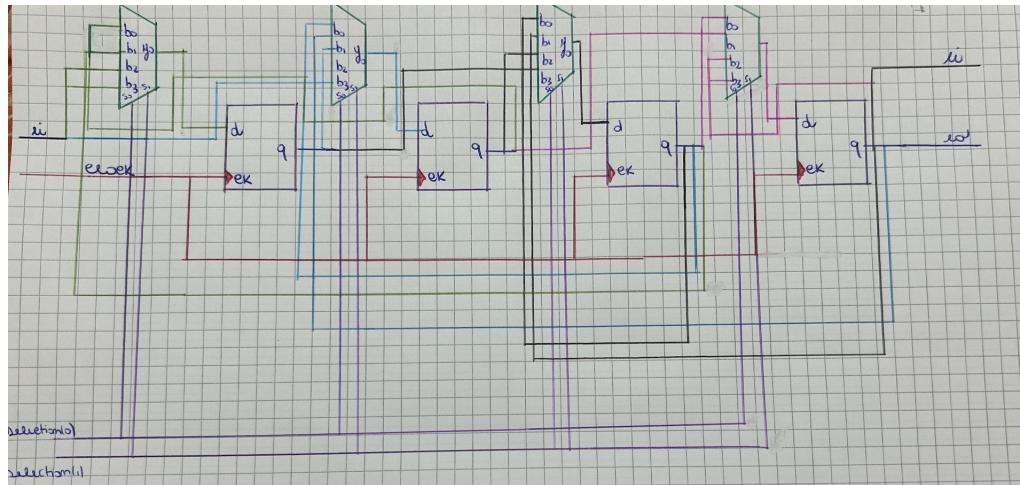


Figure 2.8: Schema dello shift register.

2.3.2 Implementazione

La prima implementazione che vedremo è quella della parte comportamentale, il codice è il seguente:

```

1
2 library IEEE;
3 use IEEE.std_logic_1164.ALL;
4 use work.all;
5
6 entity shiftregisterbehavioural is
7     generic(
8         N : integer := 8
9     );
10    port (
11        EI, ck, rst : in STD_LOGIC;
12        choice : in STD_LOGIC; -- 0 shift dx, 1 shift sx
13        y : in std_logic_vector(1 downto 0);
14        memo : out STD_LOGIC_VECTOR(N-1 downto 0);
15    );
16 end shiftregisterbehavioural;
17
18 architecture behavioural of shiftregisterbehavioural is
19     signal temp : STD_LOGIC_VECTOR(N-1 downto 0) := (others =>
20             '0');

```

```

21 begin
22     process(ck)
23         variable shifttemp : STD_LOGIC_VECTOR(N-1 downto 0) := 
24             (others => '0');
25         begin
26             if rising_edge(ck) then
27                 if rst = '1' then
28                     temp <= (others => '0');
29                 else
30                     shifttemp := (others => '0');
31                 case choice is
32                     when '0' =>
33                         if y = "01" then
34                             shifttemp(0) := EI;
35                             for i in 1 to (N-1) loop
36                                 shifttemp(i) := temp(i-1);
37                             end loop;
38                         elsif y = "10" then
39                             shifttemp(0) := EI;
40                             shifttemp(1) := EI;
41                             for i in 2 to (N-1) loop
42                                 shifttemp(i) := temp(i-2);
43                             end loop;
44                         end if;
45                     temp <= shifttemp;
46                     when '1' =>
47                         if y = "01" then
48                             shifttemp(N-1) := EI;
49                             for i in 0 to (N-2) loop
50                                 shifttemp(i) := temp(i+1);
51                             end loop;
52                         elsif y = "10" then
53                             shifttemp(N-1) := EI;
54                             shifttemp(N-2) := EI;
55                             for i in 0 to (N-3) loop
56                                 shifttemp(i) := temp(i+2);
57                             end loop;
58                         end if;
59                         temp <= shifttemp;
60                         when others =>
61                             null;

```

```

61      end case;
62    end if;
63  end if;
64 end process;
65
66 memo <= temp;
67 end behavioural;

```

Listing 2.5: Shift Register implementato mediante l'architettura comportamentale.

L'entità che rappresenta il registro a scorrimento fa uso dei generics, in particolare N rappresenta il numero di bit che possono essere memorizzati contemporaneamente. I segnali di ingresso e di uscita sono gli stessi di cui abbiamo discusso precedentemente. E' stato dichiarato un processo che viene attivato ad ogni fronte di salita del clock, in quanto dobbiamo avere uno shift ad ogni colpo. Sono stati, inoltre, dichiarati un segnale di appoggio, temp, che memorizzerà il contenuto dei registri, e una variabile d'appoggio, shifttemp, che si occuperà di effettuare gli shift. Distinguiamo mediante il costrutto case il caso in cui choice è 0 (shift a destra), e il caso in cui è 1 (shift a sinistra), inoltre, gli if ci permettono di verificare il valore assunto da y, se "01" avremo uno shift di una posizione, se "10" di due posizioni. Per lo shift a destra di una posizione inseriamo nella prima posizione di shifttemp EI, e con un for sulle altre posizioni memorizziamo in shifttemp(i) il valore di shifttemp(i-1). Per lo shift a destra di due posizioni inseriamo nella prima e seconda posizione di shifttemp EI, e con un for sulle altre posizioni memorizziamo in shifttemp(i) il valore di shifttemp(i-2). Per lo shift a sinistra di una posizione inseriamo nell'ultima posizione di shifttemp EI, e con un for sulle altre posizioni memorizziamo in shifttemp(i) il valore di shifttemp(i+1). Per lo shift a sinistra di due posizioni inseriamo nella prima e seconda posizione di shifttemp EI, e con un for sulle altre posizioni memorizziamo in shifttemp(i) il valore di shifttemp(i+2). Alla fine di ciascun if a temp verrà assegnato il valore di shifttemp, e al termine del processo memo memorizzerà il valore di temp.

Passiamo all'implementazione della parte strutturale. Come è stato già detto i componenti basilari dell'architettura sono i multiplexer 4:1, già implementati negli esercizi precedenti, e i flip-flop di tipo D, su cui verrà posta la nostra attenzione:

```

1
2 library IEEE;
3 use IEEE.std_logic_1164.ALL;
4 use work.all;
5

```

```

6 entity flipflopd is
7     port (
8         ck      : in STD_LOGIC;
9         d       : in STD_LOGIC;
10        q       : out STD_LOGIC
11    );
12 end flipflopd;
13
14 architecture dataflow of flipflopd is
15 begin
16     process (ck)
17     begin
18         if rising_edge(ck) then
19             q <= d;
20         end if;
21     end process;
22 end dataflow;

```

Listing 2.6: implementazione del flip-flop D

L'uscita q del flip flop si limita a seguire l'ingresso d quando si verifica il fronte di salita del clock. Il collegamento tra flip-flop e multiplexer è esplicitato nel seguente codice:

```

1
2 library IEEE;
3 use IEEE.std_logic_1164.ALL;
4 use IEEE.numeric_std.ALL;
5
6 entity shiftregisterstructural is
7     generic (
8         n : integer := 8
9     );
10    port (
11        ei, clock, reset: in std_logic;
12        selection : in std_logic_vector(1 downto 0); -- 00
13            01 sinistra 1, 01 sinistra 2, 10 destra 1, 11 destra 2
14        eo : out std_logic_vector(n-1 downto 0)
15    );
16 end shiftregisterstructural;
17 architecture structural of shiftregisterstructural is

```

```

18     signal temp : std_logic_vector(n-1 downto 0) := (others =>
19         '0');
20     signal y_temp: std_logic_vector(n-1 downto 0) := (others
21         => '0');
22     signal s : std_logic_vector(1 downto 0);
23
24
25     component mux41 is
26         port (
27             b0, b1, b2, b3, s0, s1 : in std_logic;
28             y0 : out std_logic
29         );
30     end component;
31
32
33     component flipflop is
34         port (
35             ck : in std_logic;
36             d : in std_logic;
37             q : out std_logic
38         );
39     end component;
40
41
42 begin
43     -- Connessione del segnale di selezione ai multiplexer
44     s <= selection;
45
46     -- Logica di reset asincrono per i flip-flop
47     process(clock, reset)
48     begin
49         if reset = '1' then
50             y_temp <= (others => '0');
51         elsif rising_edge(clock) then
52             y_temp <= temp;
53         end if;
54     end process;
55
56     mux0: mux41
57         port map(
58             b0 => y_temp(1),
59             b1 => y_temp(2),
60             b2 => ei,
61             b3 => ei,
62             s0 => s(0),
63             s1 => s(1),
64             y0 => y
65         );

```

```

57           s0 => s(0),
58           s1 => s(1),
59           y0 => temp(0)
60       );
61
62   mux1: mux41
63       port map(
64           b0 => y_temp(2),
65           b1 => y_temp(3),
66           b2 => y_temp(0),
67           b3 => ei,
68           s0 => s(0),
69           s1 => s(1),
70           y0 => temp(1)
71       );
72
73   mux(n-2): mux41
74       port map(
75           b0 => y_temp(n-1),
76           b1 => ei,
77           b2 => y_temp(n-3),
78           b3 => y_temp(n-4),
79           s0 => s(0),
80           s1 => s(1),
81           y0 => temp(n-2)
82       );
83
84   mux(n-1): mux41
85       port map(
86           b0 => ei,
87           b1 => ei,
88           b2 => y_temp(n-2),
89           b3 => y_temp(n-3),
90           s0 => s(0),
91           s1 => s(1),
92           y0 => temp(n-1)
93       );
94
95   creazione_mux: for i in 2 to (n-3) generate
96       mux : mux41
97           port map (

```

```

98          b0 => y_temp(i+1),
99          b1 => y_temp(i+2),
100         b2 => y_temp(i-1),
101         b3 => y_temp(i-2),
102         s0 => s(0),
103         s1 => s(1),
104         y0 => temp(i)
105     );
106   end generate creazione_mux;
107
108   creazione_flip_flop : for i in 0 to (n-1) generate
109     flip_flop : flipflop
110       port map (
111         ck => clock,
112         d => temp(i),
113         q => y_temp(i)
114     );
115   end generate creazione_flip_flop;
116
117
118   eo <= y_temp;
119
120 end structural;

```

Listing 2.7: Shift Register implementato mediante l’architettura strutturale.

Anche in questo caso si fa affidamento all’utilizzo dei generics per specificare il numero di flip-flop e multiplexer che verranno sfruttati come componenti. Vengono sfruttati in questo caso due segnali di appoggio: temp, che rappresenta il vettore degli ingressi a ciascun flip-flop, e, dunque, le uscite dei multiplexer, ed y_temp, che rappresenta il vettore delle uscite dei flip-flop. Per quanto riguarda la generazione dei multiplexer, ciascuno di essi avrà come ingresso b0 l’uscita del registro successivo y_temp(i+1) (shift a sinistra di 1), ingresso b1 l’uscita del registro due posizioni in avanti y_temp(i+2) (shift a sinistra di due posizioni), ingresso b2 l’uscita del registro precedente y_temp(i-1) (shift a destra di una posizione), e ingresso b3 l’uscita del registro due posizioni indietro y_temp(i-2) (shift a destra di due posizioni). Quando saremo al di fuori dei limiti agli ingressi verrà assegnato il valore di ei. La modalità di shift viene scelta mediante gli ingressi di selezione dei mux, comuni a tutti. Alla fine della generazione dei componenti ad eo verrà mappato il valore del segnale y_temp.

2.3.3 Simulazione

Passiamo ora al test del registro a scorrimento implementato mediante l'architettura comportamentale:

```
1
2 library IEEE;
3 use IEEE.std_logic_1164.ALL;
4 use work.all;
5
6 entity tb_shiftregisterbehavioural is
7 end tb_shiftregisterbehavioural;
8
9 architecture testbench of tb_shiftregisterbehavioural is
10    signal datain, clock, reset : STD_LOGIC := '0';
11    signal scelta : STD_LOGIC := '0'; -- Modifica in STD_LOGIC
12    signal shift : STD_LOGIC_VECTOR(1 downto 0) := "01";
13    signal lucario : STD_LOGIC_VECTOR(7 downto 0);
14
15    constant CLK_PERIOD : time := 100 ns;
16
17    component shiftregisterbehavioural
18        generic (
19            N : integer := 8
20        );
21        port (
22            EI, ck, rst : in STD_LOGIC;
23            choice : in STD_LOGIC; -- Modifica in STD_LOGIC
24            y : in STD_LOGIC_VECTOR(1 downto 0);
25            memo : out STD_LOGIC_VECTOR(N-1 downto 0);
26        );
27    end component;
28
29 begin
30    DUT: shiftregisterbehavioural
31        generic map (
32            N => 8
33        )
34        port map (
35            EI => datain,
36            ck => clock,
```

```

37          rst => reset,
38          choice => scelta,
39          y => shift,
40          memo => lucario,
41      );
42
43  process
44  begin
45      clock <= '0';
46      wait for CLK_PERIOD / 2;
47      clock <= '1';
48      wait for CLK_PERIOD / 2;
49  end process;
50
51  process
52  begin
53      reset <= '1';
54      datain <= '1';
55      scelta <= '0';
56      shift <= "01";
57      wait for 3*CLK_PERIOD / 2;
58
59      reset <= '0';
60      wait for 3*CLK_PERIOD / 2;
61
62      -- shift a destra di 1.
63      scelta <= '0';
64      shift <= "01";
65      wait for 3*CLK_PERIOD / 2;
66
67      -- shift a sinistra di 2.
68      datain <= '0';
69      scelta <= '1';
70      shift <= "10";
71      wait for 3*CLK_PERIOD / 2;
72
73      --shift a destra di 1.
74      datain <= '0';
75      scelta <= '0'; -- Modifica in STD_LOGIC
76      shift <= "01"; -- Modifica in STD_LOGIC_VECTOR
77      wait for 3*CLK_PERIOD / 2;

```

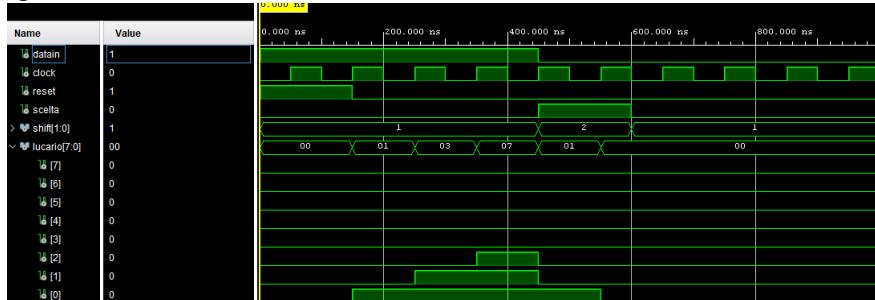
```

78
79      wait;
80  end process;
81
82 end testbench;
83
84 -- 0 shift destra.
85 -- 1 shift sinistra.

```

Listing 2.8: Simulazione dello Shift Register implementato mediante l’architettura comportamentale.

Inizialmente, è stata dichiarata un’entity vuota per il testbench. Questa entity non rappresenta un componente fisico da implementare, ma è pensata solo per eseguire la simulazione e verificare il corretto funzionamento del sistema. Successivamente, nell’architettura rtl, vengono dichiarati i segnali necessari per il testbench, come segnali di clock (clock), di reset (reset), di dati in ingresso (datain), la scelta di direzione dello shift (scelta), il vettore di controllo dello shift (shift), e il vettore di output del registro a scorrimento (lucario). La componente shiftregisterbehavioural viene poi istanziata con il nome DUT (Device Under Test) e vengono collegati i segnali appropriati ai suoi port. La componente viene configurata con un parametro generico N pari a 8. Sono presenti due processi. Il primo gestisce il segnale di clock, generando un’onda quadra con un periodo CLK_PERIOD pari 100 ns. Il secondo processo simula diversi scenari di utilizzo del componente shiftregisterbehavioural. Le operazioni eseguite sono intervallate di $3 \times \text{CLK_PERIOD}/2$ in modo tale da consentire l’esecuzione di alcune operazioni di shift sui registri. In primo luogo è stato abilitato il segnale di reset per pulire i registri, in seguito, con input pari ad 1, è stato effettuato uno shift a destra di 1, poi, con ingresso pari a 0, prima uno shift a sinistra di due, seguito da uno shift a destra di 1. Il risultato della simulazione è il seguente:



Iniziamo con un contenuto dei registri pari a "00000000". Dopo il primo shift a destra di 1, utilizzando un valore di ingresso pari a 1, otteniamo, in successione per ogni colpo di clock nell’intervallo di osservazione specificato, le stringhe "00000001", "00000011" e "00000111". È importante notare che nel primo registro

viene memorizzato il Least Significant Bit (LSB), e così via. Pertanto, per comprendere i valori nei registri, è necessario leggere le stringhe in uscita al contrario. Successivamente, dopo il secondo shift a sinistra di due posizioni con un ingresso pari a 0, passiamo dalla stringa "00000001" alla stringa "00000000". Questo stato rimarrà invariato anche con l'ultimo shift.

Passiamo alla simulazione della logica strutturale:

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use IEEE.numeric_std.ALL;
4 use work.all;
5
6 entity shiftregisterstructural_tb is
7 end shiftregisterstructural_tb;
8
9 architecture rtl of shiftregisterstructural_tb is
10    signal clk, rst, datain : std_logic := '0';
11    signal sel : std_logic_vector(1 downto 0) := "00";
12    signal dataout : std_logic_vector(7 downto 0);
13
14 component shiftregisterstructural
15     generic (
16         n : integer := 8
17     );
18     port (
19         ei, clock, reset: in std_logic;
20         selection : in std_logic_vector(1 downto 0);
21         eo : out std_logic_vector(n-1 downto 0)
22     );
23 end component;
24
25 begin
26     dut: shiftregisterstructural
27         port map (
28             ei => datain,
29             clock => clk,
30             reset => rst,
31             selection => sel,
32             eo => dataout
33         );

```

```

34
35      clock_process: process
36      begin
37          clk <= '0';
38          wait for 5 ns;
39          clk <= '1';
40          wait for 5 ns;
41      end process clock_process;
42
43      stimulus_process: process
44      begin
45          rst <= '1'; -- Reset attivo basso
46          wait for 100 ns;
47          rst <= '0'; -- Rilascia il reset
48          wait for 10 ns;
49
50          -- Esempio di sequenza di input
51          -- shift a sx di 1.
52          datain <= '1'; --10000000
53          sel <= "00";
54          wait for 10 ns;
55
56          -- shift a sx di 2.
57          datain <= '0'; --00100000
58          sel <= "01";
59          wait for 10 ns;
60
61          -- shift a dx di 1.
62          datain <= '1'; --01000001
63          sel <= "10";
64          wait for 10 ns;
65
66          --shift a dx di 2.
67          datain <= '0'; --00000100
68          sel <= "11";
69          wait for 10 ns;
70
71          shift a sx di 1.
72          datain <= '1'; --10000010
73          sel <= "00";
74          wait for 10 ns;

```

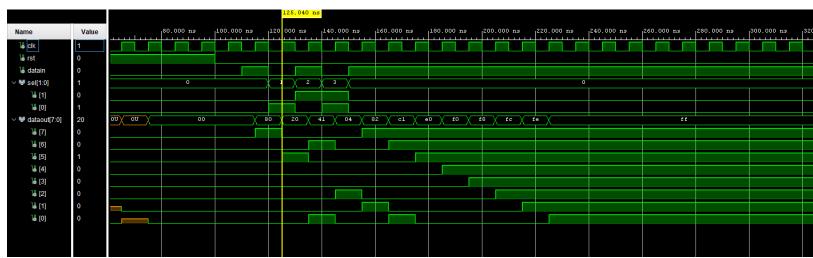
```

75
76      wait;
77  end process stimulus_process;
78
79 end rtl;

```

Listing 2.9: Simulazione dello Shift Register implementato mediante l’architettura strutturale.

Inizialmente, è stata creata un’entity vuota per il testbench. Nell’architettura RTL, sono stati definiti i segnali necessari al testbench, tra cui il clock (clk), il reset (rst), i dati in ingresso (datain), la selezione della modalità di shift (sel), e il vettore di output del registro a scorrimento (eo). Successivamente, la componente shiftregisterstructural è stata istanziata con il nome DUT (Device Under Test), e i segnali appropriati sono stati collegati ai suoi port. La configurazione della componente ha incluso un parametro generico N pari a 8. Il testbench comprende due processi. Il primo gestisce il segnale di clock, generando un’onda quadra con un periodo di 10 ns. Il secondo processo simula vari scenari di utilizzo del componente shiftregisterstructural, intervallando le operazioni ogni 10 ns per consentire l’esecuzione di operazioni di shift sui registri. Nel dettaglio, il processo prevede le seguenti operazioni: abilitazione del segnale di reset per pulire i registri, shift a sinistra di 1 con input pari a 1, shift a sinistra di 2 con input pari a 0, shift a destra di 1 con input pari a 1, shift a destra di 2 con input pari a 0, e infine shift a sinistra di 1 con input pari a 1. Il risultato della simulazione è il seguente:



Partiamo da una configurazione iniziale dei registri con un contenuto "00000000". Dopo il primo shift a sinistra di 1, utilizzando un valore di ingresso pari a 1, otteniamo la stringa "10000000". È rilevante notare che il Least Significant Bit (LSB) viene memorizzato nel primo registro, e così via. Pertanto, per interpretare correttamente i valori nei registri, è essenziale leggere le stringhe in uscita in ordine inverso. Successivamente, durante il secondo shift a sinistra di due posizioni con un ingresso pari a 0, la configurazione passa a "00100000". Dopo uno shift a destra di una posizione, otteniamo la stringa "01000001", seguita da uno shift di due posizioni che restituisce "00000100". Infine, l’ultimo shift a sinistra di uno produce "10000010", e procede ad inserire un '1' ad ogni colpo di clock fino a raggiungere

la configurazione "1111111".

2.4 Esercizio 5.1 - Cronometro

Progettare, implementare in VHDL e testare mediante simulazione un cronometro, in grado di scandire secondi, minuti e ore a partire da una base dei tempi prefissata (es. si consideri il clock a disposizione sulla board). Il progetto deve prevedere la possibilità di inizializzare il cronometro con un valore iniziale, sempre espresso in termini di ore, minuti e secondi, mediante un opportuno ingresso di set, e deve prevedere un ingresso di reset per azzerare il tempo. Il componente deve essere realizzato utilizzando un approccio strutturale, collegando opportunamente dei contatori secondo uno schema a scelta.

2.4.1 Progetto e architettura

Il cronometro offre un mezzo essenziale per misurare con precisione il tempo. Questo dispositivo trova applicazione in svariate situazioni, e la progettazione, potremo adattare il cronometro alle specifiche esigenze, garantendo una gestione del tempo affidabile e precisa nel contesto applicativo desiderato. Con lo scopo di realizzare un cronometro che tenesse conto dei secondi, minuti ed ore, si è pensato di interconnettere tra di loro tre contatori, i primi due modulo 60, rappresentanti secondi e minuti, l'ultimo modulo 24 per le ore. La struttura dei contatori è la stessa, cambia solo il modulo.

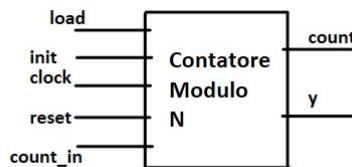


Figure 2.9: Schema del contatore.

I tre contatori offrono la possibilità di inizializzare i propri valori. Inoltre, per quanto riguarda l'interconnessione, le uscite dei contatori si alzano ogni qual volta viene raggiunto il valore massimo di conteggio, dunque, l'uscita del contatore dei secondi è stata collegata al segnale di abilitazione del conteggio dei minuti, mentre l'uscita del contatore dei minuti è stata collegata al segnale di abilitazione del conteggio delle ore. Vediamone lo schema:

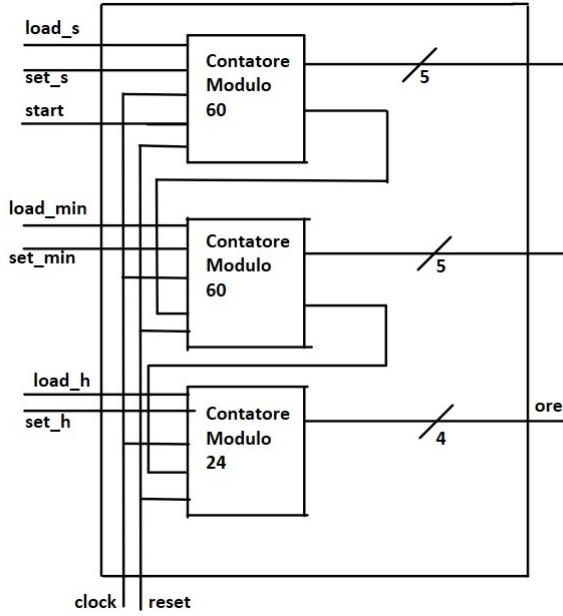


Figure 2.10: Schema del cronometro.

2.4.2 Implementazione

In prima battuta passiamo all’analisi del componente basilare dell’architettura del cronometro, il contatore.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity cont_mod_n is
6     generic(
7         N : positive := 2;
8         max : positive := 60
9     );
10    port( load: in std_logic;
11          init : in std_logic_vector(N-1 downto 0);
12          clock, reset: in std_logic;
13          count_in: in std_logic;
14          count: out std_logic_vector(N-1 downto 0);
15          y : out std_logic
16      );
17 end cont_mod_n;
18
19 architecture behavioural of cont_mod_n is

```

```

20     signal c: std_logic_vector(N downto 0) := (others=>'0');
21     signal temp_end : std_logic := '0';
22
23 begin
24 CM8: process(clock)
25 begin
26
27     if(load = '1') then
28         c(N-1 downto 0) <= init;
29     end if;
30     if(rising_edge(clock)) then
31         if(temp_end = '1') then
32             temp_end <= '0';
33         end if;
34         if(reset = '1') then
35             c <= (others=>'0');
36         else
37             if(count_in = '1') then
38                 c <= std_logic_vector(unsigned(c) + 1);
39                 if(c = std_logic_vector(to_unsigned(max, c'length)))
40                     then
41                         c <= (others => '0');
42                         temp_end <= '1';
43                     end if;
44                 end if;
45             end if;
46         end process;
47
48     count <= c(N-1 downto 0);
49     y <= temp_end;
50
51 end behavioural;

```

Listing 2.10: Implementazione del contatore modulo N.

Sono stati dichiarati due parametri generici:

- N : che indica il modulo del contatore.
- max : che indica il massimo valore che può raggiungere il conteggio per quel determinato contatore.

Per quanto riguarda gli ingressi, essi sono:

- `load` : segnale che ci permette di caricare il valore iniziale del contatore.
- `init` : vettore di N-1 bit che ci permette di inizializzare il valore del contatore.
- `clock` : segnale di temporizzazione del componente.
- `reset` : segnale di reset del conteggio.
- `count_in` : segnale di abilitazione del conteggio.

invece le uscite:

- `count` : vettore di N-1 bit che mantiene il valore attuale di conteggio.
- `y` : bit che indica il raggiungimento del valore massimo di conteggio.

Il comportamento del contatore è definito nella sua architettura comportamentale. Innanzitutto sono stati dichiarati due segnali temporanei, `c` e `temp_end`, adibiti al mantenimento del valore del conteggio e del segnale di terminazione da aggiornare al termine del process. All'interno del process, se il segnale di `load` risulta alto, `c` viene inizializzato al valore presente in `init`. Inoltre, quando il `clock` arriva al fronte, se l'uscita risulta alta viene abbassata, in modo tale da evitare problemi successivi nel collegamento tra contatori. Invece, se è alto il `reset` il conteggio viene azzerato. Infine, se è alta l'abilitazione, `count_in`, `c` viene incrementato di 1, e se raggiunge il massimo valore allora viene azzerato e `temp_end` viene alzato.

Tre contatori vengono collegati tra di loro all'interno dell'architettura strutturale del cronometro, al fine di scandire, rispettivamente, secondi, minuti e ore:

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use IEEE.numeric_std.ALL;
4 use work.all;
5
6 entity timer is
7     port(
8         clk, rst : in std_logic;
9         start : in std_logic;
10        load_s, load_min, load_h : in std_logic;
11        set_s, set_min : in std_logic_vector(5 downto 0);
12        set_h : in std_logic_vector(4 downto 0);
13        secondi, minuti : out std_logic_vector(5 downto 0);

```

```

14          ore : out std_logic_vector(4 downto 0)
15      );
16 end timer;
17
18 architecture structural of timer is
19
20     component cont_mod_n is
21         generic(
22             N : positive := 2;
23             max : positive := 60
24         );
25         port( load : in std_logic;
26                 init : in std_logic_vector(N-1 downto 0);
27                 clock, reset: in std_logic;
28                 count_in: in std_logic;
29                 count: out std_logic_vector(N-1 downto 0);
30                 y : out std_logic
31         );
32     end component;
33
34     signal temp_en_min : std_logic;
35     signal temp_en_h : std_logic;
36     signal temp_uscita : std_logic;
37
38     begin
39         counter_sec : cont_mod_n
40             generic map(6, 60)
41             port map(
42                 load => load_s,
43                 init => set_s,
44                 clock => clk,
45                 reset => rst,
46                 count_in => start,
47                 count => secondi,
48                 y => temp_en_min
49             );
50
51         counter_min : cont_mod_n
52             generic map(6, 60)
53             port map(
54                 load => load_min,

```

```

55         init => set_min,
56         clock => clk,
57         reset => rst,
58         count_in => temp_en_min,
59         count => minut,
60         y => temp_en_h
61     );
62
63     counter_h : cont_mod_n
64         generic map(5, 24)
65         port map(
66             load => load_h,
67             init => set_h,
68             clock => clk,
69             reset => rst,
70             count_in => temp_en_h,
71             count => ore,
72             y => temp_uscita
73     );
74
75 end structural;

```

Listing 2.11: Implementazione del cronometro.

Il timer vede come ingressi:

- `clk` : segnale di clock di sistema.
- `rst` : segnale di reset di sistema.
- `start` : segnale di avvio di conteggio.
- `load_s`, `load_min`, `load_h` : segnali di abilitazione di caricamento dei valori iniziali dei contatori di secondi, minuti e ore.
- `set_s`, `set_min` : vettori di 5 bit contenenti i valori di inizializzazione di secondi e minuti.
- `set_h` : vettore di 4 bit contenente il valori di inizializzazione delle ore.

e come uscite:

- `secondi`, `minuti` : conteggio di minuti e secondi.
- `ore` : conteggio delle ore.

All'interno dell'architettura strutturale del cronometro sono stati istanziati tre contatori, quelli dei secondi e dei minuti, di modulo 6 e valore massimo 60, e quello delle ore di modulo 5 e valore massimo 24. Sono stati utilizzati solo tre segnali temporanei:

- `temp_en_min` : per collegare l'uscita del contatore dei secondi all'abilitazione del contatore dei minuti.
- `temp_en_h` : per collegare l'uscita del contatore dei minuti all'abilitazione del contatore dei secondi.
- `temp_uscita` : per verificare l'uscita del contatore delle ore.

L'abilitazione del contatore dei secondi è collegata al segnale di segnale del processo.

2.4.3 Simulazione

Vediamo il testbench per il cronometro:

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3
4 ENTITY timer_tb IS
5 END timer_tb;
6
7 ARCHITECTURE behavioural OF timer_tb IS
8
9     COMPONENT timer
10    PORT (
11        clk, rst : IN std_logic;
12        start : IN std_logic;
13        load_s, load_min, load_h : IN std_logic;
14        set_s, set_min : IN std_logic_vector(5 DOWNTO 0);
15        set_h : IN std_logic_vector(4 DOWNTO 0);
16        secondi, minuti : OUT std_logic_vector(5 DOWNTO 0);
17        ore : OUT std_logic_vector(4 DOWNTO 0)
18    );
19    END COMPONENT;
20
21    signal clk, rst : std_logic := '0';
22    signal start : std_logic := '0';

```

```

23     signal load_s, load_min, load_h : std_logic := '0';
24     signal set_s, set_min : std_logic_vector(5 DOWNTO 0) :=
25         (others => '0');
26     signal set_h : std_logic_vector(4 DOWNTO 0) := (others =>
27         '0');
28
29
30     constant clk_period : time := 10 ns;
31
32 BEGIN
33
34     uut: timer PORT MAP (
35         clk => clk,
36         rst => rst,
37         start => start,
38         load_s => load_s,
39         load_min => load_min,
40         load_h => load_h,
41         set_s => set_s,
42         set_min => set_min,
43         set_h => set_h,
44         secondi => secondi,
45         minutii => minutii,
46         ore => ore
47     );
48
49     clk_process :process
50     begin
51         clk <= '0';
52         wait for clk_period/2;
53         clk <= '1';
54         wait for clk_period/2;
55     end process;
56
57     stim_proc: process
58     begin
59         rst <= '1';
60         wait for clk_period*10;
61         rst <= '0';

```

```

62      wait for clk_period*10;
63
64      load_s <= '1';
65      load_min <= '1';
66      load_h <= '1';
67      set_s <= "001010"; -- 10 seconds
68      set_min <= "000011"; -- 3 minutes
69      set_h <= "00010"; -- 2 hours
70      wait for clk_period*2;
71      load_s <= '0';
72      load_min <= '0';
73      load_h <= '0';
74      start <= '1';
75
76      wait for 200 ns;
77      wait;
78  end process;
79
80 END behavioural;

```

Listing 2.12: Simulazione del cronometro.

L'unità sotto test (*uut*) è ovviamente il timer, ai quali ingressi e uscite sono stati collegati dei segnali omonimi per effettuare i test. All'interno della simulazione sono presenti due processi: `clk_process`, che fa partire un impulso di clock ogni 10 nanosecondi, e `stim_process`, il quale inizialmente abilita il reset del sistema, dopodiché abilita i tre load dei contatori per inizializzarli con 2 ore, 3 minuti e 10 secondi, per poi abbassare dopo due periodi di clock i load e dare lo start. Simuliamo per 200 nanosecondi. Vediamo lo scandire del tempo nel seguente estratto di simulazione:

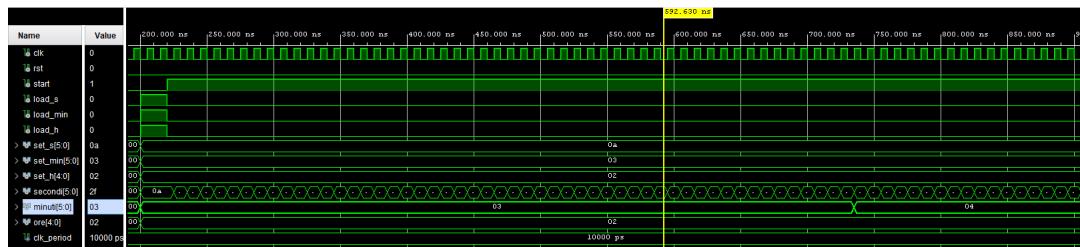


Figure 2.11: Simulazione del cronometro.

2.5 Esercizio 5.2 - Cronometro su board parte 1

Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando i display a 7 segmenti per la visualizzazione dell'orario (o una

combinazione di display e led nel caso in cui i display a disposizione siano in numero inferiore a quello necessario), gli switch per l'immissione dell'orario iniziale e due bottoni, uno per il set dell'orario e uno per il reset. Si utilizzi una codifica a scelta dello studente per la visualizzazione dell'orario sui display (esadecimale o decimale).

2.5.1 Progetto e architettura

L'esercizio richiede di utilizzare il display a 7 segmenti, la cui implementazione è già stata fornita. Il display richiede per ogni cifra mostrata 4 bit, e noi dobbiamo dare esattamente 6 cifre (se vogliamo rappresentare in base 10): unità e decine per i secondi, unità e decine per i minuti, e unità e decine per le ore, oltre che due trattini "-" (rappresentati con "1111") per separare i secondi dai minuti e i minuti dalle ore. Dunque, per avere due cifre per secondi, minuti ed ore, è necessario un convertitore, che prende in ingresso le uscite dei tre contatori. Sono stati utilizzati anche tre debouncer per effettuare il load dei secondi, dei minuti e delle ore per l'inizializzazione, oltre che un debouncer per effettuare il reset del sistema. Inoltre, l'ultimo componente aggiuntivo è il `clock_filter` già utilizzato nell'implementazione del display a 7 segmenti, utile a ridurre la frequenza del cronometri ad 1 Hz.

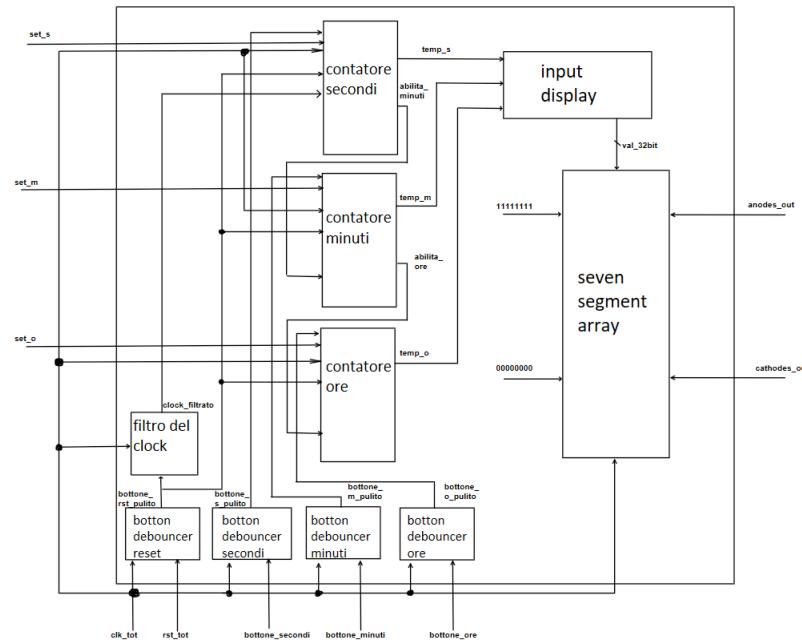


Figure 2.12: Schema Implementazione su board del cronometro.

2.5.2 Implementazione

Osserviamo l'implementazione del componente che rappresenta il convertitore:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.numeric_std.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6 entity convertitore is
7 Port (
8     secondi : in std_logic_vector(5 downto 0);
9     minuti : in std_logic_vector(5 downto 0);
10    ore : in std_logic_vector(3 downto 0); --12 ore.
11    uscita : out std_logic_vector(31 downto 0)
12 );
13 end convertitore;
14
15 architecture behavioural of convertitore is
16 signal secondi_unita : integer;
17 signal secondi_decine : integer;
18 signal minuti_unita : integer;
19 signal minuti_decine : integer;
20 signal ore_unita : integer;
21 signal ore_decine : integer;
22
23 signal secondi_tot : std_logic_vector(7 downto 0);
24 signal minuti_tot : std_logic_vector(7 downto 0);
25 signal ore_tot : std_logic_vector(7 downto 0);
26 signal uscita_temp : std_logic_vector(31 downto 0);
27
28 begin
29     -- su ogni schermino del display va una cifra, dunque
      otteniamo le cifre da dare al display.
30     secondi_unita <= to_integer(unsigned(secondi)) mod 10;
31     secondi_decine <= to_integer(unsigned(secondi)) / 10;
32     minuti_unita <= to_integer(unsigned(minuti)) mod 10;
33     minuti_decine <= to_integer(unsigned(minuti)) / 10;
34     ore_unita <= to_integer(unsigned(ore)) mod 10;
35     ore_decine <= to_integer(unsigned(ore)) / 10;
36
37     --ogni schermino del display prende 4 bit in ingresso,
      quindi trasformiamo ogni
38     --cifra da mettere sullo schermino in un vettore di 4 bit.

```

```

39      secondi_tot (3 downto 0) <=
        std_logic_vector(to_unsigned(secondi_unita, 4));
40      secondi_tot (7 downto 4) <=
        std_logic_vector(to_unsigned(secondi_decine, 4));
41
42      minutii_tot (3 downto 0) <=
        std_logic_vector(to_unsigned(minutii_unita, 4));
43      minutii_tot (7 downto 4) <=
        std_logic_vector(to_unsigned(minutii_decine, 4));
44
45      ore_tot (3 downto 0) <=
        std_logic_vector(to_unsigned(ore_unita, 4));
46      ore_tot (7 downto 4) <=
        std_logic_vector(to_unsigned(ore_decine, 4));
47
48      uscita_temp(7 downto 0) <= secondi_tot;
49      uscita_temp(11 downto 8) <= "1111";
50      uscita_temp(19 downto 12) <= minutii_tot;
51      uscita_temp(23 downto 20) <= "1111";
52      uscita_temp(31 downto 24) <= ore_tot;
53
54      uscita <= uscita_temp;
55
56 end behavioural;

```

Listing 2.13: Implementazione del convertitore.

Prevede i seguenti ingressi:

- *secondi, minutii, ore* : sono le uscite dei tre contatori che dovranno essere convertite in sei cifre da 4 bit.

e l'unica uscita:

- *uscita* : vettore di 32 bit che andrà in ingresso al display.

Sono stati dichiarati sei segnali d'appoggio interi adibiti al salvataggio del valore intero delle unità e delle decine di secondi, minutii ed ore (modulo 10 per ottenere le unità e diviso 10 per ottenere le decine). Sono stati dichiarati altri tre segnali (8 bit), *secondi_tot*, *minutii_tot*, e *ore_tot*, volti alla memorizzazione del valore di unità e decine convertite su 4 bit. Infine sul segnale d'appoggio di 32 bit *temp_uscita* abbiamo la concatenazione di secondi, minutii, ore e trattini.

Passiamo all'implementazione del cronometro su board:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity cronometro_board is
6     Port (
7         rst_tot : in STD_LOGIC;
8         clk_tot : in STD_LOGIC;
9         anodes_out : out STD_LOGIC_VECTOR (7 downto 0);
10        cathodes_out : out STD_LOGIC_VECTOR (7 downto 0);
11        bottone_secondi : in STD_LOGIC;
12        bottone_minuti : in STD_LOGIC;
13        bottone_ore : in STD_LOGIC;
14        set_s : in std_logic_vector(5 downto 0);
15        set_m : in std_logic_vector(5 downto 0);
16        set_o : in std_logic_vector(3 downto 0)
17    );
18 end cronometro_board;
19
20 architecture structural of cronometro_board is
21
22     component cont_mod_n is
23         generic(
24             N : positive := 2;
25             max : positive := 60
26         );
27         port( bottone_load: in std_logic;
28                 init : in std_logic_vector(N-1 downto 0);
29                 clock, reset: in std_logic;
30                 count_in: in std_logic;
31                 count: out std_logic_vector(N-1 downto 0);
32                 y : out std_logic
33         );
34     end component;
35
36     component button_debouncer
37         generic (
38             CLK_period: integer := 10; -- Period of the
39                         board's clock in nanoseconds
40             btn_noise_time: integer := 10000000 -- Estimated
41                         button bounce duration in nanoseconds

```

```

40      );
41      port (
42          RST : in STD_LOGIC;
43          CLK : in STD_LOGIC;
44          BTN : in STD_LOGIC;
45          CLEARED_BTN : out STD_LOGIC
46      );
47  end component;
48
49  component display_seven_segments
50      generic(
51          CLKIN_freq : integer := 100000000;
52          CLKOUT_freq : integer := 500
53      );
54      port ( CLK : in STD_LOGIC;
55          RST : in STD_LOGIC;
56          VALUE : in STD_LOGIC_VECTOR (31 downto 0);
57          ENABLE : in STD_LOGIC_VECTOR (7 downto 0); --
58              decide quali cifre abilitare
59          DOTS : in STD_LOGIC_VECTOR (7 downto 0); --
60              decide quali punti visualizzare
61          ANODES : out STD_LOGIC_VECTOR (7 downto 0);
62          CATHODES : out STD_LOGIC_VECTOR (7 downto 0)
63      );
64  end component;
65
66  component clock_filter
67      generic(
68          CLKIN_freq : integer := 100000000;
69          CLKOUT_freq : integer := 1
70      );
71      port (
72          clock_in : IN std_logic;
73          reset : in STD_LOGIC;
74          clock_out : OUT std_logic
75      );
76  end component;
77
78  component convertitore is
79      Port (
80          secondi : in std_logic_vector(5 downto 0);

```

```

79         minut : in std_logic_vector(5 downto 0);
80         ore : in std_logic_vector(3 downto 0);
81         uscita : out std_logic_vector(31 downto 0)
82     );
83 end component;
84
85     signal abilita_minuti : std_logic;
86     signal abilita_ore : std_logic;
87     signal temp : std_logic;
88
89     signal bottone_s_pulito : std_logic;
90     signal bottone_m_pulito : std_logic;
91     signal bottone_o_pulito : std_logic;
92     signal bottone_RST_pulito : std_logic;
93
94     signal val_32bit : std_logic_vector(31 downto 0);
95
96
97     signal temp_s : std_logic_vector(5 downto 0) := (others =>
98             '0');
99     signal temp_m : std_logic_vector(5 downto 0) := (others =>
100            '0');
101    signal temp_o : std_logic_vector(3 downto 0) := (others =>
102            '0');
103
104    begin
105        clk_filter: clock_filter
106        generic map(
107            CLKIN_freq => 100000000,
108            CLKOUT_freq => 1
109        )
110        port map(
111            clock_in => clk_tot,
112            reset => bottone_RST_pulito,
113            clock_out => clock_filtrato
114        );
115
116        cont_secondi: cont_mod_n

```

```

117     generic map (
118         N => 6,
119         max => 60
120     )
121     port map (
122         bottone_load => bottone_s_pulito,
123         init => set_s,
124         clock => clk_tot,
125         reset => bottone_RST_pulito,
126         count_in => clock_filtrato,
127         count => temp_s,
128         y => abilita_minuti
129     );
130
131     cont_minuti: cont_mod_n
132         generic map (
133             N => 6,
134             max => 60
135         )
136         port map (
137             bottone_load => bottone_m_pulito,
138             init => set_m,
139             clock => clk_tot,
140             reset => bottone_RST_pulito,
141             count_in => abilita_minuti,
142             count => temp_m,
143             y => abilita_ore
144         );
145
146     cont_ore: cont_mod_n
147         generic map (
148             N => 4,
149             max => 12
150         )
151         port map (
152             bottone_load => bottone_o_pulito,
153             init => set_o,
154             clock => clk_tot,
155             reset => bottone_RST_pulito,
156             count_in => abilita_ore,
157             count => temp_o,

```

```

158         y => temp
159     );
160
161     de_B_sec: button_debouncer
162         Generic map (
163             CLK_period => 10, -- Period of the board's clock
164             in 10ns
165             btn_noise_time => 10000000 -- Estimated button
166             bounce duration of 10ms
167         )
168         Port map (
169             RST => '0',
170             CLK => clk_tot,
171             BTN => bottone_secondi,
172             CLEARED_BTN => bottone_s_pulito
173         );
174
175     de_B_min: button_debouncer
176         Generic map (
177             CLK_period => 10, -- Period of the board's clock
178             in 10ns
179             btn_noise_time => 10000000 -- Estimated button
180             bounce duration of 10ms
181         )
182         Port map (
183             RST => '0',
184             CLK => clk_tot,
185             BTN => bottone_minuti,
186             CLEARED_BTN => bottone_m_pulito
187         );
188
189     de_B_ore: button_debouncer
190         Generic map (
191             CLK_period => 10, -- Period of the board's clock
192             in 10ns
193             btn_noise_time => 10000000 -- Estimated button
194             bounce duration of 10ms
195         )
196         Port map (
197             RST => '0',
198             CLK => clk_tot,

```

```

193         BTN => bottone_ore,
194         CLEARED_BTN => bottone_o_pulito
195     );
196
197     de_B_RST: button_debouncer
198     Generic map (
199         CLK_period => 10, -- Period of the board's clock
200             in 10ns
201         btn_noise_time => 10000000 -- Estimated button
202             bounce duration of 10ms
203     )
204     Port map (
205         RST => '0',
206         CLK => clk_tot,
207         BTN => rst_tot,
208         CLEARED_BTN => bottone_RST_pulito
209     );
210
211     input_display : convertitore
212     Port map (
213         secondi => temp_s,
214         minuti => temp_m,
215         ore => temp_o,
216         uscita => val_32bit
217     );
218
219     seven_segment_array: display_seven_segments
220     GENERIC MAP (
221         CLKIN_freq => 100000000, --qui inserisco i parametri
222             effettivi (clock della board e clock in uscita
223             desiderato)
224         CLKOUT_freq => 500 --inserendo un valore inferiore si
225             vedranno le cifre illuminarsi in sequenza
226     )
227     PORT MAP (
228         CLK => clk_tot,
229         RST => '0',
230         value => val_32bit,
231         enable => "11111111", --stabilisco che tutti i display
232             siano accesi
233         dots => "00000000", --stabilisco che tutti i punti siano

```

```

    spenti
228     anodes => anodes_out,
229     cathodes => cathodes_out
230   );
231
232 end structural;

```

Listing 2.14: Implementazione del cronometro su board.

Gli ingressi sono i seguenti:

- `rst_tot` : reset del cronometro.
- `clk_tot` : clock del sistema.
- `bottone_secondi`, `bottone_minuti`, `bottone_ore` : bottoni che consentono il load di secondi, minuti ed ore.
- `set_s`, `set_m`, `set_o` : valori di inizializzazione di secondi, minuti e ore.

invece le uscite:

- `anodes_out`, `cathodes_out` : anodi e catodi dei segmenti.

Per il collegamento di tutti i componenti sono stati sfruttati i seguenti segnali d'appoggio:

- `abilita_minuti` : come già visto precedentemente collega l'uscita del contatore dei secondi all'abilitazione del contatore dei minuti.
- `abilita_ore` : collega l'uscita del contatore dei minuti all'abilitazione del contatore delle ore.
- `bottone_s_pulito` : il debouncer dei secondi prende in ingresso `bottone_secondi`, lo ripulisce e lo restituisce al contatore dei secondi.
- `bottone_m_pulito` : il debouncer dei minuti prende in ingresso `bottone_minuti`, lo ripulisce e lo restituisce al contatore dei minuti.
- `bottone_o_pulito` : il debouncer delle ore prende in ingresso `bottone_ore`, lo ripulisce e lo restituisce al contatore delle ore.
- `bottone_RST_pulito` : il debouncer del reset prende in ingresso `rst_tot` e lo restituisce ai contatori e il filtro del clock.
- `val_32bit` : è il valore in uscita dal convertitore che andrà in ingresso al display.

- `temp_s`, `temp_m`, `temp_o` : valori in uscita dei contatori.

Abbiamo collegato al clock di sistema il clock della board:

```
## Clock signal
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports {clk_tot}]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk_tot}];
```

Figure 2.13: collegamenti.

I valori di inizializzazione di secondi, minuti ed ore agli switch:

```
##Switches
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { set_s[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { set_s[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { set_s[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { set_s[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { set_s[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { set_s[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { set_m[0] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { set_m[1] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMOS18 } [get_ports { set_m[2] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS18 } [get_ports { set_m[3] }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMOS33 } [get_ports { set_m[4] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMOS33 } [get_ports { set_m[5] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMOS33 } [get_ports { set_o[0] }]; #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMOS33 } [get_ports { set_o[1] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMOS33 } [get_ports { set_o[2] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { set_o[3] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]
```

Figure 2.14: collegamenti.

Anodi e catodi al segment display, e i bottoni di load e reset ai bottoni della board:

```
##7 segment display
set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[1] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[2] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[6] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15 IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { anodes_out[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { anodes_out[1] }]; #IO_L23N_T3_FW_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMOS33 } [get_ports { anodes_out[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { anodes_out[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMOS33 } [get_ports { anodes_out[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { anodes_out[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2 IOSTANDARD LVCMOS33 } [get_ports { anodes_out[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMOS33 } [get_ports { anodes_out[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons
#set_property -dict { PACKAGE_PIN C12 IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
set_property -dict { PACKAGE_PIN N17 IOSTANDARD LVCMOS33 } [get_ports { bottone_minuti }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18 IOSTANDARD LVCMOS33 } [get_ports { rst_tot }]; #IO_L4N_T0_D05_14 Sch=bttnu
set_property -dict { PACKAGE_PIN P17 IOSTANDARD LVCMOS33 } [get_ports { bottone_secondi }]; #IO_L12P_T1_MRCC_14 Sch=btndl
set_property -dict { PACKAGE_PIN M17 IOSTANDARD LVCMOS33 } [get_ports { bottone_ore }]; #IO_L10N_T1_D15_14 Sch=bttnr
#set_property -dict { PACKAGE_PIN P18 IOSTANDARD LVCMOS33 } [get_ports { BTND }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

Figure 2.15: collegamenti.

2.5.3 Timing Analysis

Per effettuare la timing analysis in Vivado è necessario prima di tutto effettuare la sintesi, che consentirebbe di eseguire già l'analisi ma con delle somme approssimate,

quindi abbiamo effettuato anche l'implementazione. Nella schermata "Report Timing Summary" è possibile configurare i parametri della timing analysis, che specificano il tipo di report da generare e il contenuto da mostrare una volta eseguita l'analisi. In particolare, abbiamo selezionato min e max, come "Path delay type", per misurare il Worst Negative Slack (WNS), cioè il tempo che impiega un segnale di input a stabilizzarsi prima del fronte successivo del clock, tale che le uscite raggiungano il valore desiderato e il Worst Hold Slack (WHS), cioè il tempo per cui un segnale di input deve restare stabile dopo il fronte del clock per consentire all'output di raggiungere il valore desiderato. Lo slack indica la differenza tra require time e arrival time. In questa analisi viene misurato anche il Worst Pulse Width Slack (WPWS) che indica il peggiore tra tutti i controlli considerando i ritardi sia minimi che massimi. Il parametro maximum number of path per clock indica quanti percorsi possono essere analizzati simultaneamente in un singolo ciclo di clock, mentre il maximum number of worst paths per endpoint consente di limitare il numero massimo di percorsi peggiori analizzati per ciascun endpoint.

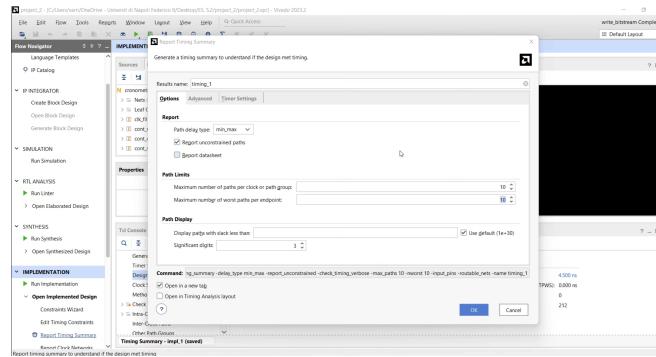


Figure 2.16: Report Timing Summary

È importante che il WNS sia positivo, perché altrimenti vuol dire che il percorso sarà fallito. Di seguito sono riportati i risultati per quanto riguarda il clock creato:

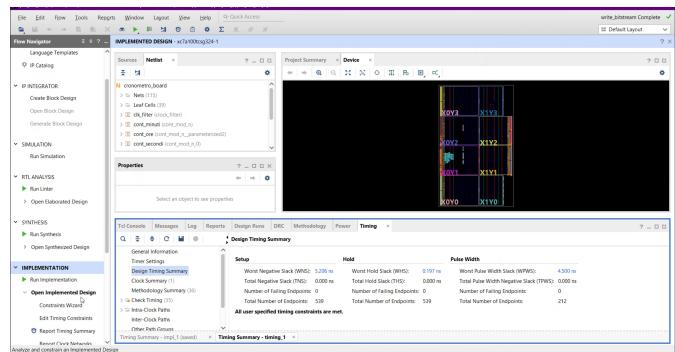


Figure 2.17: Risultati.

Una volta controllato questo risultato, abbiamo calcolato anche la FMAX, cioè

la frequenza massima di funzionamento. Essa non è fornita esplicitamente nei report ma l'abbiamo stimata con la seguente formula:

$$\frac{1}{T-WNS}$$

dove T è il periodo del clock target. Per trovare questo valore è possibile diminuire progressivamente il periodo del clock di design, finché non si ottiene un WNS negativo. In particolare, abbiamo diminuito il periodo fino a 3.5 ns con una forma d'onda con fronte di salita in 0 ns e fronte di discesa a 1.75 ns period 3.50 waveform 1.75 Il valore approssimato è 281 MHZ. Oltre questa frequenza i vincoli temporali non sono più rispettati.

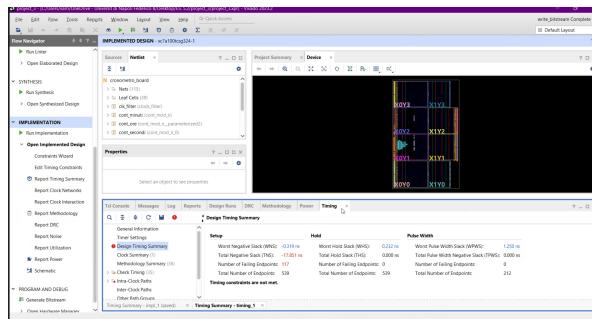


Figure 2.18: Risultati con WNS negativo.

2.6 Esercizio 5.3 Cronometro su board parte 2

Estendere il componente sviluppato ai punti precedenti in modo che sia in grado di acquisire e memorizzare internamente fino ad un numero N di intertempi in corrispondenza di un ingresso di stop. Opzionalmente, il componente può prevedere una modalità di visualizzazione in cui, alla pressione di un bottone, vengano visualizzati sui display gli intertempi memorizzati (uno per ogni pressione).

2.6.1 Progetto e architettura

Per implementare il salvataggio degli intertempi si è deciso di aggiungere una memoria ed un contatore, il contatore viene utilizzato per scandire gli indirizzi ogni volta che il bit di stop è alto. Il numero di possibili salvataggi viene definito all'interno del generic della memoria come numero_locazioni. Quello che succede è che, alla pressione del bottone di stop si effettua un salvataggio nella memoria il segnale `intertempo`, che non è altro che il valore di uscita del contatore delle ore, dei minuti e dei secondi concatenati in quest'ordine. Effettuato il salvataggio, questo valore viene messo in out e letto sui 16 led che vanno a rappresentare, i primi 4 le ore, gli altri 6 i minuti e gli ultimi 6 i secondi. Lo schema è il seguente:

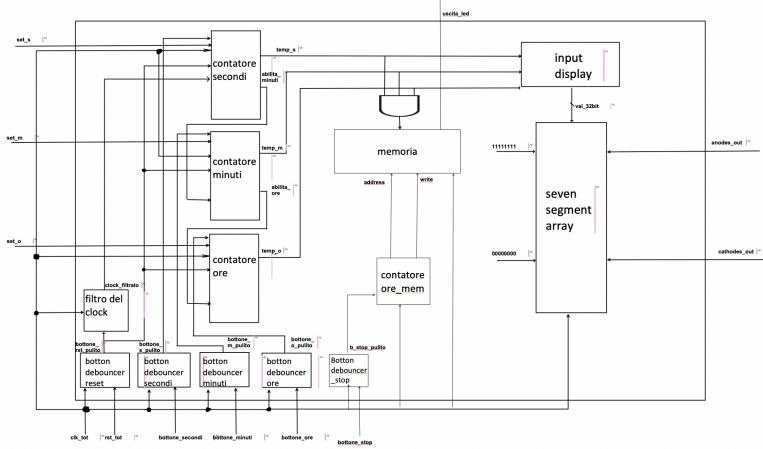


Figure 2.19: Cronometro board

2.6.2 Implementazione

L'implementazione di memoria e contatore è già stata illustrata precedentemente, per questo motivo, si è deciso di illustrare solo i cambiamenti rispetto al precedente esercizio che riguardano l'aggiunta di un bottone stop e un'uscita da collegare ai led della board:

```

1  entity cronometro_board is
2    Port (
3      rst_tot : in STD_LOGIC;
4      clk_tot : in STD_LOGIC;
5      anodes_out : out STD_LOGIC_VECTOR (7 downto 0);
6      cathodes_out : out STD_LOGIC_VECTOR (7 downto 0);
7      bottone_secondi : in STD_LOGIC;
8      bottone_minuti : in STD_LOGIC;
9      bottone_ore : in STD_LOGIC;
10     set_s : in std_logic_vector(5 downto 0);
11     set_m : in std_logic_vector(5 downto 0);
12     set_o : in std_logic_vector(3 downto 0);
13     bottone_stop : in std_logic;
14     uscita_led : out std_logic_vector(15 downto 0)
15   );
16 end cronometro_board;
```

Listing 2.15: Modifiche dell'entity del cronometro su board.

Per effettuare i collegamenti alla board di bottone_stop e uscita_led è stato necessario modificare il file **Nexys-A7-100T-Master.xdc** cambiando, rispetto all'esercizio precedente solo questi valori:

```

1 ##Buttons
2 #set_property -dict { PACKAGE_PIN C12    IOSTANDARD LVCMOS33 }
   [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15
   Sch=cpu_resetn
3 set_property -dict { PACKAGE_PIN N17    IOSTANDARD LVCMOS33 }
   [get_ports { bottone_minuti }]; #IO_L9P_T1_DQS_14 Sch=btnc
4 set_property -dict { PACKAGE_PIN M18    IOSTANDARD LVCMOS33 }
   [get_ports { rst_tot }]; #IO_L4N_T0_D05_14 Sch=btlu
5 set_property -dict { PACKAGE_PIN P17    IOSTANDARD LVCMOS33 }
   [get_ports { bottone_secondi }]; #IO_L12P_T1_MRCC_14
   Sch=btln
6 set_property -dict { PACKAGE_PIN M17    IOSTANDARD LVCMOS33 }
   [get_ports { bottone_ore }]; #IO_L10N_T1_D15_14 Sch=btnr
7 set_property -dict { PACKAGE_PIN P18    IOSTANDARD LVCMOS33 }
   [get_ports { bottone_stop }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd

```

Listing 2.16: Aggiunta del bottone di stop.

```

1 ## LEDs
2 set_property -dict { PACKAGE_PIN H17    IOSTANDARD LVCMOS33 }
   [get_ports { uscita_led[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
3 set_property -dict { PACKAGE_PIN K15    IOSTANDARD LVCMOS33 }
   [get_ports { uscita_led[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
4 set_property -dict { PACKAGE_PIN J13    IOSTANDARD LVCMOS33 }
   [get_ports { uscita_led[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
5 set_property -dict { PACKAGE_PIN N14    IOSTANDARD LVCMOS33 }
   [get_ports { uscita_led[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
6 set_property -dict { PACKAGE_PIN R18    IOSTANDARD LVCMOS33 }
   [get_ports { uscita_led[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
7 set_property -dict { PACKAGE_PIN V17    IOSTANDARD LVCMOS33 }
   [get_ports { uscita_led[5] }]; #IO_L18N_T2_A11_D27_14
   Sch=led[5]
8 set_property -dict { PACKAGE_PIN U17    IOSTANDARD LVCMOS33 }
   [get_ports { uscita_led[6] }]; #IO_L17P_T2_A14_D30_14
   Sch=led[6]
9 set_property -dict { PACKAGE_PIN U16    IOSTANDARD LVCMOS33 }
   [get_ports { uscita_led[7] }]; #IO_L18P_T2_A12_D28_14
   Sch=led[7]
10 set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 }
   [get_ports { uscita_led[8] }]; #IO_L16N_T2_A15_D31_14

```

```

Sch=led[8]
11 set_property -dict { PACKAGE_PIN T15      IO_STANDARD LVCMOS33 }
      [get_ports { uscita_led[9] }]; #IO_L14N_T2_SRCC_14
      Sch=led[9]
12 set_property -dict { PACKAGE_PIN U14      IO_STANDARD LVCMOS33 }
      [get_ports { uscita_led[10] }]; #IO_L22P_T3_A05_D21_14
      Sch=led[10]
13 set_property -dict { PACKAGE_PIN T16      IO_STANDARD LVCMOS33 }
      [get_ports { uscita_led[11] }];
      #IO_L15N_T2_DQS_DOUT_CS0_B_14 Sch=led[11]
14 set_property -dict { PACKAGE_PIN V15      IO_STANDARD LVCMOS33 }
      [get_ports { uscita_led[12] }]; #IO_L16P_T2_CSI_B_14
      Sch=led[12]
15 set_property -dict { PACKAGE_PIN V14      IO_STANDARD LVCMOS33 }
      [get_ports { uscita_led[13] }]; #IO_L22N_T3_A04_D20_14
      Sch=led[13]
16 set_property -dict { PACKAGE_PIN V12      IO_STANDARD LVCMOS33 }
      [get_ports { uscita_led[14] }]; #IO_L20N_T3_A07_D23_14
      Sch=led[14]
17 set_property -dict { PACKAGE_PIN V11      IO_STANDARD LVCMOS33 }
      [get_ports { uscita_led[15] }]; #IO_L21N_T3_DQS_A06_D22_14
      Sch=led[15]

```

Listing 2.17: Abilitazione dei LED.

2.7 Esercizio 6.1 - Sistema di lettura-elaborazione-scrittura PO PC

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.

2.7.1 Progetto e architettura

Per l'implementazione del sistema di lettura-elaborazione-scrittura abbiamo utilizzato un **approccio strutturale**, dato che il sistema può essere decomposto in sistemi più piccoli e semplici che collegati tra loro consentono di generare il comportamento desiderato. Lo schema rappresentativo del sistema è il seguente:

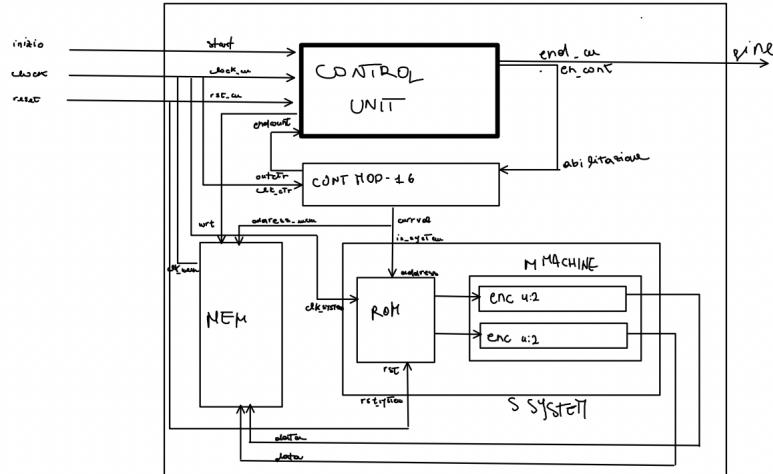


Figure 2.20: Schema del sistema PO-PC

Inoltre, come è possibile notare dallo schema sovrastante, è stata implementata una **Control Unit**, quest'ultima implementa il seguente **automa a stati finiti** per consentire il funzionamento del sistema.

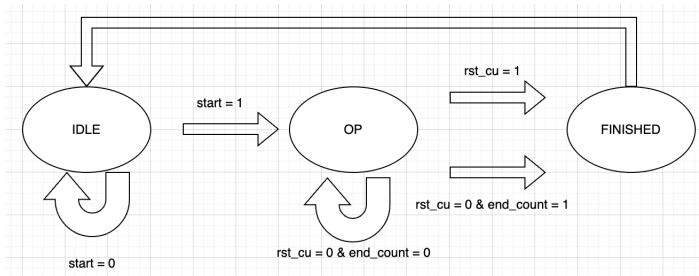


Figure 2.21: ASF - Control Unit - POPC

Possiamo individuare 3 stati in totale: **IDLE** è uno stato nel quale si permane fintanto che il segnale di **start** risulterà essere pari a 0, in caso contrario, ci si porterà dallo stato di **idle** allo stato di **OP**, nel quale verranno effettuate le operazioni vere e proprie; come vedremo nel paragrafo successivo la Control Unit è caratterizzata da diversi segnali di input e output, è proprio in questo stato che si controlla il valore del segnale di **reset** della Control Unit, fintanto che esso assume valore basso, si permane nello stato corrente e vengono settati i valori dei restanti segnali nel seguente modo:

- en_cu = 0: è il segnale di abilitazione della Control Unit.
- rd_cu=1: è il segnale che abilita l'operazione di lettura.
- wrt_cu=1: è il segnale che abilita l'operazione di scrittura
- end_count=0: è il segnale che viene inoltrato al contatore e che gli notifica di terminare il conteggio.

Nel momento in cui il segnale di reset diventa alto, i segnali visti precedentemente assumono i seguenti valori:

- rst_cu = 0
- en_cu = 1
- rd_cu=1
- wrt_cu=1
- end_count=1

Inoltre, si transiterà nello stato **FINISHED**. È possibile notare che sono presenti due transizioni che consentono il passaggio allo stato appena citato, la seconda, è quella nella quale i segnali assumono i seguenti valori:

- rst_cu = 1
- en_cu = 0
- rd_cu=1
- wrt_cu=1
- end_count=1

Infine, nel momento in cui i segnali assumono i seguenti valori:

- en_cu = 0
- rd_cu=0
- wrt_cu=0
- end_cu=1

il segnale di fine viene alzato sancendo il ritorno allo stato di **IDLE**

2.7.2 Implementazione

ROM

```
1
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4 use work.all;
5
6 entity rom is
7     port (
8         address : in std_logic_vector(3 downto 0);
9         clk : in std_logic;
10        rd : in std_logic;
11        out_value : out std_logic_vector(7 downto 0)
12    );
13 end entity rom;
```

Listing 2.18: Implementazione memoria ROM

In questa prima parte di codice viene dichiarata una entity chiamata **rom** caratterizzata da 4 segnali: **address** (input a 4 bit) che rappresenta l'indirizzo di memoria passato, **clk**, segnale di input che determina più esattamente il segnale di temporizzazione che stiamo prendendo in considerazione, **rd** (input di controllo di lettura), e **out_value** (output a 8 bit) segnale che rappresenta l'uscita della memoria.

Successivamente abbiamo dichiarato l'architecture di tipo **dataflow**:

```
1
2 architecture dataflow of rom is
3     type MEMORY_16_8 is array (0 to 15) of std_logic_vector(7
4         downto 0);
5     constant ROM_16_8 : MEMORY_16_8 := (
6         "00010001",
7         "00100010",
8         "01000100",
9         "10001000",
10        "00010010",
11        "00010100",
12        "00011000",
13        "00100001",
14        "00100100",
15        "00101000",
16        "01000001",
17        "01000010",
```

```

17      "01001000",
18      "10000001",
19      "10000100",
20      "10000010"
21  );

```

Listing 2.19: Architecture memoria ROM

Questo dichiara un tipo **MEMORY_16_8**, che è un array di 16 vettori a 8 bit. Viene inoltre dichiarata una costante **ROM_16_8** che inizializza la memoria con i valori specificati superiormente e che nello specifico rappresentano valori da 8 bit. Infine, all'interno dell'architecture è stato previsto il seguente process:

```

1
2  begin
3      process(clk)
4          begin
5              if(rising_edge(clk)) then
6                  if(rd = '1') then
7                      out_value <=
8                          ROM_16_8(to_integer(unsigned(address)));
9                  end if;
10             end if;
11         end process;
12
13 end architecture dataflow;

```

Listing 2.20: Architecture memoria ROM

Quest'ultimo è sensibile al fronte di salita del clock (clk). Se rd è attivato, l'output out_value assume il valore memorizzato nella ROM all'indirizzo specificato da address.

Questo codice descrive essenzialmente una ROM sincrona che emette un valore in base all'indirizzo di input quando il segnale di controllo di lettura (rd) è attivato sul fronte di salita del clock.

M Machine

Il codice VHDL che descrive una macchina sequenziale (m_machine) che utilizza un componente chiamato **encoder42** per effettuare l'encoding di due gruppi di segnali di input. In particolare, l'input **a_machine** è diviso in due gruppi di 4 bit ciascuno e sottoposto ad encoding separato attraverso due istanze del componente encoder42. L'output dell'encoding è quindi concatenato per formare **y_machine**, un segnale di output a 4 bit. Ecco una spiegazione più dettagliata del codice:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.all;
4
5 entity m_machine is
6     port( a_machine : in STD_LOGIC_VECTOR(7 downto 0);
7           y_machine : out STD_LOGIC_VECTOR(3 downto 0)
8       );
9 end m_machine;

```

Listing 2.21: Implementazione della macchina M

L'entity m_machine ha un input a_machine di 8 bit e un output y_machine di 4 bit.

```

1 architecture structural of m_machine is
2
3     component encoder42 is
4
5         port(   a : in STD_LOGIC_VECTOR (0 to 3);
6               y : out STD_LOGIC_VECTOR (0 to 1)
7           );
8
9     end component;
10
11 begin
12     encoder0 : encoder42
13         port map(
14             a(0) => a_machine(0),
15             a(1) => a_machine(1),
16             a(2) => a_machine(2),
17             a(3) => a_machine(3),
18             y(0) => y_machine(0),
19             y(1) => y_machine(1)
20         );
21
22     encoder1 : encoder42
23         port map(
24             a(0) => a_machine(4),
25             a(1) => a_machine(5),
26             a(2) => a_machine(6),
27             a(3) => a_machine(7),
28             y(0) => y_machine(2),
29             y(1) => y_machine(3)

```

```

30      );
31  end structural;

```

Listing 2.22: Architecture della macchina M

L'architecture structural descrive la struttura interna della macchina. Viene dichiarato un componente encoder42 con un input a di 4 bit e un output y di 2 bit. Vengono istanziate due copie del componente encoder42 chiamate encoder1 e encoder2. Ciascuna istanza riceve un gruppo di 4 bit da a_machine e produce un output a 2 bit chiamato y, che viene poi concatenato per formare l'output finale y_machine della macchina.

Encoder

Possiamo descrivere più dettagliatamente l'encoder che è stato utilizzato per la realizzazione della macchina M, vista precedentemente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4
5 entity encoder42 is
6
7   port(  a : in STD_LOGIC_VECTOR (3 downto 0);
8         y : out STD_LOGIC_VECTOR (1 downto 0)
9   );
10
11 end encoder42;
12
13
14
15 architecture dataflow of encoder42 is
16
17 begin
18   y      <=  "00" when a="0001" else
19                  "01" when a="0010" else
20                  "10" when a="0100" else
21                  "11" when a="1000" else
22                  "--";
23
24 end dataflow;

```

Listing 2.23: Implementazione dell'Encoder

Il codice VHDL definisce un componente chiamato **encoder42**. Esso prende in input un vettore di segnali **a** di 4 bit e produce un vettore di output **y** di 2 bit, effettuando

un'operazione di encoding. L'architecture dataflow descrive il comportamento del componente. L'assegnazione degli output y è basata sui valori dell'input a. Se a è uguale a "0001", y è impostato su "00". Se a è uguale a "0010", y è impostato su "01". Se a è uguale a "0100", y è impostato su "10". Se a è uguale a "1000", y è impostato su "11". In tutti gli altri casi, y è impostato su "-".

Counter

Il codice VHDL definisce un contatore modulo 16 (counter mod 16) che conta da 0 a 15 e si riavvolge nuovamente a 0 quando raggiunge il massimo valore. Ecco una spiegazione più dettagliata del codice:

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use IEEE.numeric_std.ALL;
4
5 entity counter_mod_16 is
6     port (
7         clk_ctr, rst_ctr, en_ctr : in std_logic;
8         curr_value : out std_logic_vector(3 downto 0);
9         out_ctr : out std_logic
10    );
11 end counter_mod_16;
12
13 architecture behavioral of counter_mod_16 is
14     signal current_value : std_logic_vector(3 downto 0) := 
15         "0000";
16     signal exit_ctr : std_logic := '0';
17     signal temp_value : unsigned(3 downto 0);
18
19 begin
20     mod_16: process(clk_ctr, rst_ctr)
21     begin
22         if rst_ctr = '1' then
23             current_value <= "0000";
24         elsif rising_edge(clk_ctr) then
25             if en_ctr = '1' then
26                 temp_value <= unsigned(current_value) + 1;
27                 current_value <= std_logic_vector(temp_value);
28                 if current_value = "1111" then
29                     current_value <= "0000";
30                     exit_ctr <= '1';
31
32 end process;
33 end behavioral;

```

```

30           else
31                 exit_ctrl <= '0';
32             end if;
33           end if;
34         end if;
35       end process mod_16;
36
37     curr_value <= current_value;
38     out_ctrl <= exit_ctrl;
39
40 end behavioral;

```

Listing 2.24: Implementazione del Counter

L'entity counter_mod_16 ha tre segnali di input (clk_ctr, rst_ctr, e en_ctr), due segnali di output (curr_value e out_ctrl), e rappresenta un contatore modulo 16. L'architecture behavioral descrive il comportamento del contatore. Vengono dichiarati due segnali interni: current_value rappresenta il valore corrente del contatore (a 4 bit) e exit_ctrl indica se il contatore ha raggiunto il valore massimo (15). Il processo mod_16 controlla il contatore. Se il segnale di reset (rst_ctrl) è alto, il valore corrente del contatore viene reimpostato a "0000". Altrimenti, al fronte di salita del clock (rising_edge(clk_ctr)), se il segnale di abilitazione (en_ctrl) è alto, il contatore incrementa di 1. Se il contatore raggiunge il valore massimo ("1111"), viene riportato a "0000" e exit_ctrl viene impostato a '1' per indicare che il contatore ha raggiunto il massimo. In caso contrario, exit_ctrl è impostato a '0'. Gli output curr_value e out_ctrl vengono assegnati rispettivamente ai valori di current_value e exit_ctrl.

MEM

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.numeric_std.ALL;
4 use work.all;
5
6 entity mem is
7   port (
8     address_mem : in std_logic_vector(3 downto 0);
9     data        : in std_logic_vector(3 downto 0);
10    clk_mem     : in std_logic;
11    wrt         : in std_logic;
12    out_value_mem : out std_logic_vector(3 downto 0)
13  );
14 end entity mem;

```

```

15
16 architecture dataflow of mem is
17     type MEMORY_16_4 is array (0 to 15) of std_logic_vector(3
18         downto 0);
19     signal MEM_16_4 : MEMORY_16_4 := (others => "0000");
20
21 begin
22     process(clk_mem)
23         begin
24             if rising_edge(clk_mem) then
25                 if wrt = '1' then
26                     MEM_16_4(to_integer(unsigned(address_mem))) <=
27                         data;
28                     out_value_mem <= data;
29                 end if;
30             end if;
31         end process;
32
33 end architecture dataflow;

```

Listing 2.25: Implementazione della Memoria

Il codice VHDL sovrastante implementa una memoria sincrona a 16 locazioni, ciascuna in grado di contenere un valore a 4 bit. In primo luogo, l'entità mem ha cinque porti:

- address_mem: Un ingresso a 4 bit utilizzato per selezionare una delle 16 locazioni di memoria.
- data: Un ingresso a 4 bit che rappresenta il dato da scrivere in memoria.
- clk_mem: Un segnale di clock utilizzato per sincronizzare le operazioni di memoria.
- wrt: Un segnale di controllo che, quando alto ('1'), abilita la scrittura dei dati in memoria.
- out_value_mem: Un'uscita a 4 bit che restituisce il dato scritto nell'ultima operazione di scrittura.

Per quanto riguarda l'architettura, viene definito un tipo di dato MEMORY_16_4, che è un array di 16 elementi, ciascuno dei quali è un std_logic_vector di 4 bit. Questo tipo rappresenta la struttura della memoria. Il segnale MEM_16_4 è un'istanza del tipo di dato MEMORY_16_4 e viene inizializzato con tutti zeri, esso rappresenta

la memoria fisica dove i dati vengono memorizzati. Il comportamento di questo componente è molto semplice: il processo inizia con l'ascolto del segnale di clock, nel momento in cui viene rilevato un fronte di salita del clock, il sistema controlla il segnale wrt. Se quest'ultimo è alto, il dato presente sul segnale data viene scritto nella locazione di memoria specificata dall'address_mem. La conversione da std_logic_vector a integer è necessaria per utilizzare l'indirizzo come indice nell'array di memoria. Il dato scritto viene assegnato immediatamente all'uscita out_value_mem, indipendentemente dal fatto che il dato sia nuovo o meno nella locazione di memoria specificata.

Control Unit

Il codice, definisce un'unità di controllo (control_unit) per gestire il controllo di un sistema sequenziale a stati finiti. La macchina a stati finiti implementata ha tre stati: **IDLE**, **OP** e **FINISHED**. La macchina transita attraverso questi stati in base ai segnali di input.

Ecco il codice:

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use IEEE.numeric_std.ALL;
4
5 entity control_unit is
6   port (
7     start      : in std_logic;
8     clk_cu    : in std_logic;
9     end_count  : in std_logic;
10    rst_cu    : in std_logic;
11    rd_cu     : out std_logic;
12    wrt_cu   : out std_logic;
13    en_cu     : out std_logic;
14    end_cu   : out std_logic
15  );
16 end control_unit;
17
18 architecture behavioural of control_unit is
19 type state_cu is (IDLE, OP, FINISHED);
20 signal current_state : state_cu := IDLE;
21 signal next_state   : state_cu := IDLE;
22
23 begin
24   process(clk_cu)
25     begin

```

```

26      if rising_edge(clk_cu) then
27          current_state <= next_state;
28
29      case current_state is
30          when IDLE =>
31              if start = '1' then
32                  next_state <= OP;
33              else
34                  next_state <= IDLE;
35          end if;
36
37          when OP =>
38              if rst_cu = '1' then
39                  next_state <= FINISHED;
40              else
41                  en_cu    <= '1';
42                  rd_cu    <= '1';
43                  wrt_cu   <= '1';
44
45                  if end_count = '1' then
46                      next_state <= FINISHED;
47                  else
48                      next_state <= OP;
49          end if;
50
51      when FINISHED =>
52          en_cu    <= '0';
53          rd_cu    <= '0';
54          wrt_cu   <= '0';
55          end_cu   <= '1';
56          next_state <= IDLE;
57
58      end case;
59  end if;
60
61 end process;
62
63 end behavioural;

```

Listing 2.26: Implementazione della Control Unit

In primo luogo viene dichiarato il componente mediante una entity **control_unit** caratterizzata da diversi segnali di input (start, clk_cu, end_count, rst_cu) e diversi segnali di output (rd_cu, wrt_cu, en_cu, end_cu). L’architecture behavioural descrive

il comportamento dell'unità di controllo. È stato dichiarato un tipo enumerativo **state_cu** per rappresentare gli stati della macchina a stati finiti. Vengono anche dichiarati i segnali `current_state` e `next_state` per mantenere lo stato corrente e lo stato successivo della macchina. Infine, il processo principale è sensibile al fronte di salita del clock (`rising_edge(clk_cu)`) e gestisce la logica di transizione degli stati. In base allo stato corrente, vengono eseguite azioni specifiche. Ad esempio: Quando nello stato IDLE e il segnale `start` è alto, la macchina passa allo stato OP. Nello stato OP, se il segnale `rst_cu` è alto, la macchina passa allo stato FINISHED. Altrimenti, vengono attivati i segnali `en_cu`, `rd_cu`, e `wrt_cu`. La macchina rimane nello stato OP finché il segnale `end_count` non diventa alto, dopodiché passa allo stato FINISHED. Nello stato FINISHED, vengono disattivati i segnali `en_cu`, `rd_cu`, e `wrt_cu`, viene attivato il segnale `end_cu`, e la macchina torna allo stato IDLE.

S_System

Il codice VHDL definisce un componente denominato `s_system`, che rappresenta un sistema sequenziale composto da una ROM (`rom`) e una macchina sequenziale. Il sistema legge un indirizzo dalla ROM e fornisce l'output della macchina sequenziale.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.all;
4
5 entity s_system is
6     port(    in_system : in STD_LOGIC_VECTOR(3 downto 0);
7                 clk_system : in std_logic;
8                 rd_system : in std_logic;
9                 out_system : out STD_LOGIC_VECTOR(3 downto 0)
10            );
11 end s_system;
12
13 architecture structural of s_system is
14     signal u : STD_LOGIC_VECTOR(7 downto 0);
15
16     component rom is
17         port(
18             address : in std_logic_vector(3 downto 0);
19             clk : in std_logic;
20             rd : in std_logic;
21             out_value : out std_logic_vector(7 downto 0)
22         );

```

```

23      end component;
24
25      component m_machine is
26          port( a_machine : in STD_LOGIC_VECTOR(7 downto 0);
27                  y_machine : out STD_LOGIC_VECTOR(3 downto 0)
28              );
29      end component;
30
31      begin
32          ROM_component : rom
33              port map( address => in_system,
34                         clk => clk_system,
35                         rd => rd_system,
36                         out_value => u
37                 );
38
39          M_MACHINE_component : m_machine
40              port map( a_machine => u,
41                         y_machine => out_system
42                 );
43      end structural;

```

Listing 2.27: Implementazione del S System

L'entity s_system ha quattro porti: un input in_system (indirizzo a 4 bit), un input clk_system (segnale di clock), un input rd_system (segnale di controllo di lettura), e un output out_system (dato a 4 bit). L'architecture structural descrive la struttura interna del componente. Viene dichiarato un segnale interno **u** a 8 bit per memorizzare i dati letti dalla ROM. Vengono dichiarati due componenti: rom rappresenta una ROM e m_machine rappresenta una macchina sequenziale. Vengono poi istanziate due copie dei componenti. La ROM viene collegata all'input in_system, al clock clk_system, e al controllo di lettura rd_system, e l'output viene immagazzinato nel segnale u. Questo segnale u viene quindi utilizzato come input per la macchina sequenziale m_machine, la cui uscita è collegata all'output out_system del componente principale.

PO_PC

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use IEEE.numeric_std.ALL;
4
5 entity PO_PC is

```

```

6   port (
7     clock : in std_logic;
8     reset : in std_logic;
9     inizio : in std_logic;
10    uscita : out std_logic_vector(3 downto 0);
11    fine : out std_logic
12  );
13 end PO_PC;
14
15 architecture structural of PO_PC is
16
17   signal fine_conteggio : std_logic;
18   signal lettura : std_logic;
19   signal scrittura : std_logic;
20   signal abilitazione_contatore : std_logic;
21   signal indirizzo : std_logic_vector(3 downto 0);
22   signal dato : std_logic_vector(3 downto 0);
23
24   component s_system is
25     port (
26       in_system : in std_logic_vector(3 downto 0);
27       clk_system : in std_logic;
28       rd_system : in std_logic;
29       out_system : out std_logic_vector(3 downto 0)
30     );
31   end component;
32
33   component mem is
34     port (
35       address_mem : in std_logic_vector(3 downto 0);
36       data : in std_logic_vector(3 downto 0);
37       clk_mem : in std_logic;
38       wrt : in std_logic
39     );
40   end component;
41
42   component counter_mod_16 is
43     port (
44       clk_ctr, rst_ctr, en_ctr : in std_logic;
45       curr_value : out std_logic_vector(3 downto 0);
46       out_ctr : out std_logic

```

```

47      );
48  end component;
49
50  component control_unit is
51    port (
52      start    : in std_logic;
53      clk_cu   : in std_logic;
54      end_count : in std_logic;
55      rst_cu   : in std_logic;
56      rd_cu    : out std_logic;
57      wrt_cu   : out std_logic;
58      en_cu    : out std_logic;
59      end_cu   : out std_logic
60    );
61  end component;
62
63 begin
64   cu0: control_unit
65     port map(
66       start => inizio,
67       clk_cu => clock,
68       end_count => fine_conteggio,
69       rst_cu => reset,
70       rd_cu => lettura,
71       wrt_cu => scrittura,
72       en_cu => abilitazione_contatore,
73       end_cu => fine
74     );
75
76   ctr0 : counter_mod_16
77     port map(
78       clk_ctr => clock,
79       rst_ctr => reset,
80       en_ctr => abilitazione_contatore,
81       curr_value => indirizzo,
82       out_ctr => fine_conteggio
83     );
84
85   s_system0: s_system
86     port map(
87       in_system => indirizzo,

```

```

88         clk_system => clock,
89         rd_system => lettura,
90         out_system => dato
91     );
92
93     mem0 : mem
94     port map(
95         address_mem => indirizzo,
96         data => dato,
97         clk_mem => clock,
98         wrt => scrittura
99     );
100
101    uscita <= dato;
102 end structural;

```

Listing 2.28: Implementazione del PO PC

L'entity PO_PC ha quattro porti: un input clock (segnale di clock), un input reset (segnale di reset), un input inizio, un output fine che indica la fine di un'operazione o di un ciclo, è presente un ulteriore segnale di uscita, che consente di dare il dato memorizzato nel sistema PO_PC. L'architecture structural descrive la struttura interna del sistema. Vengono dichiarati vari segnali interni che vengono utilizzati per collegare i componenti e gestire lo stato del sistema. Vengono dichiarati poi quattro componenti: s_system, mem, counter_mod_16, e control_unit, che rappresentano rispettivamente un sistema sequenziale, una memoria ROM, un contatore modulo 16, e un'unità di controllo. Vengono istanziate quattro copie dei componenti, e i loro segnali di port map sono collegati tra loro. L'unità di controllo control_unit controlla il flusso del sistema in base a diversi segnali di input. Il contatore counter_mod_16 conta e segnala la fine del conteggio. Il processore sequenziale s_system opera sulla memoria in base all'indirizzo fornito. La memoria mem immagazzina e recupera dati in base all'indirizzo fornito.

2.7.3 Simulazione

Per simulare il componente è stato realizzato un **testbench**.

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use IEEE.numeric_std.ALL;
4
5 entity PO_PC_tb is
6 end PO_PC_tb;

```

```

7
8 architecture dataflow of PO_PC_tb is
9
10
11     signal clk : std_logic := '0';
12     signal rst : std_logic := '0';
13     signal start : std_logic := '0';
14     signal uscita_tb : std_logic_vector(3 downto 0);
15     signal fine_processo : std_logic;
16
17     constant clock_period : time := 10 ns;
18
19 begin
20     UUT: entity work.PO_PC
21         port map (
22             clock      => clk,
23             reset      => rst,
24             inizio     => start,
25             uscita     => uscita_tb,
26             fine       => fine_processo
27         );
28
29
30     timing_clock : process
31     begin
32         while now < 1000 ns loop
33             clk <= not clk;
34             wait for clock_period / 2;
35         end loop;
36         wait;
37     end process;
38
39
40     process
41     begin
42         rst <= '1';
43         start <= '0';
44         wait for 10 ns;
45
46         rst <= '0';
47         wait for 10 ns;

```

```

48      start <= '1';
49      wait for 50 ns;
50
51      start <= '0';
52      wait for 50 ns;
53      wait;
54  end process;
55
56 end dataflow;

```

Listing 2.29: Testbench del sistema PO PC

Questo testbench VHDL è progettato per simulare e verificare il comportamento di un'entità VHDL chiamata PO_PC. Il testbench non include l'implementazione di questo componente ma definisce come dovrebbe essere stimolata e testata all'interno dell'ambiente di simulazione. In primo luogo, è stata dichiarata una entity, PO_PC_tb, vuota dato che non rappresenta un oggetto fisico da realizzare, ma anzi è necessaria solo al fine di condurre i test. L'architettura dataflow del testbench definisce i segnali interni usati per stimolare l'entità sotto test:

- **clk**: Un segnale di clock utilizzato per sincronizzare l'entità.
- **rst**: Un segnale di reset per inizializzare o resettare l'entità.
- **start**: Un segnale utilizzato per indicare l'inizio di un processo nell'entità.
- **fine_processo**: Un segnale utilizzato per indicare la fine di un processo nell'entità.
- **uscita_tb**: Un segnale che porta in se l'uscita del sistema di elaborazione.

L'entità PO_PC è istanziata con il nome UUT (Unit Under Test) e i suoi porti sono mappati ai segnali definiti nel testbench. Il processo di generazione del clock, genera un segnale di clock che oscilla con un periodo definito dalla costante clock_period. Il loop interno inverte il valore del segnale di clock (clk) ogni metà periodo, creando così un'onda quadra. Il processo di stimolazione, definisce una sequenza di stimoli per i segnali rst e start: Inizialmente, rst è impostato ad alto ('1') e start a basso ('0'), poi dopo 10 ns, rst è portato a basso ('0') per uscire dalla condizione di reset. Dopo altri 10 ns, start è impostato ad alto ('1') per un periodo di 50 ns, indicando l'avvio di un processo nell'entità PO_PC. Infine, start viene riportato a basso ('0') e la simulazione continua per altri 50 ns prima di fermarsi. I risultati sono riportati nelle figure sottostanti:

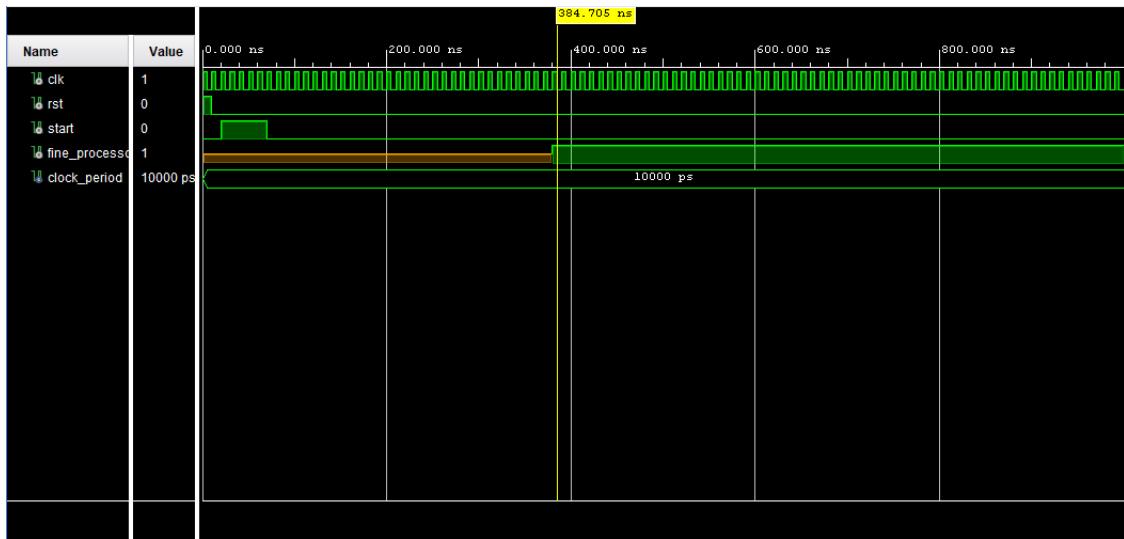


Figure 2.22: Simulazione

2.8 Esercizio 6.2 - Implementazione su board del PO_PC

Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando due bottoni per i segnali di read e reset rispettivamente e i led per la visualizzazione delle uscite della macchina istante per istante.

2.8.1 Progetto e architettura

Per portare su board il componente sviluppato abbiamo avuto la necessità di introdurre un nuovo componente il **Debouncer**, dato che quest'ultimo consente di pulire il segnale mandato grazie al click del bottone. Pertanto, è stato poi necessario introdurre un ulteriore file, `sistema_totale.vhd` che consentisse di istanziare il componente PO_PC e il componente Debouncer. Quest'ultimo è stato descritto più dettagliatamente nel capitolo successivo, ed in particolare nell'esercizio 7.2. Lo schema del componente è il seguente:

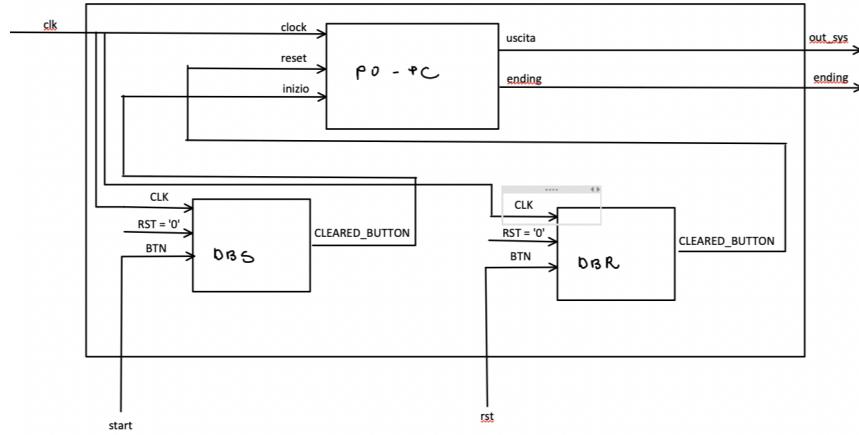


Figure 2.23: Schema POPC su board.

Il codice del sistema totale implementato, è riportato di seguito:

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use IEEE.numeric_std.ALL;
4
5 entity sistema_totale is
6     port (
7         clk: in std_logic;
8         rst: in std_logic;
9         start: in std_logic;
10        out_sis: out std_logic_vector(3 downto 0);
11        ending: out std_logic
12    );
13 end sistema_totale;
14
15 architecture structural of sistema_totale is
16     signal start_pulito: std_logic;
17     signal rst_pulito: std_logic;
18
19     component PO_PC is
20         port (
21             clock : in std_logic;
22             reset : in std_logic;
23             inizio : in std_logic;
24             uscita : out std_logic_vector(3 downto 0);
25             fine : out std_logic
26         );

```

```

27      end component;
28
29  component ButtonDebouncer is
30      generic (
31          CLK_period: integer := 10;    -- periodo del clock
32                      (della board) in nanosecondi
33          btn_noise_time: integer := 10000000 -- durata
34                      stimata dell'oscillazione del bottone in
35                      nanosecondi
36                      -- il valore
37                      di default
38                      10
39                      millisecondi
40      );
41
42      Port ( RST : in STD_LOGIC;
43                  CLK : in STD_LOGIC;
44                  BTN : in STD_LOGIC;
45                  CLEARED_BTN : out STD_LOGIC);
46
47  end component;
48
49
50
51  begin
52      popc: PO_PC
53      port map(
54          clock => clk,
55          reset => rst_pulito,
56          inizio => start_pulito,
57          uscita => out_sis,
58          fine => ending
59      );
60
61      dbs: ButtonDebouncer
62      generic map (
63          CLK_period => 10,
64          btn_noise_time => 10000000
65      )
66      port map(
67          RST => '0',
68          CLK => clk,
69          BTN => start,
70          CLEARED_BTN => start_pulito
71      );

```

```

62
63     dbr: ButtonDebouncer
64     generic map (
65         CLK_period => 10,
66         btn_noise_time => 10000000
67     )
68     port map (
69         RST => '0',
70         CLK => clk,
71         BTN => rst,
72         CLEARED_BTN => rst_pulito
73     );
74 end structural;

```

Listing 2.30: Implementazione su board del sistema PO PC

Come è possibile notare, l'approccio utilizzato è strutturale, e sono istanziati come anticipavamo, il sistema PO.PC e due debouncer dato che sono utilizzati due bottoni della board per dare il segnale di **start** e il segnale di reset. Questi due componenti avranno il compito di andare a pulire i segnali inoltrati grazie al click del bottone, restituendo in uscita i segnali di reset e start, però puliti. Per l'abilitazione dei bottoni e al fine di mostrare il segnale di **ending** sulla board, il file dei constraint è stato modificato abilitando i pin relativi ai bottoni **C12** ed **M18**. Il file è riportato di seguito (sono state riportate solo le parti fondamentali che abbiamo modificato):

```

1 ## This file is a general .xdc for the Nexys A7-100T
2 ## To use it INP a project:
3 ## - uncomment the lines corresponding to used pins
4 ## - rename the used ports (INP each line, after get_ports)
5 ## according to the top level signal names INP the project
6
7 ## Clock signal
8 set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 }
9     [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
10
11 ##Switches
12 #set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 }
13     [get_ports { in_system[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
14 #set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 }
15     [get_ports { in_system[1] }]; #IO_L3N_T0_DQS_EMCCCLK_14

```

```

Sch=sw[1]

14 #set_property -dict { PACKAGE_PIN M13    IOSTANDARD LVCMOS33 }
     [get_ports { in_system[2] }]; #IO_L6N_T0_D08_VREF_14
     Sch=sw[2]

15 #set_property -dict { PACKAGE_PIN R15    IOSTANDARD LVCMOS33 }
     [get_ports { in_system[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
16 #set_property -dict { PACKAGE_PIN R17    IOSTANDARD LVCMOS33 }
     [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
17 #set_property -dict { PACKAGE_PIN T18    IOSTANDARD LVCMOS33 }
     [get_ports { SW[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
18 #set_property -dict { PACKAGE_PIN U18    IOSTANDARD LVCMOS33 }
     [get_ports { SW[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
19 #set_property -dict { PACKAGE_PIN R13    IOSTANDARD LVCMOS33 }
     [get_ports { SW[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
20 #set_property -dict { PACKAGE_PIN T8     IOSTANDARD LVCMOS18 }
     [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
21 #set_property -dict { PACKAGE_PIN U8     IOSTANDARD LVCMOS18 }
     [get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
22 #set_property -dict { PACKAGE_PIN R16    IOSTANDARD LVCMOS33 }
     [get_ports { SW[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
23 #set_property -dict { PACKAGE_PIN T13    IOSTANDARD LVCMOS33 }
     [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
24 #set_property -dict { PACKAGE_PIN H6     IOSTANDARD LVCMOS33 }
     [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
25 #set_property -dict { PACKAGE_PIN U12    IOSTANDARD LVCMOS33 }
     [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
26 #set_property -dict { PACKAGE_PIN U11    IOSTANDARD LVCMOS33 }
     [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14
     Sch=sw[14]

27 #set_property -dict { PACKAGE_PIN V10    IOSTANDARD LVCMOS33 }
     [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]
28
29 ## LEDs
30 set_property -dict { PACKAGE_PIN H17    IOSTANDARD LVCMOS33 }
     [get_ports { out_sys[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
31 set_property -dict { PACKAGE_PIN K15    IOSTANDARD LVCMOS33 }
     [get_ports { out_sys[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
32 set_property -dict { PACKAGE_PIN J13    IOSTANDARD LVCMOS33 }
     [get_ports { out_sys[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
33 set_property -dict { PACKAGE_PIN N14    IOSTANDARD LVCMOS33 }
     [get_ports { out_sys[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]

```

```

34 #set_property -dict { PACKAGE_PIN R18    IOSTANDARD LVCMOS33 }
      [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
35 #set_property -dict { PACKAGE_PIN V17    IOSTANDARD LVCMOS33 }
      [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
36 #set_property -dict { PACKAGE_PIN U17    IOSTANDARD LVCMOS33 }
      [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
37 #set_property -dict { PACKAGE_PIN U16    IOSTANDARD LVCMOS33 }
      [get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
38 #set_property -dict { PACKAGE_PIN V16    IOSTANDARD LVCMOS33 }
      [get_ports { LED[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
39 #set_property -dict { PACKAGE_PIN T15    IOSTANDARD LVCMOS33 }
      [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
40 #set_property -dict { PACKAGE_PIN U14    IOSTANDARD LVCMOS33 }
      [get_ports { LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
41 #set_property -dict { PACKAGE_PIN T16    IOSTANDARD LVCMOS33 }
      [get_ports { LED[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14
      Sch=led[11]
42 #set_property -dict { PACKAGE_PIN V15    IOSTANDARD LVCMOS33 }
      [get_ports { LED[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
43 #set_property -dict { PACKAGE_PIN V14    IOSTANDARD LVCMOS33 }
      [get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
44 #set_property -dict { PACKAGE_PIN V12    IOSTANDARD LVCMOS33 }
      [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
45 #set_property -dict { PACKAGE_PIN V11    IOSTANDARD LVCMOS33 }
      [get_ports { LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14
      Sch=led[15]
46
47 ## RGB LEDs
48 set_property -dict { PACKAGE_PIN R12    IOSTANDARD LVCMOS33 }
      [get_ports { ending }]; #IO_L5P_T0_D06_14 Sch=led16_b
49 #set_property -dict { PACKAGE_PIN M16    IOSTANDARD LVCMOS33 }
      [get_ports { LED16_G }]; #IO_L10P_T1_D14_14 Sch=led16_g
50 #set_property -dict { PACKAGE_PIN N15    IOSTANDARD LVCMOS33 }
      [get_ports { LED16_R }]; #IO_L11P_T1_SRCC_14 Sch=led16_r
51 #set_property -dict { PACKAGE_PIN G14    IOSTANDARD LVCMOS33 }
      [get_ports { LED17_B }]; #IO_L15N_T2_DQS_ADV_B_15
      Sch=led17_b
52 #set_property -dict { PACKAGE_PIN R11    IOSTANDARD LVCMOS33 }
      [get_ports { LED17_G }]; #IO_0_14 Sch=led17_g
53 #set_property -dict { PACKAGE_PIN N16    IOSTANDARD LVCMOS33 }
      [get_ports { LED17_R }]; #IO_L11N_T1_SRCC_14 Sch=led17_r

```

```

54
55 ##7 segment display
56 #set_property -dict { PACKAGE_PIN T10    IOSTANDARD LVCMOS33 }
57     [get_ports { cathodes[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
58 #set_property -dict { PACKAGE_PIN R10    IOSTANDARD LVCMOS33 }
59     [get_ports { cathodes[1] }]; #IO_25_14 Sch=cb
60 #set_property -dict { PACKAGE_PIN K16    IOSTANDARD LVCMOS33 }
61     [get_ports { cathodes[2] }]; #IO_25_15 Sch=cc
62 #set_property -dict { PACKAGE_PIN K13    IOSTANDARD LVCMOS33 }
63     [get_ports { cathodes[3] }]; #IO_L17P_T2_A26_15 Sch=cd
64 #set_property -dict { PACKAGE_PIN P15    IOSTANDARD LVCMOS33 }
65     [get_ports { cathodes[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
66 #set_property -dict { PACKAGE_PIN T11    IOSTANDARD LVCMOS33 }
67     [get_ports { cathodes[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
68 #set_property -dict { PACKAGE_PIN L18    IOSTANDARD LVCMOS33 }
69     [get_ports { cathodes[6] }]; #IO_L4P_T0_D04_14 Sch=cg
70 #set_property -dict { PACKAGE_PIN H15    IOSTANDARD LVCMOS33 }
71     [get_ports { cathodes[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
72 #set_property -dict { PACKAGE_PIN J17    IOSTANDARD LVCMOS33 }
73     [get_ports { anodes[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
74 #set_property -dict { PACKAGE_PIN J18    IOSTANDARD LVCMOS33 }
75     [get_ports { anodes[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
76 #set_property -dict { PACKAGE_PIN T9     IOSTANDARD LVCMOS33 }
77     [get_ports { anodes[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
78 #set_property -dict { PACKAGE_PIN J14    IOSTANDARD LVCMOS33 }
79     [get_ports { anodes[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
80 #set_property -dict { PACKAGE_PIN P14    IOSTANDARD LVCMOS33 }
81     [get_ports { anodes[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
82 #set_property -dict { PACKAGE_PIN T14    IOSTANDARD LVCMOS33 }
83     [get_ports { anodes[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
84 #set_property -dict { PACKAGE_PIN K2     IOSTANDARD LVCMOS33 }
85     [get_ports { anodes[6] }]; #IO_L23P_T3_35 Sch=an[6]
86 #set_property -dict { PACKAGE_PIN U13    IOSTANDARD LVCMOS33 }
87     [get_ports { anodes[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

88
89 ##Buttons
90 set_property -dict { PACKAGE_PIN C12    IOSTANDARD LVCMOS33 }
91     [get_ports { start }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
92 #set_property -dict { PACKAGE_PIN N17    IOSTANDARD LVCMOS33 }
93     [get_ports { reset }]; #IO_L9P_T1_DQS_14 Sch=btnc
94 set_property -dict { PACKAGE_PIN M18    IOSTANDARD LVCMOS33 }

```

```
[get_ports { rst }]; #IO_L4N_T0_D05_14 Sch=btnu  
77 #set_property -dict { PACKAGE_PIN P17 IOSTANDARD LVCMOS33 }  
[get_ports { buttoni[0] }]; #IO_L12P_T1_MRCC_14 Sch=btnl  
78 #set_property -dict { PACKAGE_PIN M17 IOSTANDARD LVCMOS33 }  
[get_ports { buttoni[1] }]; #IO_L10N_T1_D15_14 Sch=btnr  
79 #set_property -dict { PACKAGE_PIN P18 IOSTANDARD LVCMOS33 }  
[get_ports { BTND }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

Listing 2.31: Constraint del sistema PO PC

CHAPTER 3

MACCHINE ARITMETICHE

3.1 Esercizio 7.1 - Moltiplicatore di Booth

Progettare, implementare in VHDL e simulare una macchina moltiplicatore di Booth in grado di effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna.

3.1.1 Progetto e architettura

Il moltiplicatore di Booth è un **moltiplicatore sequenziale**, ed è stato progettato per eseguire la moltiplicazione binaria con segno in modo efficiente, riducendo il numero di operazioni necessarie per ottenere il prodotto di due numeri binari. L'algoritmo sfrutta la **codifica di Booth**, la quale effettua una mappatura di due bit adiacenti con un valore dell'intervallo -1, 0, 1:

$x_j x_{j-1}$	codifica
00/11	0
01	+1
10	-1

Figure 3.1: codifica di Booth.

mediante la codifica è possibile **ridurre il numero di operazioni da effettuare**:

$x_j x_{j-1}$	
00/11	Non viene effettuata né la somma né la sottrazione, ma solo lo shift
01	Y viene aggiunto al prodotto parziale corrente
10	Y viene sottratto dal prodotto parziale corrente

Figure 3.2: operazioni

Gli elementi che permettono la realizzazione del moltiplicatore sono i seguenti:

- **registro AQ** : è uno **shift register** di 17 bit, adibito al contenimento dei prodotti parziali. Notiamo che esso è di 17 bit, in quanto **dobbiamo distinguere le casistiche in base a q0 e q-1**.
- **registro M** : un registro semplice ad **8 bit** che contiene il moltiplicatore.

- **multiplexer 2:1** : adibito alla selezione del contenuto di AQ, lo shift register, dunque, può scegliere tra il valore iniziale del moltiplicando o il risultato parziale appena calcolato.
- **contatore modulo 8** : scandisce gli 8 passaggi della moltiplicazione.
- **addizionatore/sottrattore** : un ripple carry adder che aggiunge la modalità di sottrazione. Sceglie la modalità a seconda dei bit q0 e q-1.

Questi elementi formano insieme l'**unità operativa**.

- **unità di controllo** : coordina tutte le operazioni da eseguire in base allo stato in cui si trova.

Lo schema finale realizzato è il seguente:

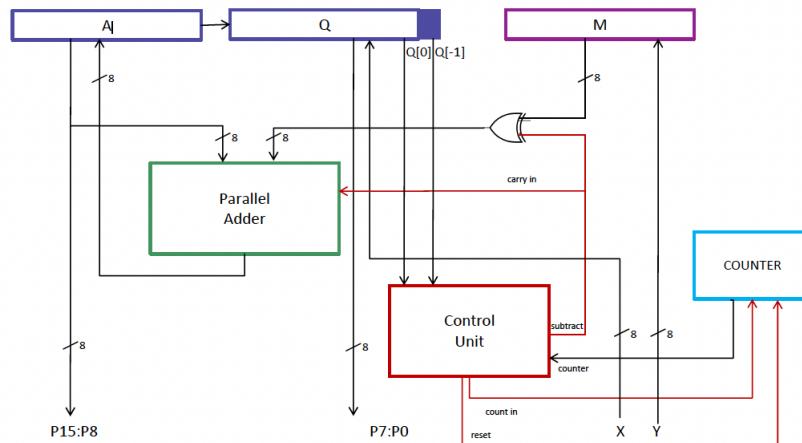


Figure 3.3: architettura del moltiplicatore di Booth.

3.1.2 Implementazione

Non riportiamo le implementazioni di shift register, multiplexer 2:1 e contatore modulo 8, in quanto già presenti nei capitoli precedenti. Discutiamo adesso dell'implementazione dell'addizionatore/sottrattore, il quale ci consente, a seconda dei casi, di effettuare addizione o sottrazione. Per prima cosa, è possibile descrivere il componente **full-adder**:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity full_adder is
5     port (

```

```

6   a,b: in std_logic;
7   cin: in std_logic;
8   cout, s: out std_logic);
9 end full_adder;

10

11

12 architecture rtl of full_adder is

13

14 begin

15

16   s<= a xor b xor cin;
17   cout<= (a and b) or (cin and (a xor b));
18

19 end rtl;

```

Esso presenta come ingressi i due bit degli **addendi**, a e b , e il riporto in ingresso cin , mentre come uscite il riporto uscente $cout$, e il risultato della somma s . Nell'architettura rtl è esplicitata la legge per il calcolo della somma e del riporto uscente.

Più full-adder posti in cascata realizzano un **ripple carry adder**:

```

,
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity ripple_carry is
5   port( X, Y: in std_logic_vector(7 downto 0);
6     c_in: in std_logic;
7     c_out: out std_logic;
8     Z: out std_logic_vector(7 downto 0));
9 end ripple_carry;

10

11 architecture structural of ripple_carry is
12   component full_adder is
13   port(
14     a,b: in std_logic;
15     cin: in std_logic;
16     cout, s: out std_logic);
17   end component;

18

19   signal temp: std_logic_vector(7 downto 0);
20

```

```

21   begin
22
23     RA0: full_adder port map(X(0), Y(0), c_in, temp(0), Z(0));
24
25     RA1to6: FOR i IN 1 TO 6 GENERATE
26       RA: full_adder port map(X(i), Y(i), temp(i-1), temp(i),
27         Z(i));
28     END GENERATE;
29
30     RA7: full_adder port map(X(7), Y(7), temp(6), c_out, Z(7));
31 end structural;

```

Esso prevede in ingresso i due **addendi ad 8 bit**, X ed Y, ed il riporto in ingresso c_in, mentre in uscita il riporto uscente c_out, e la somma Z, sempre ad 8 bit. Nell'architettura strutturale vengono istanziati 8 full-adder, uno per ciascun bit degli addendi. Ciascun full-adder prende in ingresso la coppia di bit i-esima degli addendi, e, fatta eccezione per il primo, mediante il segnale d'appoggio temp, ciascun riporto entrante sarà il riporto uscente del precedente FA.

L'addizionatore/sottrattore utilizza l'RCA e incorpora la meccanica per il calcolo dell'addendo Y, operando una XOR con il riporto in ingresso. Infatti, se il riporto in ingresso è pari a 1, la sottrazione sarà abilitata, e la XOR trasformerà Y nel suo complemento. Al contrario, se il riporto è 0, la XOR con 0 sarà neutra, consentendo di effettuare una semplice addizione.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity adder_sub is
5   port( X, Y: in std_logic_vector(7 downto 0);
6         cin: in std_logic;
7         Z: out std_logic_vector(7 downto 0);
8         cout: out std_logic);
9 end adder_sub;
10
11 architecture structural of adder_sub is
12   component ripple_carry is
13     port( X, Y: in std_logic_vector(7 downto 0);
14           c_in: in std_logic;
15           c_out: out std_logic;
16           Z: out std_logic_vector(7 downto 0));

```

```

17  end component;

18

19  signal complementoy: std_logic_vector(7 downto 0);

20

21  begin

22

23  complemento_y: FOR i IN 0 TO 7 GENERATE
24      complementoy(i)<=Y(i) xor cin;
25  END GENERATE;

26

27  RA: ripple_carry port map(X, complementoy, cin, cout, Z);

28

29 end structural;

```

Listing 3.1: Implementazione dell'addizionatore/sottrattore.

Analizziamo adesso l'unità di controllo, disegnata come un automa a stati finiti, la quale si occupa del prelievo degli operandi della moltiplicazione, della verifica degli ultimi bit meno significativi al fine di stabilire la prossima operazione da eseguire e di abilitare i conteggi.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity unita_controllo is
6     port( q : in std_logic_vector(1 downto 0);
7             clock, reset, start: in std_logic;
8             count: in std_logic_vector(2 downto 0);
9             loadM, count_in, loadAQ, en_shift: out std_logic;
10            selAQ, subtract, stop_cu: out std_logic
11        );
12 end unita_controllo;
13
14 architecture structural of unita_controllo is
15     type state is (idle, fetch, wait_op, scan, rshift,
16                     increment, fine);
17     signal current_state, next_state: state;
18
19  begin
    reg_stato: process(clock) --processo per effettuare il
        cambio di stato.

```

```

20      begin
21          if(clock'event and clock='1') then
22              if(reset='1') then
23                  current_state <= idle;
24              else
25                  current_state <= next_state;
26              end if;
27          end if;
28      end process;
29
30      comb: process(current_state, start, count)
31      begin
32          count_in <= '0'; --inizializzazione dei segnali di
33          input.
34          subtract <= '0';
35          selAQ <= '0';
36          loadAQ <= '0';
37          loadM <= '0';
38          stop_cu <= '0';
39          en_shift <= '0';
40
41          case current_state is
42              when idle =>
43                  if(start = '1') then -- quando viene dato lo
44                  start
45                  --possiamo prelevare gli operandi della moltiplicazione
46                  next_state <= fetch;
47              else
48                  next_state <= idle;
49              end if;
50
51              when fetch =>
52                  loadM <= '1'; -- carichiamo M ed AQ con il
53                  moltiplicando e
54                  -- il prodotto parziale.
55                  loadAQ <= '1';
56                  next_state <= wait_op;
57
58              when wait_op =>
59                  next_state <= scan; -- attesa.
60
61      end begin;
62
63      end component;
64
65      end architecture;

```

```

58      when scan =>
59          if(q = "01") then -- somma + shift.
60              selAQ <= '1';
61              loadAQ <= '1';
62              next_state <= rshift;
63          elsif(q = "10") then -- sottrazione + shift.
64              subtract <= '1';
65              selAQ <= '1';
66              loadAQ <= '1';
67              next_state <= rshift;
68          elsif (q = "00" or q = "11") then -- shift.
69              next_state <= rshift;
70      end if;
71
72      when rshift =>
73          en_shift <= '1';
74          if(count = "111") then -- fine conteggio
75              quando abbiamo eseguito
76                  --8 volte le operazioni.
77                  next_state <= fine;
78          else
79              next_state <= increment;
80      end if;
81
82      when increment =>
83          count_in <= '1';
84          next_state <= scan;
85
86      when fine =>
87          stop_cu <= '1';
88          next_state <= idle;
89      end case;
90  end process;
91 end structural;

```

Listing 3.2: Implementazione dell’unità di controllo.

In particolare, i suoi ingressi sono i seguenti:

- q : vettore contenente i due bit meno significativi presenti all’interno dello shift register AQ.
- $clock$: il clock di sistema.

- `reset` : segnale di reset del sistema.
- `start` : segnale di avvio di tutte le operazioni.
- `count` : vettore rappresentante il conteggio del contatore.

e le uscite:

- `loadM` : segnale di load per il registro M.
- `count_in` : segnale di temporizzazione per il contatore.
- `loadAQ` : segnale di load per lo shift register AQ.
- `en_shift` : segnale di abilitazione per lo shift a destra.
- `sel_AQ` : segnale di selezione per il multiplexer collegato all'ingresso dello shift register AQ.
- `subtract` : segnale di abilitazione della modalità "sottrattore" dell'addizionatore/sottrattore.
- `stop_cu` : segnale di terminazione delle attività della control unit.

Gli stati che in cui può trovarsi la richiesta sono:

- `idle` : stato di inattività.
- `fetch` : stato in cui avviene il prelievo degli operandi.
- `wait_op` : stato in cui si attende la terminazione del prelievo degli operandi.
- `scan` : stato in cui vengono analizzati i due LSB in modo da determinare la prossima operazione.
- `rshift` : stato in cui avviene lo shift a destra.
- `increment` : stato in cui viene incrementato il conteggio.
- `fine` : stato di terminazione.

Nell'architettura strutturale del componente, vengono innanzitutto dichiarati due segnali di supporto, indicanti lo stato corrente (`current_state`) e lo stato successivo (`next_state`). Successivamente, sono implementati due processi: il primo gestisce il **cambiamento di stato** al fronte di clock, mentre il secondo si occupa di determinare la **transizione da uno stato all'altro**. Concentriamoci sul secondo processo. Inizialmente, vengono inizializzati i segnali di uscita del componente. Successivamente, mediante il costrutto `case`, vengono esaminati i diversi stati. Nel caso in

cui l'automa si trovi nello stato **"idle"** e il segnale di start venga alzato, avverrà il passaggio allo stato **"fetch"**; in caso contrario, rimarrà in **"idle"**. Se si trova nello stato **"fetch"**, vengono attivati i segnali di caricamento dei registri loadM e loadAQ. Successivamente, si transita allo stato **"wait_op"**, da cui si passa direttamente allo stato **"scan"**. Nello stato **"scan"**, vengono analizzati i due bit q0 e q-1. Se essi risultano pari a **"01"**, si verifica una somma e uno shift. Di conseguenza, vengono attivati i segnali selAQ (per acquisire il contenuto parziale di AQ) e loadAQ (per ricaricare AQ con un nuovo risultato parziale), prima di passare allo stato **"shift"**. Nel caso in cui i bit siano pari a **"10"**, si verifica una sottrazione e uno shift, con l'attivazione del segnale subtract. Infine, se i bit sono pari a **"11"** o **"00"**, si procede direttamente allo stato di shift **"rshift"**. Nello stato **"rshift"**, viene abilitato il segnale en_shift e viene effettuato un controllo sulla fine del conteggio per determinare se passare allo stato **"fine"** o **"increment"**, nel quale viene incrementato il conteggio. Nello stato di **"fine"**, verrà abilitato il segnale stop_cu, consentendo di transitare nello stato **"idle"**

Passiamo adesso all'unità operativa

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity unita_operativa is
6     port( X, Y: in std_logic_vector(7 downto 0);
7             clock, reset: in std_logic;
8             loadAQ, shift, loadM, sub, selAQ, count_in: in
9                 std_logic;
10            count: out std_logic_vector(2 downto 0);
11            P: out std_logic_vector(16 downto 0));
12
13 architecture structural of unita_operativa is
14
15     component adder_sub is
16         port( X, Y: in std_logic_vector(7 downto 0);
17                 cin: in std_logic;
18                 Z: out std_logic_vector(7 downto 0);
19                 cout: out std_logic);
20
21     end component;
```

```

22 component registro8 is
23     port( A: in std_logic_vector(7 downto 0);
24             clk, res, load: in std_logic;
25             B: out std_logic_vector(7 downto 0));
26 end component;
27
28
29 component mux_21 is
30     generic (width : integer range 0 to 17 := 8);
31     port( x0, x1: in std_logic_vector(width-1 downto 0);
32             s: in std_logic;
33             y: out std_logic_vector(width-1 downto 0));
34 end component;
35
36
37 component shift_register is
38     port( parallel_in: in std_logic_vector(16 downto 0);
39             serial_in: in std_logic;
40             clock, reset, load, shift: in std_logic;
41             parallel_out: out std_logic_vector(16 downto
42             0));
43 end component;
44
45 component cont_mod8 is
46     port( clock, reset: in std_logic;
47             count_in: in std_logic;
48             count: out std_logic_vector(2 downto 0));
49 end component;
50
51 signal Mreg: std_logic_vector(7 downto 0);
52 signal AQ_init: std_logic_vector(16 downto 0);
53 signal AQ_in: std_logic_vector(16 downto 0);
54 signal AQ_out: std_logic_vector(16 downto 0);
55 signal partial: std_logic_vector(7 downto 0);
56 signal AQ_sum_in : std_logic_vector(16 downto 0);
57 signal riporto: std_logic;
58 signal SRserialIn: std_logic;
59
60 begin
61

```

```

62     M: registro8 port map(Y, clock, reset, loadM, Mreg);
63
64     AQ_init <= "00000000" & X & "0";
65
66     AQ_sum_in <= partial & AQ_out(8 downto 0);
67
68     MUX_SR_parallel_in : mux_21 generic map (width => 17) port map
69         (AQ_init, AQ_sum_in, selAQ, AQ_in);
70
71     SR: shift_register port map(AQ_in, SRserialIn, clock,
72         reset, loadAQ, shift, AQ_out);
73
74     ADD_SUB: adder_sub port map(AQ_out(16 downto 9), Mreg,
75         sub, partial, riporto); --
76
77     CONT: cont_mod8 port map(clock, reset, count_in, count);
78
79     P <= AQ_out;
80 end structural;

```

Listing 3.3: Implementazione dell’unità operativa.

presenta come ingressi:

- X e Y : vettore di 8 bit che memorizza moltiplicando e moltiplicatore.
- clock : clock di sistema.
- reset : segnale di reset.
- loadAQ : segnale di load per lo shift register AQ.
- shift : segnale di abilitazione per lo shift a destra.
- loadM : segnale di load per il registro M.
- sub : segnale di abilitazione della modalità ”sottrattore” dell’addizionatore/sottrattore, il quale corrisponde al riporto entrante dell’addizionatore/sottrattore.
- sel_AQ : segnale di selezione per il multiplexer collegato all’ingresso dello shift register AQ.
- count_in : segnale di temporizzazione per il contatore.

e le seguenti uscite:

- count : conteggio attuale del contatore.
- P : prodotto della moltiplicazione.

Nell'architettura strutturale sono stati dichiarati come componenti: l'**addizionatore** (adder/sub), il **registro** che conterrà il moltiplicatore Y (registro 8), il **multiplexer** 2:1 che permette la selezione del valore da caricare in AQ, dunque, tra la stringa iniziale e la somma parziale (mux_21), il registro a scorrimento che conterrà il moltiplicando X (shift_register), e il **contatore** modulo 8 che indica quante volte effettuare le operazioni (cont_mod8). Inoltre, sono stati dichiarati i seguenti segnali temporanei per i collegamenti:

- signal Mreg : permette il collegamento tra il registro del moltiplicatore e l'addizionatore/sottrattore.
- AQ_init : serve ad inizializzare lo shift register, dunque, rappresenta uno degli ingressi del multiplexer collegato ad AQ.
- AQ_in : collega l'uscita del MUX all'ingresso shift register AQ.
- AQ_out : contenuto (uscita) di AQ, i cui primi 8 bit andranno in ingresso all'addizionatore/sottrattore.
- partial : somma parziale in uscita dall'addizionatore/sottrattore.
- AQ_sum_in : concatenazione della somma parziale e la stringa presente nel registro Q, che verrà collegata ad uno degli ingressi del mux.
- riporto : segnale di riporto.
- SRserialIn : ingresso dello shift register, che coincide al bit più significativo della sua uscita.

Infine collegheremo a P l'uscita del registro a scorrimento AQ.

Infine, discutiamo dell'implementazione del moltiplicatore:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity molt_booth is
```

```

6   port( clock, reset, start: in std_logic;
7     X, Y: in std_logic_vector(7 downto 0);
8     product: out std_logic_vector(15 downto 0);
9     stop_cu: out std_logic);
10 end molt_booth;
11
12 architecture structural of molt_booth is
13   component unita_controllo is
14     port( q : std_logic_vector(1 downto 0);
15       clock, reset, start: in std_logic;
16       count: in std_logic_vector(2 downto 0);
17       loadM, count_in, loadAQ, en_shift: out std_logic;
18       selAQ, subtract, stop_cu: out std_logic);
19   end component;
20
21   component unita_operativa is
22     port( X, Y: in std_logic_vector(7 downto 0);
23       clock, reset: in std_logic;
24       loadAQ, shift, loadM, sub, selAQ, count_in: in std_logic;
25       count: out std_logic_vector(2 downto 0);
26       P: out std_logic_vector(16 downto 0));
27   end component;
28
29
30   signal tempq : std_logic_vector(1 downto 0);
31   signal temp_selAQ, temp_sub, temp_loadAQ: std_logic;
32   signal temp_count: std_logic_vector(2 downto 0);
33   signal temp_p: std_logic_vector(16 downto 0);
34   signal temp_count_in : std_logic;
35   signal temp_shift: std_logic;
36   signal temp_loadM: std_logic;
37
38 begin
39
40   UC: unita_controllo port map
41     q => tempq,
42     clock => clock,
43     reset => reset,
44     start => start,
45     count => temp_count,
46     loadM => temp_loadM,

```

```

47      count_in => temp_count_in,
48      loadAQ => temp_loadAQ,
49      en_shift => temp_shift,
50      selAQ => temp_selAQ,
51      subtract => temp_sub,
52      stop_cu => stop_cu
53  );
54
55  UO: unita_operativa port map (
56      X => X,
57      Y => Y,
58      clock => clock,
59      reset => reset,
60      loadAQ => temp_loadAQ,
61      shift => temp_shift,
62      loadM => temp_loadM,
63      sub => temp_sub,
64      selAQ => temp_selAQ,
65      count_in => temp_count_in,
66      count => temp_count,
67      P => temp_p
68  );
69
70  tempq<=temp_p(1 downto 0);
71  product<=temp_p(16 downto 1);
72
73 end structural;

```

Listing 3.4: Implementazione del moltiplicatore di Booth.

Vede i seguenti ingressi:

- **clock** : il clock di sistema.
- **reset** : il segnale di reset del sistema
- **start** : il segnale di avvio dell'intero processo di moltiplicazione.
- **X** : moltiplicando di 8 bit.
- **Y** : moltiplicatore di 8 bit.

E come uscite:

- **product** : risultato della moltiplicazione a 16 bit.

- stop_cu : segnale di stop che indica il termine del processo di moltiplicazione.

L'entità molt_booth ci consente di effettuare gli opportuni collegamenti tra l'unità di controllo e l'unità operativa, difatti, sono stati dichiarati i seguenti segnali d'appoggio:

- tempq : permette di salvare gli ultimi due bit del registro di scorrimento.
- temp_selAQ : consente il collegamento per il segnale di selezione presente all'interno dell'unità operativa.
- temp_sub : collegamento per l'abilitazione del segnale di sottrazione all'interno dell'unità operativa.
- temp_loadAQ : collegamento per il caricamento del registro di scorrimento.
- temp_count : mediante il quale l'unità operativa comunica il conteggio attuale all'unità di controllo.
- temp_p : segnale d'appoggio che mantiene il contenuto attuale del registro di scorrimento.
- temp_count_in : mediante il quale l'unità di controllo abilita il conteggio del contatore all'interno dell'unità operativa.
- temp_shift : collegamento per l'abilitazione dello shift all'interno dell'unità operativa.
- temp_loadM : collegamento per il caricamento del registro M.

3.1.3 Simulazione

Passiamo ora all'analisi del codice del testbench:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.numeric_std.ALL;
4
5 entity molt_booth_tb is
6 end molt_booth_tb;
7
8 architecture behavioural of molt_booth_tb is
9
```

```

10 component molt_booth is
11   port( clock, reset, start: in std_logic;
12         X, Y: in std_logic_vector(7 downto 0);
13         product: out std_logic_vector(15 downto 0);
14         stop_cu: out std_logic);
15 end component;
16
17 signal inputx, inputy: std_logic_vector(7 downto 0);
18 signal prod: std_logic_vector(15 downto 0);
19 signal clk, res, start: std_logic;
20 signal t_stop_cu: std_logic;
21 constant clk_period : time := 20 ns;
22
23 signal end_sim : std_logic := '0';
24
25
26 begin
27
28   uut: molt_booth port map(clk, res, start, inputx, inputy,
29                             prod, t_stop_cu);
30
31   clk_process : process
32     begin
33       while (end_sim = '0') loop
34         clk<= '1';
35         wait for clk_period/2;
36         clk <= '0';
37         wait for clk_period/2;
38       end loop;
39       wait;
40     end process;
41
42   sim: process
43     begin
44       wait for 100 ns;
45
46       res<='1';
47       wait for 20 ns;
48       res<='0';
49       wait for 10 ns;

```

```

50      inputx<="00001111"; -- 15*3=45 (002D)
51      inputy<="00000011";
52      wait for 40 ns;
53      start<='1';
54      wait for 20 ns;
55      start<='0';
56      wait for 10 ns;
57      wait;
58  end process;
59 end behavioural;

```

Listing 3.5: Implementazione del test bench del moltiplicatore di Booth.

Ovviamente l'elemento sottoposto ai test (*uut*) è il moltiplicatore di Booth, i cui ingressi e uscite sono stati mappati in questo modo:

- clock con il segnale clk.
- reset con il segnale res.
- start con il segnale start.
- X ed Y con i segnali inputx ed inputy.
- product con il segnale prod.
- stop_cu con il segnale t_stop_cu.

E' stata dichiarata una costante *clk_period*, indicante il periodo di clock, pari a 20 nanosecondi. Sono presenti due processi nell'architettura comportamentale del testbench: il primo è il processo che realizza l'impulso di clock, *clk_process*, il secondo, *sim*, fornisce gli stimoli. Concentriamoci sul secondo processo, innanzitutto avviene il reset, poi vengono inizializzati i due operandi, inputx a 15 ("00001111"), mentre input y a 3 ("00000011"), poi viene alzato start, e infine abbassato. Il risultato dell'operazione deve essere 45, che corrisponde a 002D in esadecimale, e possiamo osservarlo sul Vivado:

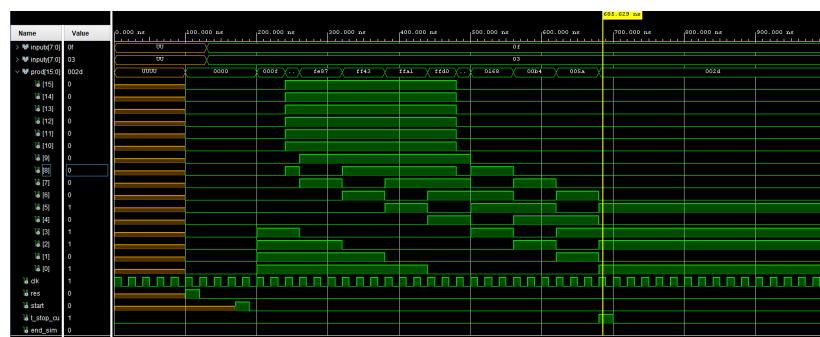


Figure 3.4: Risultato della simulazione del tb del moltiplicatore di Booth.

Possiamo anche osservare tutti i risultati intermedi.

3.2 Esercizio 7.2 - Implementazione su board

Sintetizzare il moltiplicatore implementato al punto 7.1 su FPGA e testarlo mediante l'utilizzo dei dispositivi di input/output (switch, bottoni, led, display) presenti sulla board di sviluppo in dotazione. La modalità di utilizzo degli stessi è a completa discrezione degli studenti.

3.2.1 Progetto e architettura

Per andare ad implementare su board il precedente esercizio si è scelto di riutilizzare il componente **molt_booth** con l'aggiunta di un **ButtonDebouncer** che andasse a ripulire il segnale che viene inviato mediante la pressione dei bottoni presenti sulla board, questo a causa del fatto che una pressione prolungata di un pulsante potrebbe scaturire la lettura di segnali rispetto alle intenzioni dell'utente, quindi il segnale inviato, grazie all'utilizzo di questo sistema sarà sempre unico e della durata di un colpo di clock. Lo schema del sistema è riportato nella figura seguente:

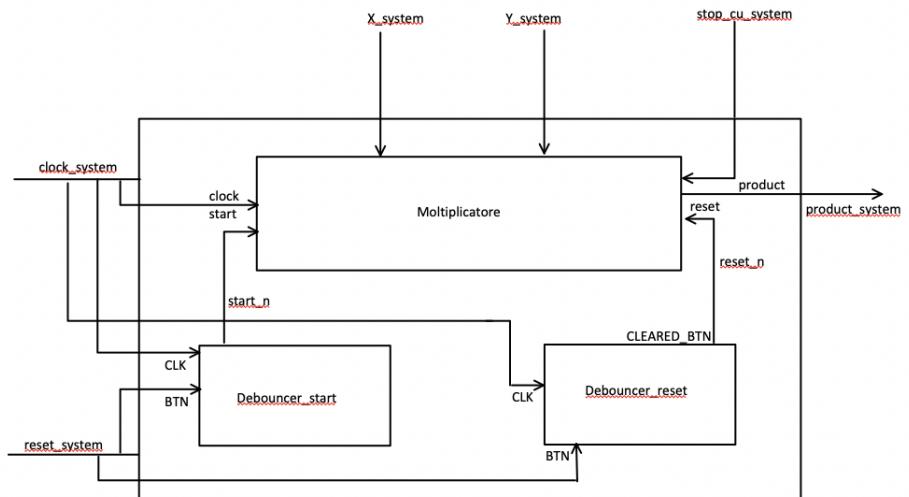


Figure 3.5: Schema moltiplicatore di Booth, board.

3.2.2 Implementazione

In primo luogo si analizza il nuovo componente:

```
entity ButtonDebouncer is
```

```

2      generic (
3          CLK_period: integer := 10;
4          btn_noise_time: integer := 10000000
5
6      );
7      Port ( RST : in STD_LOGIC;
8              CLK : in STD_LOGIC;
9              BTN : in STD_LOGIC;
10             CLEARED_BTN : out STD_LOGIC);
11 end ButtonDebouncer;
12
13 architecture Behavioral of ButtonDebouncer is
14
15
16
17 type stato is (NOT_PRESSED, CHK_PRESSED, PRESSED,
18                 CHK_NOT_PRESSED);
19 signal BTN_state : stato := NOT_PRESSED;
20 constant max_count : integer := btn_noise_time/CLK_period;
21
22 begin
23
24 deb: process (CLK)
25 variable count: integer := 0;
26
27 begin
28     if rising_edge(CLK) then
29
30         if( RST = '1') then
31             BTN_state <= NOT_PRESSED;
32             CLEARED_BTN <= '0';
33         else
34             case BTN_state is
35             when NOT_PRESSED =>
36                 if( BTN = '1' ) then
37                     BTN_state <= CHK_PRESSED;
38                 else
39                     BTN_state <= NOT_PRESSED;
40             end if;
41             when CHK_PRESSED =>

```

```

42         if(count = max_count -1) then
43             if(BTN = '1') then
44                 count:=0;
45                 CLEARED_BTN <= '1';
46                 BTN_state <= PRESSED;
47             else
48                 count:=0;
49                 BTN_state <= NOT_PRESSED;
50             end if;
51
52         else
53             count:= count+1;
54             BTN_state <= CHK_PRESSED;
55         end if;
56
57     when PRESSED =>
58         CLEARED_BTN<= '0';
59
60         if(BTN = '0') then
61             BTN_state <= CHK_NOT_PRESSED;
62         else
63             BTN_state <= PRESSED;
64         end if;
65
66     when CHK_NOT_PRESSED =>
67         if(count = max_count -1) then
68             if(BTN = '0') then
69                 count:=0;
70                 BTN_state <= NOT_PRESSED;
71             else
72                 count:=0;
73                 BTN_state <= PRESSED;
74             end if;
75
76         else
77             count:= count+1;
78             BTN_state <= CHK_NOT_PRESSED;
79         end if;
80
81     when others =>
82         BTN_state <= NOT_PRESSED;

```

```

83      end case;
84  end if;
85 end if;
86 end process;
87
88
89 end Behavioral;

```

Listing 3.6: Implementazione del ButtonDebouncer.

Partendo dall’entity sono stati inseriti, come generic, due interi che definiscono il periodo del clock della board e la durata stimata per l’oscillazione del bottone. Come port abbiamo il segnale di reset (RST), il segnale di clock (CLK), il segnale BTN che è quello che viene alzato in seguito alla pressione del bottone ed, infine, il segnale CLEARED-BTN, messo in out, che rappresenta il segnale unico che viene restituito a seguito della pressione del bottone. Il comportamento è descritto mediante un automa a stati finiti che presenta 4 stati:

- **NOT_PRESSED** : verifica l’arrivo del segnale BTN relativo alla pressione del bottone e passa al prossimo stato in presenza di segnale alto.
- **CHK_PRESSED** : se il segnale del bottone perdura per almeno 999 mila di colpi di clock, alza il segnale CLEARED-BTN e passa allo stato successivo, in caso contrario torna allo stato precedente.
- **PRESSED** : in questo stato viene abbassato il segnale CLEARED-BTN e si aspetta che il segnale derivante dalla pressione del bottone si abbassi per passare al prossimo stato.
- **CHK_NOT_PRESSED** : se il segnale BTN resta basso per almeno 999 mila di colpi di clock, si torna allo stato NOT_PRESSED, altrimenti si torna allo stato PRESSED.

Per quanto riguarda la mappatura del componente appena descritto all’interno del moltiplicatore, si è deciso di utilizzarne due, uno per il segnale di **start** ed uno per il segnale di **reset** nel seguente modo:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity molt_booth is
6   port( clock, reset, start: in std_logic;

```

```

7      X, Y: in std_logic_vector(7 downto 0);
8      product: out std_logic_vector(15 downto 0);
9      stop_cu: out std_logic);
10 end molt_booth;
11
12 architecture structural of molt_booth is
13     component unita_controllo is
14         port( q : in std_logic_vector(1 downto 0);
15             clock, reset, start: in std_logic;
16             count: in std_logic_vector(2 downto 0);
17             loadM, count_in, loadAQ, en_shift: out std_logic;
18             selaQ, subtract, stop_cu: out std_logic);
19     end component;
20
21     component unita_operativa is
22         port( X, Y: in std_logic_vector(7 downto 0);
23             clock, reset: in std_logic;
24             loadAQ, shift, loadM, sub, selaQ, count_in: in std_logic;
25             count: out std_logic_vector(2 downto 0);
26             P: out std_logic_vector(16 downto 0));
27     end component;
28
29     component ButtonDebouncer
30         Generic (
31             CLK_period: integer := 10;
32             btn_noise_time: integer := 10000000
33         );
34         Port (
35             RST : in STD_LOGIC;
36             CLK : in STD_LOGIC;
37             BTN : in STD_LOGIC;
38             CLEARED_BTN : out STD_LOGIC
39         );
40     end component;
41
42     signal tempq : std_logic_vector(1 downto 0);
43     signal temp_selAQ, temp_sub, temp_loadAQ: std_logic;
44     signal temp_count: std_logic_vector(2 downto 0);
45     signal temp_p: std_logic_vector(16 downto 0);
46     signal temp_count_in : std_logic;
47     signal temp_shift: std_logic;

```

```

48     signal temp_loadM: std_logic;
49
50     signal start_n: std_logic;
51     signal reset_n: std_logic;
52
53 begin
54
55     UC: unita_controllo port map (
56         q => tempq,
57         clock => clock,
58         reset => reset_n,
59         start => start_n,
60         count => temp_count,
61         loadM => temp_loadM,
62         count_in => temp_count_in,
63         loadAQ => temp_loadAQ,
64         en_shift => temp_shift,
65         selAQ => temp_selAQ,
66         subtract => temp_sub,
67         stop_cu => stop_cu
68     );
69
70     UO: unita_operativa port map (
71         X => X,
72         Y => Y,
73         clock => clock,
74         reset => reset_n,
75         loadAQ => temp_loadAQ,
76         shift => temp_shift,
77         loadM => temp_loadM,
78         sub => temp_sub,
79         selAQ => temp_selAQ,
80         count_in => temp_count_in,
81         count => temp_count,
82         P => temp_p
83     );
84
85     start_db: ButtonDebouncer
86         Generic map (
87             CLK_period => 10,
88             btn_noise_time => 10000000

```

```

89      )
90      Port map (
91          RST => reset_n,
92          CLK => clock,
93          BTN => start,
94          CLEARED_BTN => start_n
95      );
96
97      reset_db: ButtonDebouncer
98      Generic map (
99          CLK_period => 10,
100         btn_noise_time => 10000000
101     )
102     Port map (
103        RST => '0',
104        CLK => clock,
105        BTN => reset,
106        CLEARED_BTN => reset_n
107    );
108
109    tempq<=temp_p(1 downto 0);
110    product<=temp_p(16 downto 1);
111
112 end structural;

```

Listing 3.7: Mappatura dei componenti a seguita dell'aggiunta del debouncer.

Sono stati aggiunti i segnali temporanei *start-n* e *reset-n*, il primo ha il compito di inviare il segnale di CLEARED-BTN all'unità di controllo per permettere l'avvio; per quanto riguarda *reset-n*, questo è utilizzato come segnale di *reset* di tutti i componenti, questo, però, è il segnale di uscita del secondo Debouncer presentato, il quale non fa altro che ripulire il segnale di *reset* ottenuto dalla pressione del pulsante che si è scelto di utilizzare.

3.2.3 Simulazione

Al fine di utilizzare la board per questo esercizio si è andato a modificare il file di configurazione `Nexys-A7-100T-Master.xdc` in questo modo:

```

1 ##Switches
2 set_property -dict { PACKAGE_PIN J15    IO_STANDARD LVCMOS33 }
   [get_ports { X[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]

```

```

3 set_property -dict { PACKAGE_PIN L16    IOSTANDARD LVCMOS33 }
      [get_ports { X[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
4 set_property -dict { PACKAGE_PIN M13    IOSTANDARD LVCMOS33 }
      [get_ports { X[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
5 set_property -dict { PACKAGE_PIN R15    IOSTANDARD LVCMOS33 }
      [get_ports { X[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
6 set_property -dict { PACKAGE_PIN R17    IOSTANDARD LVCMOS33 }
      [get_ports { X[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
7 set_property -dict { PACKAGE_PIN T18    IOSTANDARD LVCMOS33 }
      [get_ports { X[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
8 set_property -dict { PACKAGE_PIN U18    IOSTANDARD LVCMOS33 }
      [get_ports { X[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
9 set_property -dict { PACKAGE_PIN R13    IOSTANDARD LVCMOS33 }
      [get_ports { X[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
10 set_property -dict { PACKAGE_PIN T8     IOSTANDARD LVCMOS18 }
      [get_ports { Y[0] }]; #IO_L24N_T3_34 Sch=sw[8]
11 set_property -dict { PACKAGE_PIN U8     IOSTANDARD LVCMOS18 }
      [get_ports { Y[1] }]; #IO_25_34 Sch=sw[9]
12 set_property -dict { PACKAGE_PIN R16    IOSTANDARD LVCMOS33 }
      [get_ports { Y[2] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
13 set_property -dict { PACKAGE_PIN T13    IOSTANDARD LVCMOS33 }
      [get_ports { Y[3] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
14 set_property -dict { PACKAGE_PIN H6     IOSTANDARD LVCMOS33 }
      [get_ports { Y[4] }]; #IO_L24P_T3_35 Sch=sw[12]
15 set_property -dict { PACKAGE_PIN U12    IOSTANDARD LVCMOS33 }
      [get_ports { Y[5] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
16 set_property -dict { PACKAGE_PIN U11    IOSTANDARD LVCMOS33 }
      [get_ports { Y[6] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
17 set_property -dict { PACKAGE_PIN V10    IOSTANDARD LVCMOS33 }
      [get_ports { Y[7] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

```

Gli **switch** sono adibiti all'inserimento dei vettori operandi *X* ed *Y*.

```

1 ## LEDs
2 set_property -dict { PACKAGE_PIN H17    IOSTANDARD LVCMOS33 }
      [get_ports { product[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
3 set_property -dict { PACKAGE_PIN K15    IOSTANDARD LVCMOS33 }
      [get_ports { product[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
4 set_property -dict { PACKAGE_PIN J13    IOSTANDARD LVCMOS33 }
      [get_ports { product[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
5 set_property -dict { PACKAGE_PIN N14    IOSTANDARD LVCMOS33 }
      [get_ports { product[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]

```

```

6 set_property -dict { PACKAGE_PIN R18    IOSTANDARD LVCMOS33 }
      [get_ports { product[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
7 set_property -dict { PACKAGE_PIN V17    IOSTANDARD LVCMOS33 }
      [get_ports { product[5] }]; #IO_L18N_T2_A11_D27_14
      Sch=led[5]
8 set_property -dict { PACKAGE_PIN U17    IOSTANDARD LVCMOS33 }
      [get_ports { product[6] }]; #IO_L17P_T2_A14_D30_14
      Sch=led[6]
9 set_property -dict { PACKAGE_PIN U16    IOSTANDARD LVCMOS33 }
      [get_ports { product[7] }]; #IO_L18P_T2_A12_D28_14
      Sch=led[7]
10 set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 }
      [get_ports { product[8] }]; #IO_L16N_T2_A15_D31_14
      Sch=led[8]
11 set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 }
      [get_ports { product[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
12 set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 }
      [get_ports { product[10] }]; #IO_L22P_T3_A05_D21_14
      Sch=led[10]
13 set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 }
      [get_ports { product[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14
      Sch=led[11]
14 set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 }
      [get_ports { product[12] }]; #IO_L16P_T2_CSI_B_14
      Sch=led[12]
15 set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 }
      [get_ports { product[13] }]; #IO_L22N_T3_A04_D20_14
      Sch=led[13]
16 set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 }
      [get_ports { product[14] }]; #IO_L20N_T3_A07_D23_14
      Sch=led[14]
17 set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 }
      [get_ports { product[15] }]; #IO_L21N_T3_DQS_A06_D22_14
      Sch=led[15]

```

I **led** sono stati utilizzati per andare a leggere il vettore che contiene il prodotto della nostra moltiplicazione.

```

1 ##Buttons
2 #set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 }
      [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15
      Sch=cpu_resetn

```

```

3 set_property -dict { PACKAGE_PIN N17    IOSTANDARD LVCMOS33 }
      [get_ports { start }]; #IO_L9P_T1_DQS_14 Sch=btnc
4 #set_property -dict { PACKAGE_PIN M18    IOSTANDARD LVCMOS33 }
      [get_ports { BTNU }]; #IO_L4N_T0_D05_14 Sch=btnu
5 #set_property -dict { PACKAGE_PIN P17    IOSTANDARD LVCMOS33 }
      [get_ports { b_secondi }]; #IO_L12P_T1_MRCC_14 Sch=btln
6 #set_property -dict { PACKAGE_PIN M17    IOSTANDARD LVCMOS33 }
      [get_ports { b_ore }]; #IO_L10N_T1_D15_14 Sch=btnr
7 set_property -dict { PACKAGE_PIN P18    IOSTANDARD LVCMOS33 }
      [get_ports { reset }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd

```

I **bottoni** utilizzati per lo start e per il reset sono, rispettivamente, **N17** e **P18**.

Per la simulazione sulla board, sono stati inseriti gli stessi valori presenti nel testbench dell'esercizio precedente.

3.2.4 Timing Analysis

Per effettuare la timing analysis in Vivado è necessario prima di tutto effettuare la sintesi, che consentirebbe di eseguire già l'analisi ma con delle somme approssimate, quindi abbiamo effettuato anche l'implementazione. Nella schermata “Report Timing Summary” è possibile configurare i parametri della timing analysis, che specificano il tipo di report da generare e il contenuto da mostrare una volta eseguita l'analisi. In particolare, abbiamo selezionato min e max, come “Path delay type”, per misurare il Worst Negative Slack (WNS), cioè il tempo che impiega un segnale di input a stabilizzarsi prima del fronte successivo del clock, tale che le uscite raggiungano il valore desiderato e il Worst Hold Slack (WHS), cioè il tempo per cui un segnale di input deve restare stabile dopo il fronte del clock per consentire all'output di raggiungere il valore desiderato. Lo slack indica la differenza tra require time e arrival time. In questa analisi viene misurato anche il Worst Pulse Width Slack (WPWS) che indica il peggiore tra tutti i controlli considerando i ritardi sia minimi che massimi. Il parametro maximum number of path per clock indica quanti percorsi possono essere analizzati simultaneamente in un singolo ciclo di clock, mentre il maximum number of worst paths per endpoint consente di limitare il numero massimo di percorsi peggiori analizzati per ciascun endpoint.

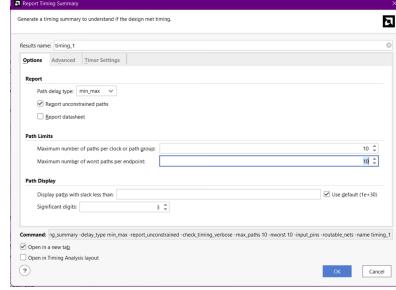


Figure 3.6: Report Timing Summary

È importante che il WNS sia positivo, perché altrimenti vuol dire che il percorso sarà fallito. Di seguito sono riportati i risultati per quanto riguarda il clock creato:

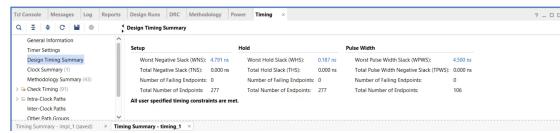


Figure 3.7: Risultati.

Una volta controllato questo risultato, abbiamo calcolato anche la FMAX, cioè la frequenza massima di funzionamento. Essa non è fornita esplicitamente nei report ma l'abbiamo stimata con la seguente formula:

$$\frac{1}{T-WNS}$$

dove T è il periodo del clock target. Per trovare questo valore è possibile diminuire progressivamente il periodo del clock di design, finché non si ottiene un WNS negativo. In particolare, abbiamo diminuito il periodo fino a 3.5 ns con una forma d'onda con fronte di salita in 0 ns e fronte di discesa a 1.75 ns period 3.50 waveform 1.75 Il valore approssimato è 281 MHZ. Oltre questa frequenza i vincoli temporali non sono più rispettati.

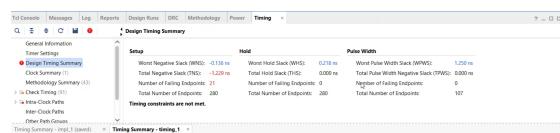


Figure 3.8: Risultati con WNS negativo.

CHAPTER 4

COMUNICAZIONE CON HANDSHAKING

4.1 Esercizio 8.1 - Comunicazione con handshaking

Progettare, implementare in VHDL e testare mediante simulazione un sistema composto da 2 nodi, A e B, che comunicano mediante un protocollo di handshaking. Il nodo A e il nodo B possiedono entrambi una memoria interna in cui sono memorizzate N stringhe di M bit, denominate X(i) e Y(i) rispettivamente ($i=0,..,N-1$). Il nodo A trasmette a B ciascuna stringa X(i) utilizzando un protocollo di handshaking; B, ricevuta la stringa X(i), calcola $S(i)=X(i)+Y(i)$ e immagazzina la somma in opportune locazioni della propria memoria interna.

Per il progetto è possibile considerare una implementazione di tipo comportamentale per effettuare la somma, mentre è necessario prevedere esplicitamente un componente contatore sia nel sistema A sia nel sistema B per scandire la trasmissione/ricezione delle stringhe e per terminare la comunicazione.

4.2 Progetto e architettura

Per l'implementazione del sistema è stato utilizzato un approccio **strutturale**, dato che il sistema può essere decomposto in sistemi più piccoli e semplici che collegati tra loro consentono di generare il comportamento desiderato. Il sistema è caratterizzato da due nodi: A e B, ciascuno dei quali è a sua volta caratterizzato da **unità di controllo** e **unità operativa**. L'unità di controllo ha il compito di generare i segnali di controllo per impartire i comandi all'unità operativa che invece effettua le operazioni necessarie a generare il comportamento atteso dal sistema. Come è possibile notare dallo schema rappresentativo del sistema: l'**unità operativa di A** è caratterizzata da una memoria **ROM** e un contatore **mod-16** che consente di scandirne le locazioni da fornire in uscita. Per quanto riguarda l'**unità operativa di B**, è invece caratterizzata da una memoria **ROM**, una **memoria** sulla quale scriveremo i risultati dell'elaborazione ed un contatore **mod-16**. L'unità operativa di B, avrà il compito di effettuare la somma tra il dato inviato dall'unità A e il dato presente nella locazione che stiamo attualmente considerando della ROM del sistema. Lo schema del sistema è rappresentato di seguito.

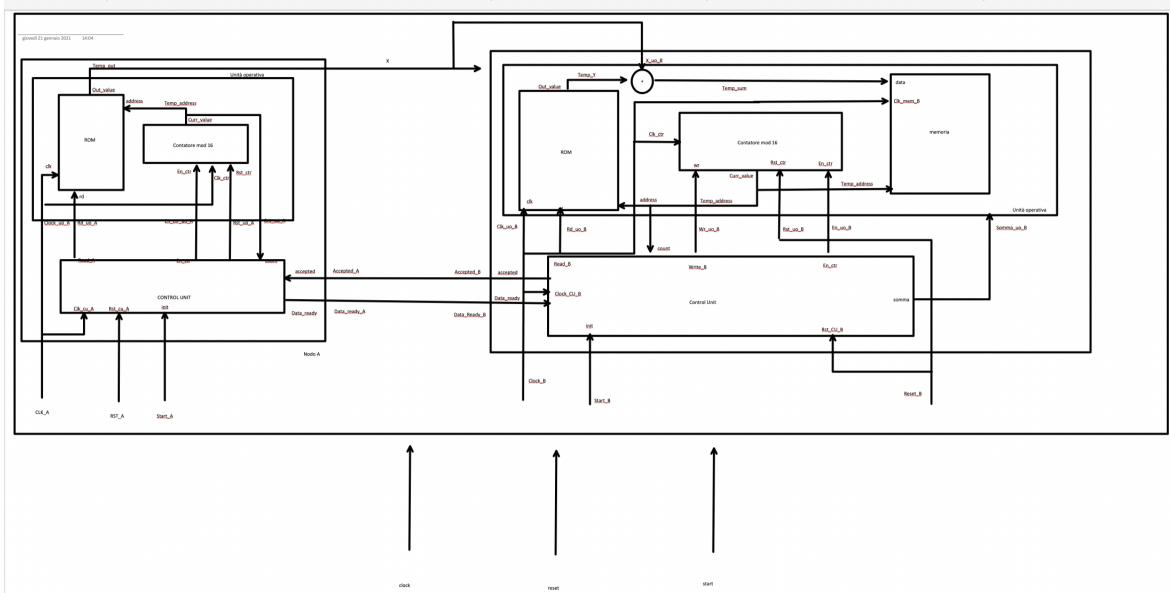


Figure 4.1: handshake tra due nodi.

Per quanto riguarda l'unità di controllo del sistema A, di seguito è riportato l'automa a stati finiti che ne descrive il comportamento:

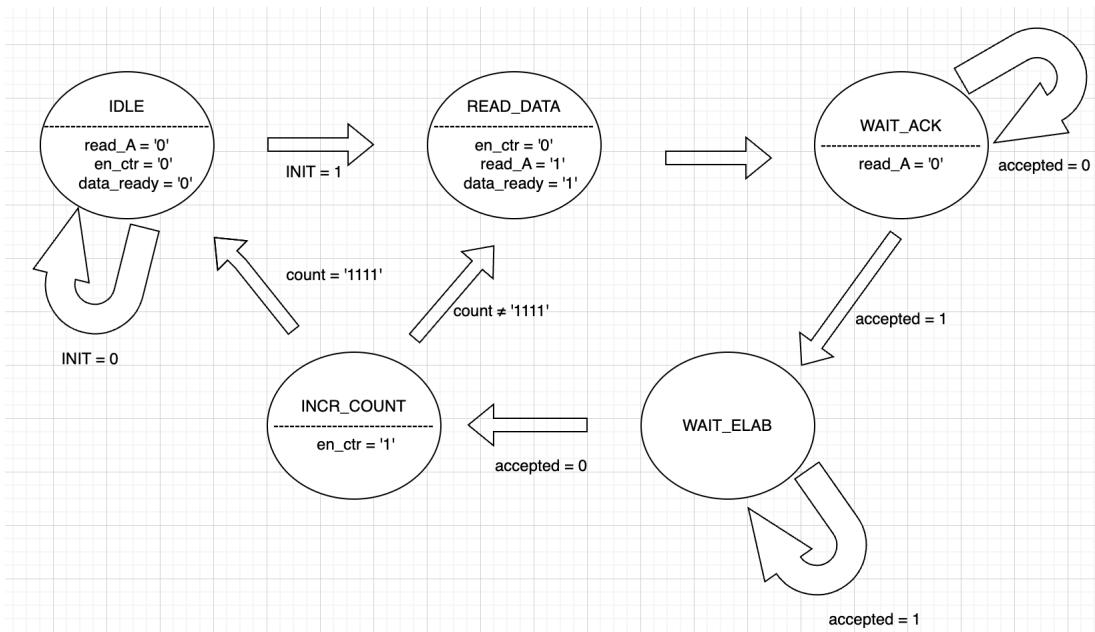


Figure 4.2: ASF control unit di A.

È possibile distinguere 5 stati differenti: **idle**, **read data**, **wait ack**, **wait elab** ed infine **incr count**.

- **IDLE**: Questo è lo stato di riposo della macchina. In questo stato, non ci sono letture in corso, il contatore è disabilitato e non ci sono dati pronti da poter inviare al nodo B mediante . Se il segnale **init** è '1', la macchina passa allo stato **READ DATA**; altrimenti, permane nello stato **IDLE**.

- **READ DATA:** In questo stato, il sistema inizia a leggere i dati ponendo a '1' il segnale di lettura e segnalando che i dati sono pronti per essere letti impostando data ready a '1'. Il contatore continua a non essere abilitato. Dopo aver letto i dati, la macchina passa allo stato **WAIT ACK**.
- **WAIT ACK:** In tale stato, la macchina attende il segnale di accepted da parte del nodo B. Se accepted è '1', significa che i dati sono stati accettati da quest'ultimo, la macchina prepara il sistema per la prossima elaborazione impostando data ready a '0' e passando allo stato **WAIT ELAB**. Se accepted non è '1', la macchina rimane in attesa (WAIT ACK) dato che il nodo B non ha terminato la lettura dei dati.
- **WAIT ELAB:** In questo stato si attende il completamento dell'elaborazione dei dati da parte del nodo B. Se il segnale accepted diventa '0', indicando che l'elaborazione è terminata, la macchina passa allo stato **INCR COUNT** per incrementare il contatore e consentire la lettura di una nuova locazione di memoria. Altrimenti, rimane nello stato **WAIT ELAB**.
- **INCR COUNT:** In questo stato, il contatore è abilitato, pertanto il conteggio viene incrementato. Sarà effettuato un controllo sul valore corrente di conteggio dato che se il contatore ha raggiunto il valore "**1111**", la macchina ritorna allo stato **IDLE**, indicando che il ciclo di operazioni è completo. Se il contatore non ha raggiunto il valore "**1111**", la macchina passa di nuovo allo stato **READ DATA** per leggere gli ulteriori dati.

In definitiva, il ciclo di stati presentato consente alla macchina a stati finiti di leggere dati, attendere l'accettazione e l'elaborazione di questi dati, e incrementare il contatore o riprendere la lettura dei dati.

Per quanto riguarda l'unità di controllo del sistema B, invece, di seguito è riportato l'automa a stati finiti che ne descrive il comportamento:

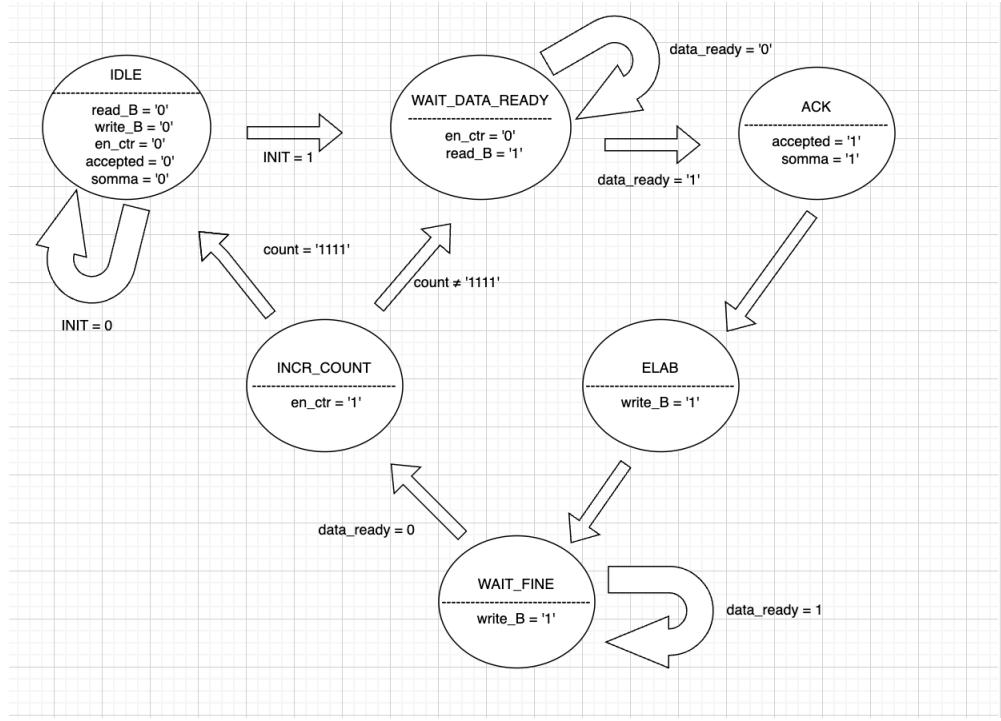


Figure 4.3: ASF control unit di B.

Anche in questo caso è possibile distinguere gli stati. L'ASF presenta 6 stati differenti: **idle**, **wait data ready**, **ack**, **elab**, **wait fine** e **incr count**.

- **IDLE**: È lo stato iniziale. In questo stato, tutti i segnali sono impostati a '0', indicando che non ci sono operazioni in corso. Se il segnale 'init' è '1', l'automa passa allo stato **WAIT DATA READY**; altrimenti, rimane in **IDLE**.
- **WAIT DATA READY**: In questo stato, l'automa si prepara a leggere i dati impostando **read B** a '1'. L'automa rimane in questo stato finché il segnale **data ready** non diventa '1', indicando che i dati sono pronti. A quel punto, passa allo stato **ACK**.
- **ACK**: Qui, l'automa segnala l'accettazione dei dati impostando **accepted** a '1' ed impedisce il comando per realizzare l'operazione di somma. Successivamente, passa allo stato **ELAB** per elaborare i dati.
- **ELAB**: In questo stato, l'automa elabora i dati e dopo questa fase, passa allo stato di **WAIT FINE** per attendere la fine dell'operazione.
- **WAIT FINE**: In stato si attende la fine dell'elaborazione dei dati. Se **data ready** diventa '0', ciò indica che l'elaborazione è terminata. L'automa resetta i segnali **somma**, **accepted** e **read B** a '0' e passa allo stato **INCR COUNT** per incrementare il contatore. Se **data ready** rimane '1', l'automa rimane in attesa.

- **INCR COUNT**: In questo stato, l'automa incrementa il contatore . Se il contatore raggiunge il valore "1111", l'automa ritorna allo stato **IDLE**, altrimenti torna allo stato **WAIT DATA READY** per iniziare un nuovo ciclo di lettura ed elaborazione dei dati.

L'automa procede dunque alla lettura e all'accettazione dei dati , elabora i dati, attende la fine dell'elaborazione ed infine incrementa il contatore.

4.2.1 Implementazione

Come è possibile notare dallo schema sono presenti numerosi componenti, possiamo analizzarli:

NODO A

il nodo A è stato realizzato mediante un approccio strutturale, interconnettendo unità di controllo ed unità operativa.

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use work.all;
4
5 entity nodo_A is
6   port (
7     start_A : in std_logic;
8     clk_A : in std_logic;
9     rst_A : in std_logic;
10    accepted_A : in std_logic;
11    data_ready_A : out std_logic;
12    X : out std_logic_vector(7 downto 0)
13  );
14 end nodo_A;
```

Listing 4.1: Entity del nodo A

L'entità nodo A rappresenta una componente che interagisce con altre parti del sistema attraverso vari segnali. È caratterizzata da diversi segnali di input: start A usato per dire al nodo quando iniziare ad operare, clk A ossia il segnale di clock, rst A utilizzato per riportare il nodo al suo stato iniziale, accepted A e data ready A, segnali a supporto dell'handshake. X è un vettore di 8 bit di dati prodotto da nodo A, che rappresenta il contenuto della locazione della memoria ROM che abbiamo appena analizzato.

```

1 architecture structural of nodo_A is
2   signal temp_rd : std_logic;
3   signal temp_cnt : std_logic_vector(3 downto 0);
```

```

4     signal temp_en_ctr : std_logic; -- Rinominato da
5         temp_en_cnt per chiarezza
6
7 component unitacontrollo_A is
8     port (
9         init : in std_logic;
10        clk_cu_A : in std_logic;
11        rst_cu_A : in std_logic;
12        read_A : out std_logic;
13        count : in std_logic_vector(3 downto 0);
14        en_ctrl : out std_logic;
15        accepted : in std_logic;
16        data_ready : out std_logic
17    );
18
19 end component;
20
21
22 component unita_operativa_A is
23     port (
24         clk_uo_A : in std_logic;
25         rst_uo_A : in std_logic;
26         en_ctrl_uo_A : in std_logic;
27         rd_uo_A : in std_logic;
28         cnt_uo_A : out std_logic_vector(3 downto 0);
29         out_data_uo_A : out std_logic_vector(7 downto 0)
30     );
31
32 end component;
33
34
35 begin
36     uc_A : unitacontrollo_A
37         port map (
38             init => start_A,
39             clk_cu_A => clk_A,
40             rst_cu_A => rst_A,
41             read_A => temp_rd,
42             count => temp_cnt,
43             en_ctrl => temp_en_ctrl,
44             accepted => accepted_A,
45             data_ready => data_ready_A
46         );
47
48     uo_A : unita_operativa_A

```

```

44     port map(
45         clk_uo_A => clk_A,
46         rst_uo_A => rst_A,
47         en_ctr_uo_A => temp_en_ctrl,
48         rd_uo_A => temp_rd,
49         cnt_uo_A => temp_cnt,
50         out_data_uo_A => X
51     );
52
53 end structural;

```

Listing 4.2: Architecture del nodo A

L'architecture sovrastante è di tipo strutturale, infatti definisce i componenti che nel loro insieme danno vita al nodo A. Come è possibile evincere, sono presenti l'unità di controllo e l'unità operativa. Queste vengono definite e mappate con i segnali definiti nell'interfaccia del componente e con i segnali di appoggio definiti nella parte dichiarativa dell'architettura.

UNITÀ OPERATIVA DI A

Vediamo adesso nel dettaglio l'unità operativa di A, ossia il componente che realizza l'operazione richiesta al componente.

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use work.all;
4
5 entity unita_operativa_A is
6     port( clk_uo_A : in std_logic;
7             rst_uo_A : in std_logic;
8             en_ctr_uo_A : in std_logic;
9             rd_uo_A : in std_logic;
10            cnt_uo_A : out std_logic_vector(3 downto 0);
11            out_data_uo_A : out std_logic_vector(7 downto 0)
12        );
13 end unita_operativa_A;

```

Listing 4.3: Entity UO di A

La definizione del componente unita operativa A mette in luce i vari porti, che le permettono di comunicare con l'esterno. I segnali di input sono: il segnale di clock, di reset, di abilitazione per il contatore ed il segnale di abilitazione alla lettura per la memoria ROM. I segnali di output invece consistono in un vettore che restituisce alla control unit il valore di conteggio e il dato letto dalla memoria ROM.

```

1 architecture structural of unita_operativa_A is
2     signal temp_address : std_logic_vector(3 downto 0);
3     signal temp_out : std_logic_vector(7 downto 0);
4
5     component memoria_A is
6         port (
7             address : in std_logic_vector(3 downto 0);
8             clk : in std_logic;
9             rd : in std_logic;
10            out_value : out std_logic_vector(7 downto 0)
11        );
12    end component;
13
14    component counter_mod_16 is
15        port (
16            clk_ctr, rst_ctr, en_ctr : in std_logic;
17            curr_value : out std_logic_vector(3 downto 0)
18        );
19    end component;
20
21    begin
22        rom_A : memoria_A
23            port map( address => temp_address,
24                         clk => clk_uo_A,
25                         rd => rd_uo_A,
26                         out_value => temp_out
27            );
28
29        ctr : counter_mod_16
30            port map( clk_ctr => clk_uo_A,
31                         rst_ctr => rst_uo_A,
32                         en_ctr => en_ctr_uo_A,
33                         curr_value => temp_address
34            );
35
36        cnt_uo_A <= temp_address;
37        out_data_uo_A <= temp_out;
38
39    end structural;

```

Per quanto riguarda l'architecture, essa è di tipo strutturale dato che collega insieme i vari componenti: ROM e counter mod 16 (per la descrizione dei componenti si

veda il capitolo 2, ed in particolare l'esercizio 6.1). Il comportamento implementato è il seguente: il contatore sulla base dei segnali di abilitazione impartiti dall'unità di controllo scandisce ad ogni colpo di clock una locazione diversa di memoria. Quest'ultima dovrà naturalmente essere abilitata da parte dell'unità di controllo, al fine di leggere il contenuto della locazione selezionato. **UNITA DI CONTROLLO DI A**

```

1      library IEEE;
2  use IEEE.std_logic_1164.ALL;
3  use work.all;
4
5  entity unitacontrollo_A is
6    port(  init : in std_logic;
7           clk_cu_A : in std_logic;
8           rst_cu_A : in std_logic;
9
10          -- ROM A
11          read_A : out std_logic;
12
13          -- CONTATORE .
14          count : in std_logic_vector(3 downto 0);
15          en_ctr : out std_logic;
16
17          --HANDSHAKE
18          accepted : in std_logic;
19          data_ready : out std_logic
20        );
21 end unitacontrollo_A;
```

L'entità rappresenta il cuore del sistema. Questa entità è come il direttore d'orchestra per vari componenti, gestendo flussi di dati, segnali di controllo e sincronizzazione. I segnali di entrata che la caratterizzano sono: il segnale di inizializzazione, il clock, il reset, il count ossia il valore del conteggio che gli viene restituito dal contatore dell'unità operativa, ed infine il segnale accepted per la gestione del protocollo di handshake. Per quanto riguarda i segnali di uscita invece, essi sono: il segnale di lettura, di abilitazione del contatore ed il segnale data ready per la realizzazione del protocollo di handshake.

```

1  architecture structural of unitacontrollo_A is
2    type state is (IDLE, READ_DATA, WAIT_ACK, WAIT_ELAB,
3                  INCR_COUNT);
4    signal current_state, next_state : state;
```

```

4
5      begin
6
7          reg_stateo: process(clk_cu_A)
8              begin
9                  if(clk_cu_A'event and clk_cu_A='1') then
10                     if(rst_cu_A='1') then
11                         current_state <=IDLE;
12                     else
13                         current_state <=next_state;
14                     end if;
15                     end if;
16                 end process;
17
18             comb: process(current_state, init, accepted, count)
19                 begin
20
21                     case current_state is
22                         when IDLE =>
23                             read_A <= '0';
24                             en_ctrl <= '0';
25                             data_ready <= '0';
26                             if(init = '1') then
27                                 next_state <= READ_DATA;
28                             else
29                                 next_state <= IDLE;
30                             end if;
31
32                         when READ_DATA =>
33                             en_ctrl <= '0';
34                             read_A <= '1';
35                             data_ready <= '1';
36                             next_state <= WAIT_ACK;
37
38                         when WAIT_ACK =>
39                             read_A <= '0';
40                             if(accepted = '1') then
41                                 data_ready <= '0';
42                                 next_state <= WAIT_ELAB;
43                             else
44                                 next_state <= WAIT_ACK;

```

```

45           end if;
46
47       when WAIT_ELAB =>
48           if(accepted = '0') then
49               next_state <= INCR_COUNT;
50           else
51               next_state <= WAIT_ELAB;
52           end if;
53
54       when INCR_COUNT =>
55           en_ctr <= '1';
56
57           if(count = "1111") then
58               next_state <= IDLE;
59           else
60               next_state <= READ_DATA;
61           end if;
62
63       when others =>
64           null;
65
66   end case;
67 end process;
68 end structural;

```

L'architecture sovrastante, descrive in maniera comportamentale l'unità di controllo che viene descritta mediante una FSM con cinque stati distinti: IDLE, READ DATA, WAIT ACK, WAIT ELAB, e INCR COUNT. Essa è realizzata mediante due processi:

Il primo, sensibile al segnale di clock, si occupa dell'aggiornamento dello stato corrente ('current state') della FSM. Quando il segnale di reset è attivo, lo stato corrente viene impostato su IDLE, altrimenti viene aggiornato allo stato successivo ad ogni fronte di salita del clock.

Il secondo processo è combinatorio e determina il prossimo stato della macchina e controlla i segnali di output in base allo stato corrente e ad alcuni segnali di input come 'init' e 'accepted', oltre al valore del contatore 'count'. La descrizione dettagliata dell'automa è stata fornita nel paragrafo precedente.

NODO B

il nodo B è stato realizzato mediante un approccio strutturale, interconnettendo unità di controllo ed unità operativa.

¹ library IEEE;

```

2 use IEEE.std_logic_1164.ALL;
3 use work.all;
4
5 entity nodo_B is
6   port (
7     start_B : in std_logic;
8     clk_B : in std_logic;
9     rst_B : in std_logic;
10    accepted_B : out std_logic;
11    data_ready_B : in std_logic;
12    X_B : in std_logic_vector(7 downto 0)
13  );
14 end nodo_B;

```

Listing 4.4: Entity del nodo B

L'entità nodo B rappresenta una componente che interagisce attraverso vari segnali. È caratterizzata da diversi segnali di input: start_B usato per dire al nodo quando iniziare ad operare, clk_B ossia il segnale di clock, rst_B utilizzato per riportare il nodo al suo stato iniziale, accepted_B e data ready_B, segnali a supporto dell'handshake. X è un vettore di 8 bit di dati prodotto da nodo A e inviatogli.

```

1 architecture structural of nodo_B is
2   signal temp_rd : std_logic;
3   signal temp_wrt : std_logic;
4   signal temp_cnt : std_logic_vector(3 downto 0);
5   signal temp_en_ctr : std_logic;
6   signal temp_somma : std_logic;
7
8   component unitacontrollo_B is
9     port (
10       init : in std_logic;
11       clock_cu_B : in std_logic;
12       rst_cu_B : in std_logic;
13       read_B : out std_logic;
14       write_B : out std_logic;
15       count : in std_logic_vector(3 downto 0);
16       en_ctr : out std_logic;
17       accepted : out std_logic;
18       data_ready : in std_logic;
19       somma : out std_logic
20       --X : in std_logic_vector(7 downto 0);
21       --Y : in std_logic_vector(7 downto 0);

```

```

22      --Z : out std_logic_vector(7 downto 0)
23  );
24 end component;
25
26 component unita_operativa_B is
27   port(
28     clk_uo_B : in std_logic;
29     rst_uo_B : in std_logic;
30     en_ctr_uo_B : in std_logic;
31     rd_uo_B : in std_logic;
32     wrt_uo_B : in std_logic;
33     X_uo_B : in std_logic_vector(7 downto 0);
34     cnt_uo_B : out std_logic_vector(3 downto 0);
35     somma_uo_B : in std_logic
36     --Y_uo_B : out std_logic_vector(7 downto 0);
37     --Z_uo_B : out std_logic_vector(7 downto 0)
38   );
39 end component;
40
41 begin
42   uc_B : unitacontrollo_B
43     port map(
44       init => start_B,
45       clock_cu_B => clk_B,
46       rst_cu_B => rst_B,
47       read_B => temp_rd,
48       write_B => temp_wrt,
49       count => temp_cnt,
50       en_ctr => temp_en_ctr,
51       accepted => accepted_B,
52       data_ready => data_ready_B,
53       somma => temp_somma
54       --X => X_B,
55       --Y => temp_Y,
56       --Z => temp_Z
57     );
58
59   uo_B : unita_operativa_B
60     port map(
61       clk_uo_B => clk_B,
62       rst_uo_B => rst_B,

```

```

63      en_ctr_uo_B => temp_en_ctr,
64      rd_uo_B => temp_rd,
65      wrt_uo_B => temp_wrt,
66      X_uo_B => X_B,
67      cnt_uo_B => temp_cnt,
68      somma_uo_B => temp_somma
69      --Y_uo_B => temp_Y,
70      --Z_uo_B => Z_B
71  );
72
73 end structural;

```

Listing 4.5: Architecture del nodo B

L'architecture è anche in questo caso di tipo strutturale, infatti definisce i componenti che nel loro insieme danno vita al nodo B. Come è possibile evincere, sono presenti l'unità di controllo e l'unità operativa. Queste vengono definite e mappate con i segnali definiti nell'interfaccia del componente e con i segnali di appoggio definiti nella parte dichiarativa dell'architettura.

UNITÀ OPERATIVA DI B

Vediamo nel dettaglio l'unità operativa di B, ossia il componente che realizza l'operazione richiesta al sistema.

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use work.all;
5
6 entity unita_operativa_B is
7   port ( clk_uo_B : in std_logic;
8         rst_uo_B : in std_logic;
9         en_ctr_uo_B : in std_logic;
10        rd_uo_B : in std_logic;
11        wrt_uo_B : in std_logic;
12        X_uo_B : in std_logic_vector(7 downto 0);
13        cnt_uo_B : out std_logic_vector(3 downto 0);
14        somma_uo_B : in std_logic
15        --Y_uo_B : out std_logic_vector(7 downto 0);
16        --Z_uo_B : out std_logic_vector(7 downto 0)
17  );
18 end unita_operativa_B;

```

Listing 4.6: Entity UO di B

La definizione dell'entity di unita operativa B mette in luce i vari porti, che le permettono di comunicare con l'esterno. I segnali di input sono: il segnale di clock, di reset, di abilitazione per il contatore, il segnale di abilitazione alla lettura per la memoria ROM, il segnale per l'abilitazione alla scrittura della memoria ed inoltre, il dato inviatogli dal nodo A. Il segnale di output invece è unico, e rappresenta il valore del conteggio restituito dal contatore.

```

1 architecture structural of unita_operativa_B is
2     signal temp_address : std_logic_vector(3 downto 0);
3     signal temp_Y_out : std_logic_vector(7 downto 0);
4     signal temp_sum : std_logic_vector(7 downto 0) :=
5         "UUUUUUUU";
6
7     component memoria_B is
8         port (
9             address : in std_logic_vector(3 downto 0);
10            clk : in std_logic;
11            rd : in std_logic;
12            out_value : out std_logic_vector(7 downto 0)
13        );
14
15    component mem_B is
16        port (
17            address_mem_B      : in std_logic_vector(3 downto 0);
18            data              : in std_logic_vector(7 downto 0);
19            clk_mem_B         : in std_logic;
20            wrt               : in std_logic
21            --out_value_mem_B : out std_logic_vector(7 downto 0)
22        );
23
24    component counter_mod_16 is
25        port (
26            clk_ctr, rst_ctr, en_ctr : in std_logic;
27            curr_value : out std_logic_vector(3 downto 0)
28        );
29
30    end component;
31
32 begin
33     rom_B : memoria_B
34         port map( address => temp_address,

```

```

35          clk => clk_uo_B,
36          rd => rd_uo_B,
37          out_value => temp_Y_out
38      );
39
40      ram_B : mem_B
41      port map( address_mem_B => temp_address,
42                  data => temp_sum,
43                  clk_mem_B => clk_uo_B,
44                  wrt => wrt_uo_B
45                  --out_value_mem_B => Z_uo_B
46
47  );
48
49      ctr : counter_mod_16
50      port map( clk_ctr => clk_uo_B,
51                  rst_ctr => rst_uo_B,
52                  en_ctr => en_ctr_uo_B,
53                  curr_value => temp_address
54  );
55
56      cnt_uo_B <= temp_address;
57      --Y_uo_B <= temp_Y_out;
58      temp_sum <= std_logic_vector(unsigned(temp_Y_out) +
59          unsigned(X_uo_B)) when (somma_uo_B = '1');
60      --Z_uo_B <= temp_sum;
61 end structural;

```

Listing 4.7: Entity UO B

Per quanto riguarda l'architecture, essa è di tipo strutturale dato che collega insieme i vari componenti: ROM, counter mod 16 (per la descrizione dei componenti si veda il capitolo 2, ed in particolare l'esercizio 6.1) e memoria, quest'ultima è una memoria nella quale, all'indirizzo scandito dal contatore salveremo il risultato dell'elaborazione, ossia la somma tra il valore in ingresso all'unità operativa X e il contenuto della locazione di memoria ROM scandita dal contatore, Y. Il comportamento implementato è il seguente: il contatore sulla base dei segnali di abilitazione impartiti dall'unità di controllo scandisce ad ogni colpo di clock una locazione diversa di memoria ROM. Quest'ultima dovrà naturalmente essere abilitata da parte dell'unità di controllo tramite il segnale di read, al fine di leggere il contenuto della locazione selezionato. Una volta ricavato il contenuto della memoria, se il segnale

somma, impartito dalla control unit sarà alto, potremmo procedere al calcolo della somma di X e Y. Quest'ultima dunque, sarà implementata in maniera comportamentale piuttosto che con l'aggiunta di un addizionatore. **UNITA DI CONTROLLO DI B**

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use IEEE.numeric_std.ALL;
4 use work.all;
5
6 entity unitacontrollo_B is
7     port(    init : in std_logic;
8             clock_cu_B : in std_logic;
9             rst_cu_B : in std_logic;
10
11             -- ROM B.
12             read_B : out std_logic;
13
14             -- MEM B.
15             write_B : out std_logic;
16
17             -- CONTATORE.
18             count : in std_logic_vector(3 downto 0);
19             en_ctr : out std_logic;
20
21             --HANDSHAKE
22             accepted : out std_logic;
23             data_ready : in std_logic;
24
25             --SOMMA
26             somma : out std_logic
27             --X : in std_logic_vector(7 downto 0);
28             --Y : in std_logic_vector(7 downto 0);
29             --Z : out std_logic_vector(7 downto 0)
30         );
31 end unitacontrollo_B;
```

Listing 4.8: Entity CU B

L'entità rappresenta il cuore del sistema. I segnali di entrata che la caratterizzano sono: il segnale di inizializzazione, il clock, il reset, il count ossia il valore del conteggio che viene restituito dal contatore dell'unità operativa, il segnale accepted

per la gestione del protocollo di handshake ed infine un segnale per abilitare alla scrittura della memoria. Per quanto riguarda i segnali di uscita invece, essi sono: il segnale di lettura, di abilitazione del contatore ed il segnale data ready per la realizzazione del protocollo di handshake.

```

1   architecture structural of unitacontrollo_B is
2     type state is(IDLE, WAIT_DATA_READY, ACK, ELAB, WAIT_FINE,
3       INCR_COUNT);
4     signal current_state, next_state : state;
5
6
7     reg_stato: process(clock_cu_B)
8     begin
9       if(clock_cu_B'event and clock_cu_B='1') then
10      if(rst_cu_B='1') then
11        current_state <=IDLE;
12      else
13        current_state <=next_state;
14      end if;
15    end if;
16  end process;
17
18  comb: process(current_state, init, data_ready, count)
19  begin
20
21    case current_state is
22      when IDLE =>
23        read_B <= '0';
24        write_B <= '0';
25        en_ctrl <= '0';
26        accepted <= '0';
27        somma <= '0';
28        if(init = '1') then
29          next_state <= WAIT_DATA_READY;
30        else
31          next_state <= IDLE;
32        end if;
33
34      when WAIT_DATA_READY =>
35        en_ctrl <= '0';
36        read_B <= '1';

```

```

37          if(data_ready = '1') then
38              next_state <= ACK;
39          else
40              next_state <= WAIT_DATA_READY;
41          end if;
42
43      when ACK =>
44          accepted <= '1';
45          somma <= '1';
46          next_state <= ELAB;
47
48
49      when ELAB =>
50          write_B <= '1';
51          next_state <= WAIT_FINE;
52
53      when WAIT_FINE =>
54          if(data_ready = '0') then
55              somma <= '0';
56              accepted <= '0';
57              read_B <= '0';
58              next_state <= INCR_COUNT;
59          else
60              next_state <= WAIT_FINE;
61          end if;
62
63
64      when INCR_COUNT =>
65          write_B <= '0';
66          en_ctrl <= '1';
67
68          if(count = "1111") then
69              next_state <= IDLE;
70          else
71              next_state <= WAIT_DATA_READY;
72          end if;
73
74      when others =>
75          null;
76
77  end case;

```

```

78      end process;
79  end structural;
```

Listing 4.9: Architecture CU B

L'architecture sovrastante, descrive in maniera comportamentale l'unità di controllo con sei stati distinti: IDLE, WAIT DATA READY, ACK, ELAB, WAIT FINE e INCR COUNT. Questa FSM gestisce un processo di elaborazione dati, con funzionalità di lettura, riconoscimento dell'avvenuta ricezione dei dati, elaborazione, e infine incremento del contatore. È caratterizzato da due processi, il primo è sensibile al segnale di clock, questo processo aggiorna lo stato corrente (current state) della FSM. Se il segnale di reset è attivo, lo stato corrente si reimposta su IDLE. Altrimenti, ad ogni fronte di salita del clock, lo stato corrente passa a quello successivo. Il secondo processo è di tipo combinatorio che definisce la logica di transizione tra gli stati e il controllo dei segnali di output in base allo stato corrente, a init, data ready, e al contatore count. Il comportamento dell'automa a stati finiti è descritto in maniera dettagliata nel paragrafo precedente.

SISTEMA

Il sistema seguente, è ottenuto mediante il collegamento opportuno dei due nodi precedentemente visti, ed è quindi realizzato mediante una architettura di tipo strutturale.

```

1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 use work.all;
4
5 entity system is
6   port( start : in std_logic;
7         clock : in std_logic;
8         reset : in std_logic
9         --data_out : out std_logic_vector(7 downto 0)
10        );
11 end system;
12
13 architecture structural of system is
14
15   signal accepted : std_logic;
16   signal data_ready : std_logic;
17   signal temp_out : std_logic_vector(7 downto 0);
18
19   component nodo_A is
20     port(
```

```

21         start_A : in std_logic;
22         clk_A : in std_logic;
23         rst_A : in std_logic;
24         accepted_A : in std_logic;
25         data_ready_A : out std_logic;
26         X : out std_logic_vector(7 downto 0)
27     );
28 end component;
29
30 component nodo_B is
31     port(
32         start_B : in std_logic;
33         clk_B : in std_logic;
34         rst_B : in std_logic;
35         accepted_B : out std_logic;
36         data_ready_B : in std_logic;
37         X_B : in std_logic_vector(7 downto 0)
38         --Z_B : out std_logic_vector(7 downto 0)
39     );
40 end component;
41
42 begin
43     A : nodo_A
44         port map( start_A => start,
45                     clk_A => clock,
46                     rst_A => reset,
47                     accepted_A => accepted,
48                     data_ready_A => data_ready,
49                     X => temp_out
50     );
51
52     B : nodo_B
53         port map( start_B => start,
54                     clk_B => clock,
55                     rst_B => reset,
56                     accepted_B => accepted,
57                     data_ready_B => data_ready,
58                     X_B => temp_out
59                     --Z_B => data_out
60     );

```

```
61 end structural;
```

Listing 4.10: sistema totale

4.2.2 (

Simulazione) Per effettuare la simulazione del componente è stato realizzato il testbench

```
1  library IEEE;
2  use IEEE.std_logic_1164.ALL;
3  use work.all;
4
5  entity tb_system is
6  end tb_system;
7
8  architecture testbench of tb_system is
9
10   signal start_tb : std_logic := '0';
11   signal clock_tb : std_logic := '0';
12   signal reset_tb : std_logic := '0';
13   --signal data_out_tb : std_logic_vector(7 downto 0);
14
15   constant clock_period : time := 10 ns; -- Adjust as needed
16
17   component system
18     port (
19       start : in std_logic;
20       clock : in std_logic;
21       reset : in std_logic
22       --data_out : out std_logic_vector(7 downto 0)
23     );
24   end component;
25
26   begin
27     uut_system : system
28     port map(
29       start => start_tb,
30       clock => clock_tb,
31       reset => reset_tb
32     );
33
34   -- Clock process
```

```

35      process
36          begin
37              while now < 1000 ns -- Simulation time, adjust as
38                  needed
39                      loop
40                          clock_tb <= not clock_tb;
41                          wait for clock_period / 2;
42                      end loop;
43
44
45          -- Stimulus process
46          process
47          begin
48              reset_tb <= '1';
49              wait for 10 ns;
50              reset_tb <= '0';
51              wait for 10 ns;
52              start_tb <= '1';
53              wait for 500 ns;
54              start_tb <= '0';
55              wait for 100 ns;
56              reset_tb <= '1';
57              wait for 10 ns;
58              wait;
59          end process;
60      end architecture;

```

Listing 4.11: Testbench del sistema

Questo testbench VHDL è progettato per simulare e verificare il comportamento di un'entità VHDL chiamata sistema. Il testbench non include l'implementazione di questo componente ma definisce come dovrebbe essere stimolata e testata all'interno dell'ambiente di simulazione. In primo luogo, ‘e stata dichiarata una entity tb system vuota dato che non rappresenta un oggetto fisico da realizzare, ma anzi è necessaria solo al fine di condurre i test. L'architettura del testbench definisce i segnali interni usati per stimolare l'entità sotto test:

- clock: Un segnale di clock utilizzato per sincronizzare l'entità.
- reset: Un segnale di reset per inizializzare o resettare l'entità.
- start: Un segnale utilizzato per indicare l'inizio di un processo nell'entità.

L'entità è istanziata con il nome UUT SYSTEM (Unit Under Test) e i suoi porti sono mappati ai segnali definiti nel testbench. Il processo di generazione del clock, genera un segnale di clock che oscilla con un periodo definito dalla costante `clockperiod`. Sono applicati poi degli stimoli per verificare il funzionamento del sistema. Di seguito sono riportati i risultati dei test condotti.

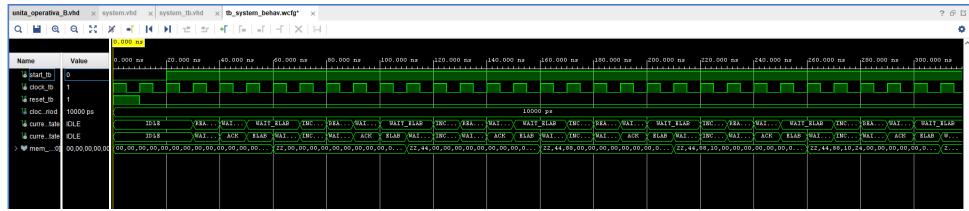


Figure 4.4: Risultato della simulazione del tb.

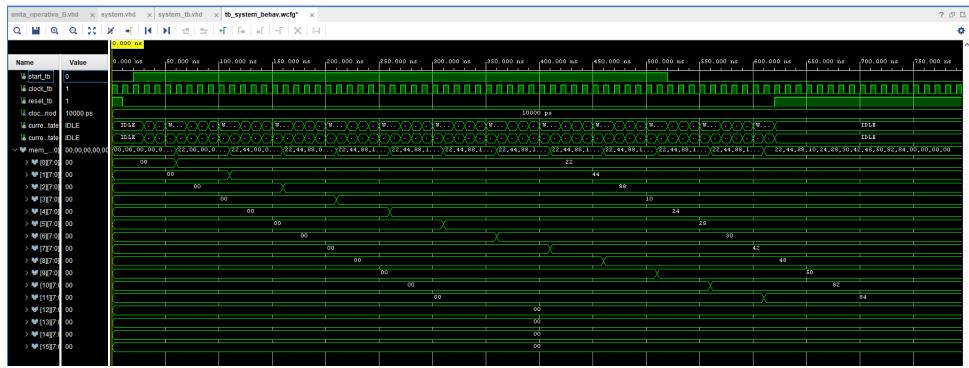


Figure 4.5: Risultato della simulazione del tb (dettaglio).

CHAPTER 5

PROCESSORE MIC-1

5.1 Esercizio 9 - Processore

A partire dall'implementazione fornita del processore operante secondo il modello IJVM,

- si proceda all'analisi dell'architettura mediante simulazione e si approfondisca lo studio del suo funzionamento per due istruzioni a scelta,
- si modifichi un codice operativo a scelta, documentando tutte le modifiche effettuate

5.1.1 Analisi dell'architettura del MIC-1

Il processore MIC-1 nasce come interprete hardware del bytecode generato da un programma Java, in particolare della **IJVM**, ovvero una sezione della macchina virtuale che tratta solo numeri interi. Utilizza una logica **micropogrammata** in cui i segnali dell'unità di controllo sono memorizzati in una micro-ROM.

Il processore utilizza è di tipo *RISC* (Reduced Instruction Set Computer) ed utilizza un'architettura a Stack, la quale implica che gli operandi da utilizzare, per esempio durante l'esecuzione di un'espressione aritmetica, sono implicitamente memorizzati in una specifica area di memoria: questo riduce la complessità delle Control Word, ovvero delle microistruzioni, poiché permette di avere una gestione delle istruzioni che non richiede di esplicitare gli operandi. A tal proposito, abbiamo deciso di approfondire i codici operativi delle istruzioni **IADD** e **BIPUSH**, modificando inoltre il codice operativo dell'istruzione **ISUB**.

Descrizione architettura

Datapath Sono presenti 10 registri e due bus tramite è possibile scambiare i dati; oltre a questi sono presenti un'ALU che consente di fare operazioni logico/aritmetiche di base, e uno Shift Register.

I registri possono scrivere sul bus B, mentre il bus C può scrivergli sopra: In particolare, attraverso il bus C è possibile scrivere in più registri contemporaneamente; invece, tramite il bus B si può leggere il contenuto di un solo registro alla volta, pertanto l'accesso è in mutua esclusione.

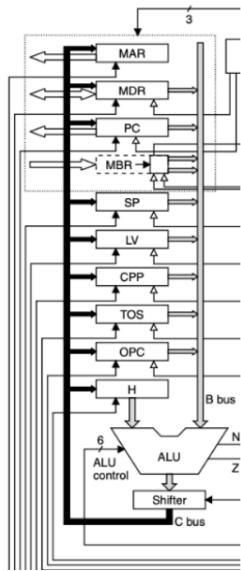


Figure 5.1: Datapath del MIC-1

Vediamo i vari registri:

- I registri di comunicazione con la memoria sono il *MAR* (*Memory Address Register*), l'*MDR* (*Memory Data Register*), il *PC* (*Program Counter*) e l'*MBR* (*Memory Byte Register*). La prima coppia opera sulla lettura e scrittura dei dati, mentre la seconda sulla lettura delle istruzioni;
- Lo **SP** (*Stack Pointer*) contiene l'indirizzo in testa allo stack;
- Il *TOS* (*Top of Stack*) contiene il valore corrispondente a tale indirizzo;
- I registri *H* (*Holding*) e *OPC* sono di appoggio: il primo permette di memorizzare temporaneamente un operando dell'ALU, mentre il secondo è un registro temporaneo usato dalle istruzioni;
- Infine, i registri *CPP* e *LV* sono usati per la gestione dello stack.

La memoria è organizzata in parole da 8 bit, ognuna identificata da un indirizzo univoco a 32 bit. L'indirizzamento tramite MAR è diverso dall' indirizzamento tramite PC: infatti, il MAR legge i successivi 4 byte a partire dall' indirizzo specificato, mentre il PC legge 1 byte alla volta.

ALU L'ALU (Arithmetic Logic Unit) del processore Mic-1 ha due ingressi A e B. L'ingresso A ‘e collegato solo al registro H, mentre B al bus omonimo. Essa presenta in ingresso sei linee di controllo, che consentono di selezionare l'operazione da realizzare.

F₀	F₁	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	\bar{A}
1	0	1	1	0	0	B
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B - A
1	1	0	1	1	0	B - 1
1	1	1	0	1	1	-A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	-1

Figure 5.2: Tabella dei segnali della ALU

Vediamo nello specifico:

- F_0 e F_1 rappresentano l'operazione da eseguire;
- ENA e ENB abilitano gli ingressi dei due operandi;
- $INVA$ abilita l'inversione di A;
- INC abilita l'incremento di 1;

Dopo che viene calcolato il risultato dall'ALU, vengono aggiunti due segnali aggiuntivi per modificare il risultato prima di inviarlo sul bus C. Questi segnali sono chiamati *SLL8* e *SRA1*. *SLL8* esegue uno spostamento di 8 bit verso sinistra, mentre *SRA1* esegue uno spostamento di 1 bit verso destra, utilizzando rispettivamente uno shift logico e uno shift aritmetico.

Inoltre, l'ALU produce due ulteriori flag chiamati N e Z. Il flag N viene utilizzato per indicare se il risultato è negativo, mentre il flag Z indica se il risultato è nullo (ossia uguale a zero).

Control Path Le istruzioni sono memorizzate all'interno di una micro-ROM formata da 512 control word da 36 bit.

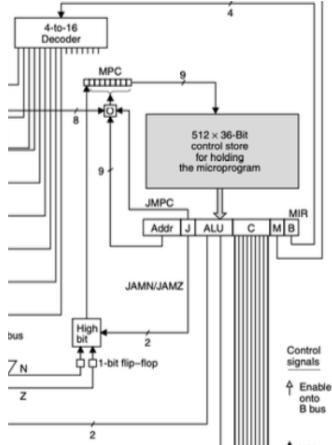


Figure 5.3: Control Path

Anche la memoria di controllo necessita di un PC, chiamato **MPC** (*Micro Program Counter*) invece nel registro **MIR** (*Micro Instruction Register*) è memorizzata la microistruzione in esecuzione, i cui bit determinano i segnali di controllo che guidano il datapath. Infine, esso include un Decoder 4:16 per fornire in uscita i corretti segnali di abilitazione per il bus B.

Se le operazioni che dobbiamo effettuare sono in sequenza, il campo Next Address indicherà la successiva istruzione da eseguire: tale campo fornisce quindi lo stato prossimo, non serve quindi un contatore per incrementare di volta in volta il PC, perché per prendere l'istruzione successiva sarà sufficiente utilizzare il valore contenuto nel Next Address.

Quando è necessario fare un salto il valore del campo Next Address viene opportunamente modificato prima di essere inserito nel PC, andando a effettuare un controllo su alcuni bit nella control word *JAM*. Vediamo i vari casi:

- Se il campo *JAM* vale 000, allora MPC sarà posto proprio al valore del Next Address presente nella control word stessa senza modifiche;
- Se *JAMN* è alto, si calcola l'OR tra il bit più significativo e il flag N, se ne pone il risultato nel bit più significativo di MPC;
- Se *JAMN* e *JAMZ* sono entrambi alti, si calcola l'OR rispetto a entrambi i flag;
- Se è alto il bit *JMPC* si effettua l'OR tra gli 8 bit di *MBR* e gli 8 bit meno significativi di next address e il risultato viene posto in MPC.

Control Word Come abbiamo visto la logica micropogrammata prevede la definizione di una control-word; sfruttando questa siamo infatti in grado di realizzare dei micro-

programmi e quindi i codici operativi che risiedono nella control-store.

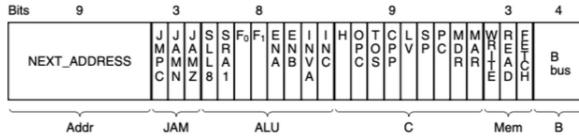


Figure 5.4: Control Word

La control word è composta da 36 bit:

- 9 bit per gestire il next address;
- 3 bit di salto;
- 8 bit per il controllo dell'ALU;
- 3 bit per l'interfacciamento con la memoria;
- 13 bit per gestire la scrittura e la lettura dai registri interni. Di questi, 9 bit sono dedicati alla scrittura, mentre 4 bit sono destinati alla lettura. Il bus B è utilizzato esclusivamente per la scrittura, quindi solo un registro alla volta può caricare i suoi dati su questo bus, garantendo la mutua esclusione. Tuttavia, per la lettura, è possibile che più registri siano abilitati contemporaneamente, consentendo al contenuto del bus C di essere scritto su più registri contemporaneamente. Nonostante l'indirizzamento del bus B richieda 9 bit, è stato scelto di complicare l'hardware aggiungendo un decoder e indirizzando il bus con soli 4 bit. Questo è stato fatto per ridurre la dimensione della control word;

5.1.2 Analisi e simulazione di due istruzioni a scelta

Nella documentazione viene fornito un programma scritto in linguaggio *MAL* (*Micro Assembly Language*), che sfrutteremo per effettuare la simulazione. In sostanza, il programma esegue due push sullo Stack dei valori in esadecimale 0xA e 0xE, per poi effettuare una somma e salvarla nella variabile 'a':

```
.main
.var
a
.endvar
BIPUSH 0xA
BIPUSH 0xE
```

```

IADD
ISTORE a
HALT
.endmethod

```

Ad ogni istruzione corrisponde un codice operativo memorizzato all'interno della RAM e a questo codice operativo corrisponde un indirizzo presente all'interno del *control store*, che indica la posizione della prima microistruzione corrispondente al codice operativo dell'istruzione.

Istruzione BIPUSH

Il microcodice corrispondente alla BIPUSH è il seguente:

```

bipush = 0x10:
SP = MAR = SP + 1
PC = PC + 1; fetch
MDR = TOS = MBR; wr; goto main

```

La BIPUSH (codice operativo 0x10) si occupa di caricare in testa allo stack un intero senza segno codificato su un byte. La prima microistruzione incrementa di 1 lo *Stack Pointer* e lo salva all'interno del *MAR*, al primo colpo di clock, in modo tale da indicare la locazione in cui verrà salvato il valore 0xA (e alla prossima BIPUSH 0xE). Al secondo colpo clock viene incrementato il *Program Counter* in modo tale da prelevare il prossimo byte dell'istruzione (dunque l'operando 0xA), per poi ordinare il *fetch*. All'ultimo colpo di clock viene posto il contenuto di *MBR* all'interno di *TOS* e all'interno di *MDR*, il quale in coppia con il *MAR* si occuperà della scrittura in memoria, infatti, viene ordinato il *write*. Infine *goto main* ci riporta al main.

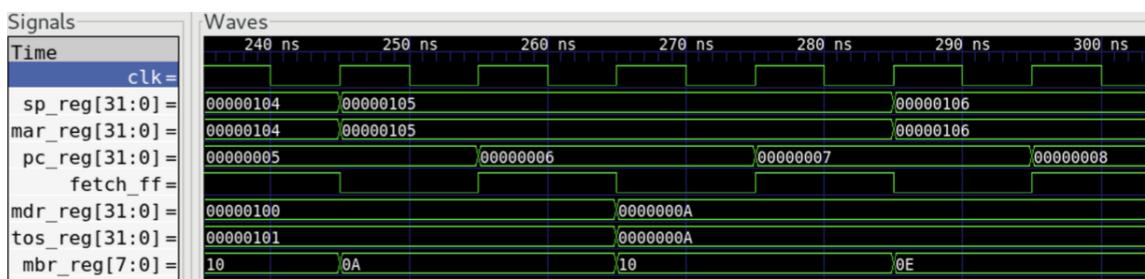


Figure 5.5: Simulazione della BIPUSH.

Istruzione IADD.

Il microcodice corrispondente alla IADD è il seguente:

```

iadd = 0x65:
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR + H; wr; goto main

```

La IADD (codice operativo 0x65) si occupa di effettuare il pop dei due elementi in testa allo stack, effettuarne la somma, per poi eseguire un push sullo stack. La prima microistruzione decremente di 1 lo *Stack Pointer* e ne assegna il valore al *MAR*, per poi ordinare la *read*, questo salverà 0xA all'interno dell'*MDR* al successivo colpo di clock. Al secondo colpo clock viene salvato l'elemento presente all'interno di *TOS* nel registro di *Holding* (0xE). All'ultimo colpo di clock vengono sommati i contenuti di *MDR* ed *H* ($0xA + 0xE = 0x18$), e posti sia nel *TOS*, sia nell'*MDR*, per effettuare la *write* nello stack. La locazione dov'era presente 0xE non è più presa in considerazione. Infine *goto main* ci riporta al main.

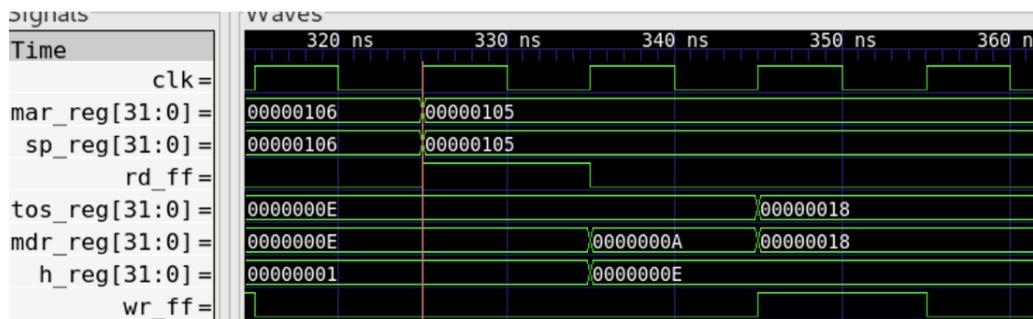


Figure 5.6: Simulazione della IADD.

Modifica il codice operativo di ISUB.

La ISUB (codice operativo 0x5C) esegue le stesse identiche operazioni della IADD, solo che all'ultimo colpo di clock esegue una sottrazione del contenuto dell'*MDR* e del contenuto di *H*.

```

isub = 0x5C:
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR - H; wr; goto main

```

La ISUB è stata modificata affinché si trasformasse in una addizione:

```

isub = 0x5C:
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR + H; wr; goto main

```

Infatti il risultato è ancora una volta 0x18:

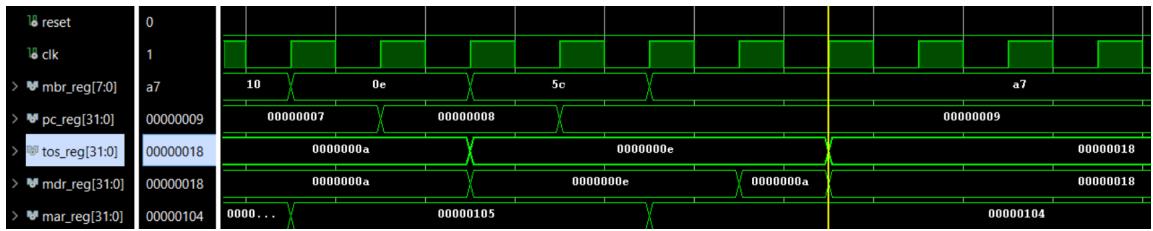


Figure 5.7: Simulazione della ISUB modificata.

CHAPTER 6

INTERFACCIA SERIALE

La **comunicazione seriale** prevede che le informazioni scambiate tra due entità, vengano scambiate un bit alla volta. La **UART** è un dispositivo hardware che supporta la comunicazione seriale asincrona, caratterizzata da due blocchi funzionali: **trasmettitore** e **ricevitore**. Comunemente impiegata nelle porte seriali di PC e sistemi embedded, la UART opera attraverso segnali di trasmissione (TXD) e ricezione (RXD), che formano il canale di trasmissione/ricezione vero e proprio. Quest'ultima è caratterizzata anche da ulteriori segnali di ingresso e di uscita come DBIN e DBOUT che sono segnali ad 8 bit rispettivamente di ingresso e di uscita. Le UART possono generare interruzioni per evitare che il microcontrollore sia bloccato in attesa durante la trasmissione o ricezione dei dati. Le interruzioni possono indicare la ricezione di nuovi dati (RDA) o la disponibilità del buffer di trasmissione (TBE). I possibili errori includono overrun (il dispositivo ricevente non elabora i dati abbastanza rapidamente), errori di framing (mancanza di stop bit) ed errori di parità (mismatch nel bit di parità). Questi sono rispettivamente gestiti mediante dei flag presenti nel blocco funzionale del ricevitore: OE, PE, FE.

6.1 Esercizio 10 - Comunicazione seriale

Partendo dall'implementazione fornita dalla Digilent di un dispositivo UART-RS232 (componente RS232RefComp.vhd), progettare, implementare e simulare in VHDL un sistema composto da 2 unità A e B che condividono lo stesso segnale di clock e comunicano tra loro mediante interfaccia seriale. Il sistema A contiene una ROM di 8 locazioni da 1 byte ciascuno, un contatore CONT_A per scandire le locazioni della ROM e una UART_A, mentre il sistema B contiene una memoria MEM di 8 locazioni da 1 byte ciascuno, un contatore CONT_B per scandire le locazioni della MEM e una UART_B. Quando un segnale WR viene asserito nell'unità A, viene prelevato un byte dalla ROM e inviato all'unità B, che dovrà riceverlo e salvarlo in MEM.

6.2 Progetto e architettura

Il progetto del componente è basato su due unità funzionali unità A e unità B che implementano la comunicazione seriale tramite l'impiego del componente UART. L'architettura del componente è mostrata di seguito:

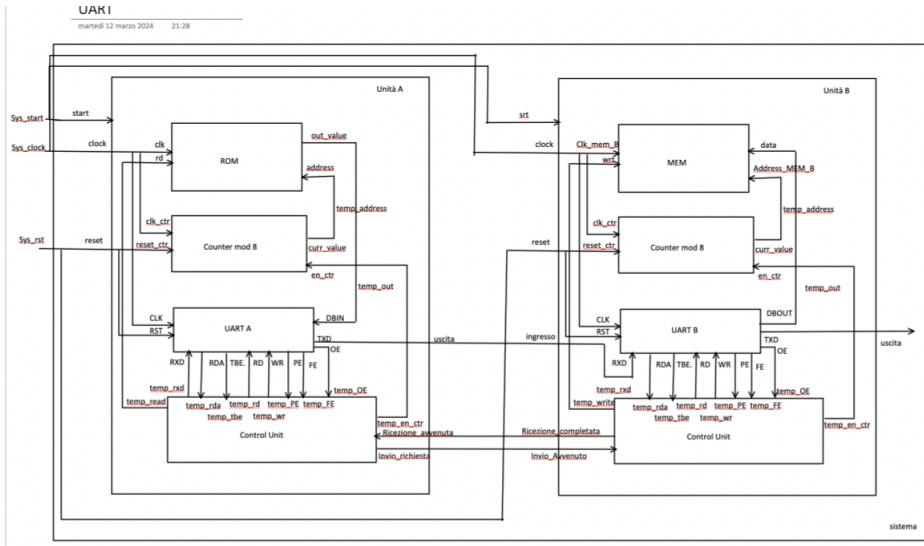


Figure 6.1: Schema del componente.

Come è possibile notare l'**unità A** è composta da 4 sottocomponenti: una **ROM** a cui accederemo, per prelevare un dato, un **contatore** per scandire gli indirizzi di memoria e la **UART** per implementare la comunicazione seriale con l'unità B e la **Control Unit**. Per quanto riguarda l'automa a stati finiti che implementa la control unit dell'unità A, è riportata di seguito:

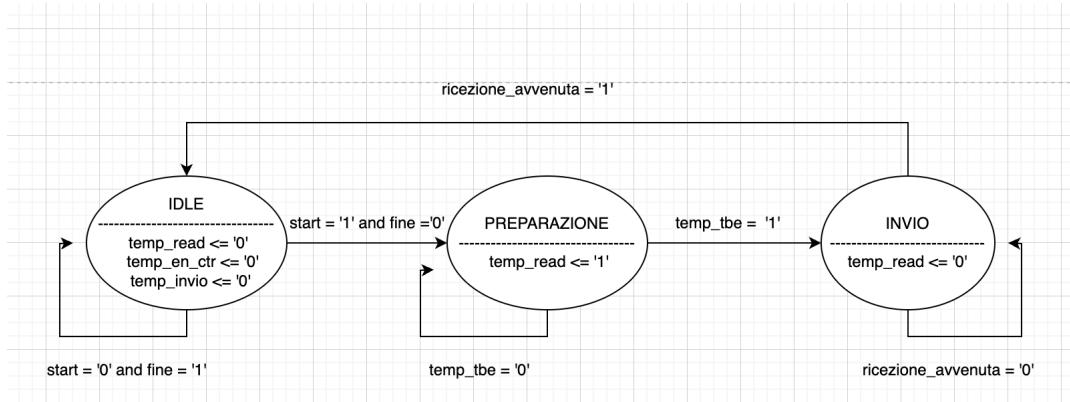


Figure 6.2: ASF della CU dell'unità A.

Questo ASF comprende tre stati: **IDLE**, **PREPARAZIONE**, **INVIO**. È possibile analizzare gli stati nello specifico:

- **IDLE**: In tale stato l'automa imposta `temp_read`, `temp_en_ctr` e `temp_invio` a '0'. Se `start = '1'` e `fine = '0'`, transita allo stato **PREPARAZIONE** e imposta `temp_write` a '1'. In caso contrario, rimane nello stato **IDLE**.
- **PREPARAZIONE**: In questo stato, `temp_read` viene impostato a '1'. Se `temp_tbe = '1'`, l'automa transita allo stato **INVIO**, impostando `temp_write` a

'0' e temp_invio a '1'. Se temp_tbe non è '1', l'automa rimane nello stato **PREPARAZIONE**.

- **INVIO:** Quando l'automa si trova nello stato **INVIO**, temp_read viene impostato a '0'. Se ricezione_avvenuta = '1', allora temp_en_ctr viene impostato a '1'. Se inoltre temp_address = "111", fine viene impostato a '1' e l'automa transita allo stato **IDLE**. Se ricezione_avvenuta non è '1', l'automa rimane nello stato **INVIO**.

Per quanto riguarda l'unità B invece, è caratterizzata anch'essa da 4 componenti: una **memoria** in cui scriveremo il dato in una specifica locazione, un **contatore** per scandire gli indirizzi di **memoria**, l'**UART** per scandire la comunicazione seriale. Anche in questo caso l'unità di controllo è realizzata mediante un ASF, riportato di seguito:

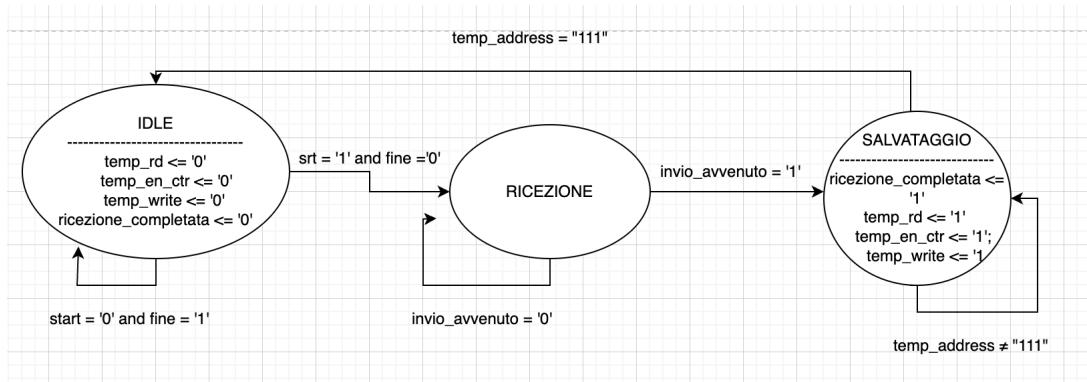


Figure 6.3: ASF della CU dell'unità B.

L'ASF è caratterizzato da 3 stati: **IDLE**, **RICEZIONE**, e **SALVATAGGIO**. Vediamone una descrizione più dettagliata di ciascuno stato:

- **IDLE:** Nello stato **IDLE**, tutte le variabili temporanee temp_rd, temp_en_ctr, temp_write e ricezione_completata vengono impostate a '0'. Se il segnale srt è '1' e fine è '0', l'automa transita allo stato **RICEZIONE**. Altrimenti, rimane nello stato **IDLE**.
- **RICEZIONE:** In questo stato, l'automa attende che invio_avvenuto sia '1'. Se invio_avvenuto è '1' e temp_rda è anche '1', l'automa transita allo stato **SALVATAGGIO**. In caso contrario, rimane nello stato **RICEZIONE** fino a quando queste condizioni non vengono soddisfatte.
- **SALVATAGGIO:** Una volta entrati nello stato **SALVATAGGIO**, ricezione_completata viene impostata a '1', e temp_rd, temp_en_ctr, temp_write vengono impostati tutti a '1', indicando che il processo di ricezione e salvataggio è completato.

Se temp_address è "111", fine viene impostato a '0' e l'automa ritorna allo stato **IDLE**. In caso contrario, l'automa ritorna direttamente allo stato **IDLE** senza modificare il valore di fine.

6.2.1 Implementazione

Unità A

```

1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.NUMERIC_STD.ALL;
4   use work.all;
5
6   entity unita_a is
7       port(
8           start : in std_logic;
9           clock, reset : in std_logic;
10          uscita : out std_logic;
11          invio_richiesta : out std_logic;
12          ricezione_avvenuta : in std_logic
13      );
14 end unita_a;
```

Listing 6.1: Implementazione entity dell'Unità A

Il codice definisce una "entity" chiamata **unita_a** che ha varie porte di input e output per comunicare con altri componenti. In particolare ha un segnale di **start**, un segnale di **clock**, uno di **reset**, uno di **uscita** ed infine due segnali per gestire invio e ricezione del bit mediante la comunicazione seriale implementata dalla UART, che sono rispettivamente: **invio_richiesta** e **ricezione_avvenuta**. La struttura e il comportamento di **unita_a** sono definiti nell'architettura structural.

```

1 architecture structural of unita_a is
2
3     component memoria_A is
4         port(
5             address : in std_logic_vector(2 downto 0);
6             clk : in std_logic;
7             rd : in std_logic;
8             out_value : out std_logic_vector(7 downto 0)
9         );
10    end component;
```

```

12      component counter_mod_8 is
13      port (
14          clk_ctr, rst_ctr, en_ctr : in std_logic;
15          curr_value : out std_logic_vector(2 downto 0)
16      );
17  end component;
18
19  component Rs232RefComp is
20      Port (
21          TXD    : out std_logic    := '1';
22          RXD    : in  std_logic;
23          CLK    : in  std_logic;
24          DBIN   : in  std_logic_vector (7 downto 0);
25          DBOUT  : out std_logic_vector (7 downto 0);
26          RDA    : inout std_logic;
27          TBE    : inout std_logic    := '1';
28          RD     : in  std_logic;
29          WR     : in  std_logic;
30          PE     : out std_logic;
31          FE     : out std_logic;
32          OE     : out std_logic;
33          RST    : in  std_logic := '0');
34  end component;
35
36  signal temp_address : std_logic_vector(2 downto 0);
37  signal fine : std_logic := '0';
38  signal temp_invio : std_logic := '0';
39  signal temp_en_ctr : std_logic;
40  signal temp_out : std_logic_vector(7 downto 0);
41  signal temp_read : std_logic;
42  signal temp_tbe : std_logic;
43  signal temp_write : std_logic;
44  signal temp_rxd : std_logic;
45  signal temp_DBOUT : std_logic_vector(7 downto 0);
46  signal temp_RDA : std_logic;
47  signal temp_RD : std_logic;
48  signal temp_PE : std_logic;
49  signal temp_FE : std_logic;
50  signal temp_OE : std_logic;
51  type state is(IDLE, PREPARAZIONE, INVIO);
52  signal current_state, next_state : state;

```

```

53
54
55 begin
56     cntr : counter_mod_8
57         port map(    clk_ctr => clock,
58                     rst_ctr => reset,
59                     en_ctr => temp_en_ctr,
60                     curr_value => temp_address
61             );
62
63     ROM : memoria_A
64         port map(    address => temp_address,
65                     clk => clock,
66                     rd => temp_read,
67                     out_value => temp_out
68             );
69
70     UART : Rs232RefComp
71         port map(    TXD => uscita,
72                     RXD => temp_rxd,
73                     CLK => clock,
74                     DBIN => temp_out,
75                     DBOUT => temp_DBOUT,
76                     RDA => temp_RDA,
77                     TBE => temp_tbe,
78                     RD => temp_RD,
79                     WR => temp_write,
80                     PE => temp_PE,
81                     FE => temp_FE,
82                     OE => temp_OE,
83                     RST => reset
84             );
85
86     invio_richiesta <= temp_invio;
87
88     reg_stato: process(clock)
89         begin
90             if(clock'event and clock='1') then
91                 if(reset = '1') then
92                     current_state <= IDLE;
93                 else

```

```

94                     current_state <=next_state;
95             end if;
96         end if;
97     end process;
98
99     comb: process(current_state, start,
100           ricezione_avvenuta, temp_tbe, fine)
101
102     begin
103
104         case current_state is
105             when IDLE =>
106                 temp_read <= '0';
107                 temp_en_ctrl <= '0';
108                 temp_invio <= '0';
109                 if(start = '1' and fine = '0') then
110                     temp_write <= '1';
111                     next_state <= PREPARAZIONE;
112                 else
113                     temp_write <= '0';
114                     next_state <= IDLE;
115             end if;
116
117             when PREPARAZIONE =>
118                 temp_read <= '1';
119                 if(temp_tbe = '1') then
120                     temp_write <= '0';
121                     temp_invio <= '1';
122                     next_state <= INVIO;
123                 else
124                     next_state <= PREPARAZIONE;
125             end if;
126
127             when INVIO =>
128                 temp_read <= '0';
129
130                 if(ricezione_avvenuta = '1') then
131                     temp_en_ctrl <= '1';
132                     if(temp_address = "111") then
133                         fine <= '1';
134                         next_state <= IDLE;
135                     else

```

```

134                     next_state <= IDLE;
135             end if;
136         else
137             next_state <= INVIO;
138         end if;
139
140     when others =>
141         null;
142
143     end case;
144 end process;
145
146 end structural;

```

Listing 6.2: Implementazione dell'Unità A

Il componente è realizzato mediante un approccio strutturale ed i componenti coinvolti sono i seguenti:

- **Memoria_A:** Si tratta di un componente di memoria(ROM) con indirizzo, clock, e segnali di lettura. Conserva dati di 8 bit e gli indirizzi sono di 3 bit, permettendo l'accesso a 8 posizioni diverse. L'accesso alle locazioni sarà scandito tramite un contatore.
- **Counter_mod_8:** È un contatore modulo 8 con segnali di clock, reset ed enable. Il suo valore attuale è rappresentato da un vettore di 3 bit, che viene posto in ingresso alla memoria ROM al fine di scadirne gli indirizzi.
- **Rs232RefComp:** Il componennte gestisce la comunicazione seriale seguendo lo standard RS-232. Ha vari segnali per la trasmissione e la ricezione dei dati, oltre ad altri per il controllo dello stato della comunicazione.

Per quanto riguarda l'unità di controllo, è realizzata mediante un ASF, i segnali che quest'ultima setterà per muovere il datapath sono numerosi e saranno utili per gestire i segnali in ingresso e in uscita alla UART ed inoltre rappresenteranno i segnali di controllo per gli ulteriori componenti. Questi segnali temporanei saranno utilizzati per connettere i componenti interni e gestire lo stato e il flusso dei dati all'interno del componente. Per l'implementazione della **control unit**, è stato realizzato un ASF, caratterizzato da 3 stati: **IDLE**, **PREPARAZIONE**, e **INVIO**. Il comportamento di `unita_a` cambia in base allo stato corrente e alle condizioni esterne di input (automa di Mealy), l'automa è stato dettagliatamente descritto nel sottoparagrafo precedente.

Unità B

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use work.all;
5
6 entity unita_b is
7 port(
8     ingresso : in std_logic;
9     srt : in std_logic;
10    ck, rst : in std_logic;
11    invio_avvenuto : in std_logic;
12    ricezione_completata : out std_logic
13 );
14 end unita_b;

```

Listing 6.3: Implementazione entity dell’Unità B

Il codice descrive unità, denominata `unita_b`, che interagisce con segnali esterni e componenti interni per svolgere delle operazioni. Questa unità è focalizzata sulla ricezione di dati, il loro salvataggio in una memoria e la gestione di un processo di ricezione completata. Proprio come l’unità `A` è caratterizzata da diversi segnali di ingresso e uscita: `ingresso`, che rappresenta l’ingresso inviato al componente dall’unità `A` (pertanto era la sua uscita), un segnale di `srt` per iniziare la comunicazione, uno di `clock` ed un altro di `reset` ed infine dei bit per gestire inizio e fine della comunicazione seriale: `invio_avvenuto` e `ricezione_completata`.

```

1 architecture structural of unita_b is
2
3     component mem_B is
4         port(
5             address_mem_B : in std_logic_vector(2 downto 0);
6             data : in std_logic_vector(7 downto 0);
7             clk_mem_B : in std_logic;
8             wrt : in std_logic
9         );
10    end component;
11
12    component counter_mod_8 is
13        port (
14            clk_ctr, rst_ctr, en_ctr : in std_logic;
15            curr_value : out std_logic_vector(2 downto 0)
16        );
17    end component;

```

```

18
19 component Rs232RefComp is
20   Port (
21     TXD : out std_logic := '1';
22     RXD : in std_logic;
23     CLK : in std_logic;
24     DBIN : in std_logic_vector (7 downto 0);
25     DBOUT : out std_logic_vector (7 downto 0);
26     RDA : inout std_logic;
27     TBE : inout std_logic := '1';
28     RD : in std_logic;
29     WR : in std_logic;
30     PE : out std_logic;
31     FE : out std_logic;
32     OE : out std_logic;
33     RST : in std_logic := '0');
34 end component;
35
36 signal temp_address : std_logic_vector(2 downto 0);
37 signal fine : std_logic := '0';
38 signal temp_write : std_logic;
39 signal temp_rd : std_logic;
40 signal temp_rda : std_logic;
41 signal temp_en_ctr : std_logic;
42 signal temp_in : std_logic_vector(7 downto 0);
43 signal temp_read : std_logic;
44 signal temp_tbe : std_logic;
45 signal temp_wr : std_logic;
46 signal temp_txd : std_logic;
47 signal temp_dbin : std_logic_vector(7 downto 0);
48 signal temp_PE : std_logic;
49 signal temp_FE : std_logic;
50 signal temp_OE : std_logic;
51 type state is (IDLE, RICEZIONE, SALVATAGGIO);
52 signal current_state, next_state : state;
53
54
55 begin
56   cntr : counter_mod_8
57   port map (    clk_ctrl => ck,
58             rst_ctrl => rst,

```

```

59           en_ctr => temp_en_ctr,
60           curr_value => temp_address
61       );
62
63   MEM : mem_B
64       port map( address_mem_B => temp_address,
65                   data => temp_in,
66                   clk_mem_B => ck,
67                   wrt => temp_write
68   );
69
70   UART : Rs232RefComp
71       port map( TXD => temp_txd,
72                   RXD => ingresso,
73                   CLK => ck,
74                   DBIN => temp_dbin,
75                   DBOUT => temp_in,
76                   RDA => temp_rda,
77                   TBE => temp_tbe,
78                   RD => temp_rd,
79                   WR => temp_wr,
80                   PE => temp_PE,
81                   FE => temp_FE,
82                   OE => temp_OE,
83                   RST => rst
84   );
85
86   reg_stato: process(ck)
87     begin
88       if(ck'event and ck='1') then
89         if(rst = '1') then
90           current_state <= IDLE;
91         else
92           current_state <= next_state;
93         end if;
94       end if;
95     end process;
96
97   comb: process(current_state, srt, invio_avvenuto,
98                 temp_rda, fine)
99     begin

```

```

99
100    case current_state is
101        when IDLE =>
102            temp_rd <= '0';
103            temp_en_ctrl <= '0';
104            temp_write <= '0';
105            ricezione_completata <= '0';
106            if(srt = '1' and fine = '0') then
107                next_state <= RICEZIONE;
108            else
109                next_state <= IDLE;
110            end if;
111
112        when RICEZIONE =>
113            if(invio_avvenuto = '1') then
114                if(temp_rda = '1') then
115                    next_state <= SALVATAGGIO;
116                else
117                    next_state <= RICEZIONE;
118                end if;
119            else
120                next_state <= RICEZIONE;
121            end if;
122
123        when SALVATAGGIO =>
124            ricezione_completata <= '1';
125            temp_rd <= '1';
126            temp_en_ctrl <= '1';
127            temp_write <= '1';
128
129            if(temp_address = "111") then
130                fine <= '0';
131                next_state <= IDLE;
132            else
133                next_state <= IDLE;
134            end if;
135
136        when others =>
137            null;
138
139    end case;

```

```

140      end process;
141
142 end structural;

```

Listing 6.4: Implementazione dell'Unità B

Anche l'unità B è realizzata mediante un approccio **strutturale**, i componenti principali sono i seguenti:

- **mem_B**: Questo componente rappresenta una memoria che accetta un indirizzo (3 bit), dati in ingresso (8 bit), un segnale di clock e un segnale di scrittura. Quando il segnale di scrittura è attivo, i dati vengono scritti nell'indirizzo specificato, che è scandito dal contatore.
- **Counter_mod_8**: il componente implementa un contatore modulo 8, avanzando il suo valore corrente a ogni segnale di enable e resettandosi con il segnale di reset. L'output è un valore di 3 bit che è utilizzato come indirizzo per la memoria.
- **Rs232RefComp**: Gestisce la comunicazione seriale secondo lo standard RS-232, con segnali per la trasmissione e la ricezione dei dati, oltre a vari controlli e segnalazioni di stato.

Sono poi impiegati una serie di segnali temporanei, per la gestione dei segnali di controllo generati dalla **control unit**, questi agiscono come intermediari tra i diversi componenti facilitando operazioni come la lettura, la scrittura, la ricezione e la trasmissione dei dati. La **control unit** è implemetata mediante un ASM con tre stati principali: **IDLE**, **RICEZIONE**, e **SALVATAGGIO**. L'automa è descritto dettagliatamente nel paragrafo precedente.

System

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use work.all;
5
6 entity system is
7   port (
8     sys_start, sys_clock, sys_reset : in std_logic
9   );
10 end system;

```

Listing 6.5: Implementazione entity del sistema

Il codice inizia con la dichiarazione dell'entità ‘**system**‘, che rappresenta il componente descritto. Questa entità ha tre ingressi: ‘**sys_start**‘, ‘**sys_clock**‘, e ‘**sys_reset**‘.

```
1 architecture structural of system is
2
3     component unita_a is
4         port (
5             start : in std_logic;
6             clock, reset : in std_logic;
7             uscita : out std_logic;
8             invio_richiesta : out std_logic;
9             ricezione_avvenuta : in std_logic
10        );
11    end component;
12
13    component unita_b is
14        port (
15            ingresso : in std_logic;
16            srt : in std_logic;
17            ck, rst : in std_logic;
18            invio_avvenuto : in std_logic;
19            ricezione_completata : out std_logic
20        );
21    end component;
22
23    signal temp : std_logic;
24    signal temp_invio : std_logic;
25    signal temp_ricezione : std_logic;
26
27    begin
28        UA : unita_a
29            port map(
30                start => sys_start,
31                clock => sys_clock,
32                reset => sys_reset,
33                uscita => temp,
34                invio_richiesta => temp_invio,
35                ricezione_avvenuta => temp_ricezione
36            );
37
38        UB : unita_b
39            port map(
```

```

40      ingresso => temp,
41      srt => sys_start,
42      ck => sys_clock,
43      rst => sys_reset,
44      invio_avvenuto => temp_invio,
45      ricezione_completata => temp_ricezione
46  );
47
48 end structural;

```

Listing 6.6: Implementazione del sistema

Segue una definizione **structural** dell’architettura. L’architettura descrive comportamento e la struttura interna del sistema. All’interno di quest’ultima, vengono definiti due componenti, ‘**unita_a**’ e ‘**unita_b**’, ciascuno con le proprie porte di ingresso/uscita che consentono la comunicazione e il controllo. Il componente ‘**unita_a**’ ha un insieme di porte tra cui ‘start’, ‘clock’, ‘reset’ per i segnali di controllo, ‘uscita’ per l’output del dato elaborato, ‘invio_richiesta’ per indicare una richiesta di trasmissione, e ‘ricezione_avvenuta’ per indicare la ricezione del dato dal lato ricezione. Questi due ultimi segnali sono usati per la gestione della comunicazione seriale. Il componente ‘**unita_b**’ riceve dati attraverso la porta ‘ingresso’ e ha segnali di controllo simili a ‘**unita_a**’, oltre a ‘invio_avvenuto’ per indicare il completamento di un invio e ‘ricezione_completata’ per segnalare la fine di una ricezione. Sono definiti dei segnali temporanei per facilitare la comunicazione tra i due componenti agendo come collegamenti intermedi. La sezione finale del codice, descrive come i componenti siano connessi tra loro a formare il componente ‘system’. ‘UA’ e ‘UB’ sono istanze dei componenti e il ‘port map’ collega i segnali dell’entità ‘system’ e i segnali intermedi ai corrispondenti segnali di ingresso/uscita dei componenti.

6.2.2 Simulazione

Per la simulazione del componente è stato realizzato un **testbench**, riportato di seguito:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity system_tb is
6 -- Testbench doesn't have ports.
7 end system_tb;
8

```

```

9 architecture tb of system_tb is
10    -- Signals corresponding to system's ports
11    signal tb_sys_start, tb_sys_clock, tb_sys_reset :
12        std_logic := '0';
13
14    -- Component declaration for the system
15    component system is
16        port (
17            sys_start, sys_clock, sys_reset : in std_logic
18        );
19    end component;
20
21 begin
22    -- Instance of the system
23    uut: system
24        port map(
25            sys_start => tb_sys_start,
26            sys_clock => tb_sys_clock,
27            sys_reset => tb_sys_reset
28        );
29
30    -- Clock generation
31    clock_gen: process
32    begin
33        while true loop
34            tb_sys_clock <= '0';
35            wait for 5 ns; -- Adjust the clock period
36                according to your design requirements
37            tb_sys_clock <= '1';
38            wait for 5 ns;
39        end loop;
40    end process;
41
42    -- Reset process
43    reset_gen: process
44    begin
45        tb_sys_reset <= '1';
46        wait for 20 ns; -- Adjust the reset duration according
47            to your design requirements
48        tb_sys_reset <= '0';
49        wait;

```

```

47      end process;
48
49      -- Stimulus process
50      stimulus_proc: process
51      begin
52          -- Initial state
53          tb_sys_start <= '0';
54          wait for 100 ns; -- Wait for system to stabilize
55
56          -- Test case 1: Start signal
57          tb_sys_start <= '1';
58          wait for 5000 ns;
59          --tb_sys_start <= '0';
60
61          -- Add more test cases as required to fully exercise
62          -- the system
63
64          wait; -- Terminate the simulation
65
66      end process;
67
68  end tb;

```

Listing 6.7: Implementazione entity dell'Unità B

In questo file vengono dichiarati alcuni segnali: tb_sys_start, tb_sys_clock e tb_sys_reset, che corrispondono ai segnali di ingresso del componente system che si desidera testare. Questi segnali saranno utilizzati per simulare le condizioni operative del sistema. All'interno dell'architettura viene dichiarato il componente system. Un'istanza di questo componente, denominata uut viene creata e i suoi ingressi sono mappati ai segnali di test definiti. È previsto un processo per la generazione del segnale di clock del componente sotto test. Il clock alterna il valore da '0' a '1' e viceversa ogni 5 nanosecondi, simulando un segnale di clock reale. È presente poi un secondo processo, per la generazione del segnale di reset per inizializzare o resettare il componente sotto test. Vengono poi sottoposti al sistema gli stimoli al fine di effettuare il test del componente sviluppato, mediante un ulteriore process. I risultati dei test condotti sono riportati di seguito:

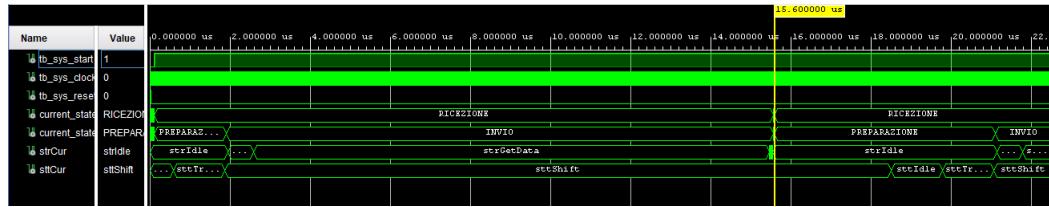


Figure 6.4: Simulazione del sistema.

Abbiamo inoltre riportato anche il contenuto della memoria ROM e della memoria MEM, per maggior chiarezza:

ROM	memoria_A(dataflow)	memoria_A_8[0:7][7:0]	11,22,44,88,12,14,18,21	Array
UART	Rs232RefComp(Behavior)	> [0][7:0]	11	Array
UB	unita_b(structural)	> [1][7:0]	22	Array
cntr	counter_mod_8(behavior)	> [2][7:0]	44	Array
MEM	mem_B(dataflow)	> [3][7:0]	88	Array
UART	Rs232RefComp(Behavior)	> [4][7:0]	12	Array
		> [5][7:0]	14	Array
		> [6][7:0]	18	Array
		> [7][7:0]	21	Array

Figure 6.5: Contenuto memoria ROM.

ROM	memoria_A(dataflow)	memoria_A_8[0:7][7:0]	11,22,44,88,12,14,18,21	Array
UART	Rs232RefComp(Behavior)	> [0][7:0]	11	Array
UB	unita_b(structural)	> [1][7:0]	22	Array
cntr	counter_mod_8(behavior)	> [2][7:0]	44	Array
MEM	mem_B(dataflow)	> [3][7:0]	88	Array
UART	Rs232RefComp(Behavior)	> [4][7:0]	12	Array
		> [5][7:0]	14	Array
		> [6][7:0]	18	Array
		> [7][7:0]	21	Array

Figure 6.6: Contenuto memoria MEM.

CHAPTER 7

SWITCH MULTISTADIO

7.1 Esercizio 11.1 - switch multistadio

Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network. Lo switch deve consentire lo scambio di messaggi di 2 bit ciascuno da un nodo sorgente a un nodo destinazione in una rete con 4 nodi, implementando uno schema a priorità fissa fra i nodi (es. nodo 1 più prioritario, con priorità decrescente fino al nodo 4).

7.2 Progetto e architettura

Lo switch multistadio è un dispositivo di rete progettato per gestire grandi quantità di dati in reti complesse. Suddivide la rete in più livelli o stadi, migliorando la scalabilità e la velocità di commutazione dei pacchetti. Questo approccio gerarchico riduce la congestione e ottimizza le prestazioni complessive della rete.

Nel nostro caso è richiesto uno switch multistadio 4x4, dunque, saranno necessari $\log_2 4 = 2$ stadi, con due switch ciascuno. L'unità fondamentale è lo switch:

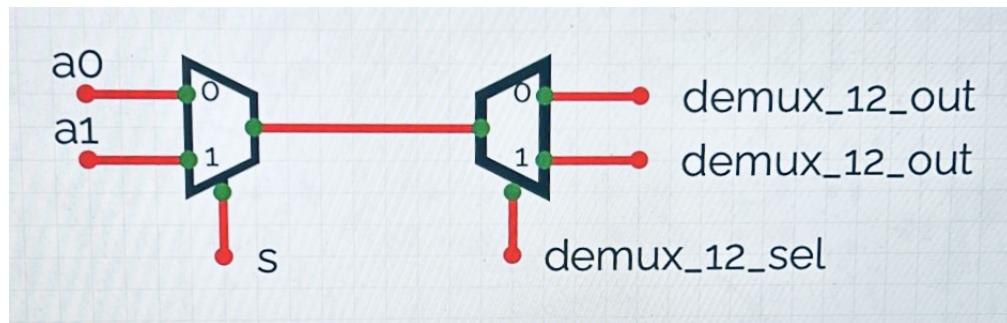


Figure 7.1: switch

costituito da un multiplexer 2:1 interconnesso opportunamente ad un demultiplexer 1:2. All'ingresso di selezione dell'i-esimo multiplexer corrisponde l'i-esimo bit dell'indirizzo della sorgente, mentre all'ingresso di selezione dell'i-esimo demultiplexer corrisponde l'esimo bit dell'indirizzo della destinazione.

Gli switch vengono interconnessi tra di loro sfruttando l'algoritmo del *perfect shuffling*. Nel nostro caso specifico abbiamo come ingressi x_0, x_1, x_2, x_3 , dividiamo in due l'insieme, e otterremo x_0, x_1 e x_2, x_3 , il che vuol dire che al

primo stadio avremo per il primo switch in ingresso x_0 ed x_2 , mentre per il secondo switch x_1 ed x_3 . Ripetiamo l'algoritmo anche per il secondo stadio, avremo gli insiemi x_0, x_1, x_2 e x_3 , il che significa che nel primo switch entreranno 0 ed 1, mentre nel secondo 2 e 3:

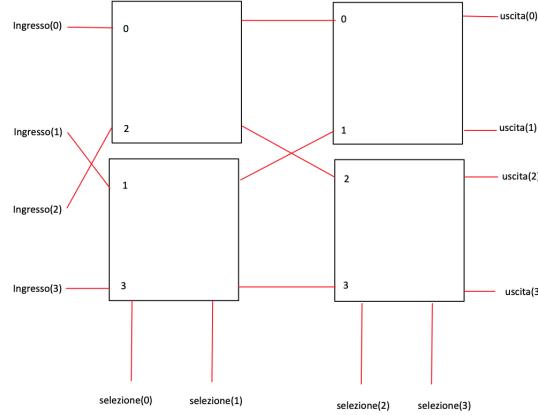


Figure 7.2: unità operativa

rappresenta l'**unità operativa** del nostro sistema.

Al fine di gestire le priorità decrescenti dei nodi è stato introdotto un nuovo componente, denominato **gestore**, il quale ha il compito di assegnare la priorità ai nodi durante la comunicazione. In particolare, l'output che fornisce corrisponde all'indirizzo della sorgente, dunque, avrà un collegamento con gli ingressi di selezione dei multiplexer.

Unità operativa e gestore vengono interconnessi per formare l'**omega network**:

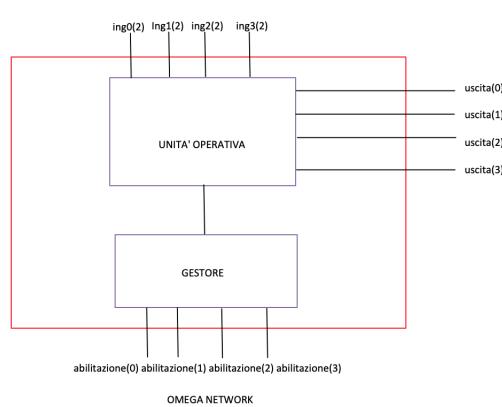


Figure 7.3: omega network

La quale presenta i seguenti ingressi:

- **abilitazione** : vettore di 4 bit che indica quali nodi vogliono cominciare la comunicazione.

- `ing0, ing1, ing2, ing3` : sono tutti vettori di 3 bit, che indicano il contenuto del messaggio di ciascun nodo, in particolare i primi due bit del messaggio rappresentano l'indirizzo destinazione, mentre l'ultimo il payload.

e come unica uscita:

- `uscita_sistema` : i bit ricevuti dai nodi sorgente.

7.2.1 Implementazione

Iniziamo con l'analisi del componente elementare, lo **switch** :

,

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity switch is
5   port (
6     x_switch: in STD_LOGIC_VECTOR(1 downto 0);
7     sel: in STD_LOGIC_VECTOR(1 downto 0);
8     y_switch: out STD_LOGIC_VECTOR(1 downto 0)
9   );
10 end switch;
11
12 architecture structural of switch is
13
14   component mux21 is
15     port (
16       a0 : in STD_LOGIC;
17       a1 : in STD_LOGIC;
18       s : in STD_LOGIC;
19       y : out STD_LOGIC
20     );
21   end component;
22
23   component demux12 is
24     port (
25       demux_12_in : in STD_LOGIC;
26       demux_12_sel : in STD_LOGIC;
27       demux_12_out : out STD_LOGIC_VECTOR(1 downto 0)
28     );
29   end component;
30
```

```

31      signal temp : STD_LOGIC;
32
33 begin
34
35     mux: mux21
36         port map(
37             a0 => x_switch(0),
38             a1 => x_switch(1),
39             s => sel(0),
40             y => temp
41         );
42
43     demux: demux12
44         port map(
45             demux_12_in => temp,
46             demux_12_sel => sel(1),
47             demux_12_out => y_switch
48         );
49
50 end structural;

```

Vede come ingressi:

- `x_switch` : vettore di due bit che rappresenta due nodi in ingresso.
- `sel` : vettore di due bit che rappresenta gli ingressi di selezione rispettivamente del multiplexer e del demultiplexer.

e uscite:

- `y_switch` : vettore di due bit che rappresenta nodi intermedi/destinazione.

Il segnale temporaneo `temp` rappresenta il punto di collegamento tra l'uscita del multiplexer e l'ingresso del demultiplexer.

Passiamo all'**unità operativa**:

```

1 ,
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4
5 entity unita_operativa is
6     port (
7         ingresso: in STD_LOGIC_VECTOR(3 downto 0);

```

```

7      selezione: in STD_LOGIC_VECTOR(3 downto 0);
8      uscita: out STD_LOGIC_VECTOR(3 downto 0)
9  );
10 end entity;
11
12 architecture structural of unita_operativa is
13     signal temp_uscita : STD_LOGIC_VECTOR(3 downto 0);
14
15     component switch is
16         port (
17             x_switch: in STD_LOGIC_VECTOR(1 downto 0);
18             sel: in STD_LOGIC_VECTOR(1 downto 0);
19             y_switch: out STD_LOGIC_VECTOR(1 downto 0)
20         );
21     end component;
22
23 begin
24     switch0: switch
25         port map(
26             x_switch(0) => ingresso(0),
27             x_switch(1) => ingresso(2),
28             sel(0) => selezione(0),
29             sel(1) => selezione(1),
30             y_switch(0) => temp_uscita(0),
31             y_switch(1) => temp_uscita(1)
32         );
33
34     switch1: switch
35         port map(
36             x_switch(0) => ingresso(1),
37             x_switch(1) => ingresso(3),
38             sel(0) => selezione(0),
39             sel(1) => selezione(1),
40             y_switch(0) => temp_uscita(2),
41             y_switch(1) => temp_uscita(3)
42         );
43
44     switch2: switch
45         port map(
46             x_switch(0) => temp_uscita(0),
47             x_switch(1) => temp_uscita(2),

```

```

48      sel(0) => selezione(2),
49      sel(1) => selezione(3),
50      y_switch(0) => uscita(0),
51      y_switch(1) => uscita(1)
52  );
53
54  switch3: switch
55    port map(
56      x_switch(0) => temp_uscita(1),
57      x_switch(1) => temp_uscita(3),
58      sel(0) => selezione(2),
59      sel(1) => selezione(3),
60      y_switch(0) => uscita(2),
61      y_switch(1) => uscita(3)
62  );
63 end architecture;

```

L'unità operativa presenta i seguenti ingressi:

- ingresso : vettore di 4 bit che rappresenta i 4 nodi sorgente.
- selezione : vettore di 4 bit, di cui i primi due sono gli ingressi di selezione del primo stadio (primo bit della sorgente e primo bit della destinazione), mentre gli altri due per il secondo stadio (ultimo bit della sorgente e ultimo bit della destinazione).

E un'unica uscita:

- uscita : vettore di 4 bit che rappresenta i 4 nodi destinazione.

Essa è composta da 4 componenti switch, come abbiamo detto precedentemente, due al primo stadio e due al secondo, collegati opportunamente secondo l'algoritmo di *perfect shuffling* specificato nella sezione precedente. Viene utilizzato un unico segnale d'appoggio, `temp_uscita`, vettore di 4 bit che permette il collegamento tra le uscite degli switch del primo livello e gli ingressi degli switch del secondo livello.

L'ultimo componente da analizzare prima di passare alla **omega network** è il **gestore di priorità**:

```

,
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3

```

```

4 entity gestore is
5   port (
6     en0, en1, en2, en3 : in STD_LOGIC;
7     sel_sorgente: out STD_LOGIC_VECTOR(1 downto 0)
8   );
9   end entity;
10
11 architecture dataflow of gestore is
12   begin
13     sel_sorgente <= "00" when en0 = '1' else
14       "01" when en1 = '1' else
15       "10" when en2 = '1' else
16       "11" when en3 = '1' else
17       "--";
18   end architecture;

```

I suoi ingressi sono:

- en0, en1, en2, en3 : che indicano quali nodi vogliono prendere parte alla comunicazione.

e l'uscita:

- sel_sorgente : che specifica qual è l'indirizzo della sorgente selezionato dal gestore.

La sua logica è specificata all'interno del *dataflow*: mediante il costrutto di assegnazione condizionale viene selezionato l'indirizzo sorgente del nodo più prioritario tra quelli abilitati.

Infine, discutiamo dell'implementazione dell'**omega network**:

```

,
1
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4
5 entity omega_network is
6   port (
7     abilitazione: in STD_LOGIC_VECTOR(3 downto 0);
8     ing0: in STD_LOGIC_VECTOR(2 downto 0);
9     ing1: in STD_LOGIC_VECTOR(2 downto 0);
10    ing2: in STD_LOGIC_VECTOR(2 downto 0);
11    ing3: in STD_LOGIC_VECTOR(2 downto 0);

```

```

12         uscita_sistema: out STD_LOGIC_VECTOR(3 downto 0)
13     );
14 end omega_network;
15
16 architecture structural of omega_network is
17
18     signal temp_sorgente: STD_LOGIC_VECTOR(1 downto 0);
19     signal temp_destinazione: STD_LOGIC_VECTOR(1 downto 0);
20
21     component gestore is
22         port(
23             en0, en1, en2, en3 : in STD_LOGIC;
24             sel_sorgente: out STD_LOGIC_VECTOR(1 downto 0)
25         );
26     end component;
27
28     component unita_operativa is
29         port(
30             ingresso: in STD_LOGIC_VECTOR(3 downto 0);
31             selezione: in STD_LOGIC_VECTOR(3 downto 0);
32             uscita: out STD_LOGIC_VECTOR(3 downto 0)
33         );
34     end component;
35
36 begin
37     gest: gestore
38         port map(
39             en0 => abilitazione(0),
40             en1 => abilitazione(1),
41             en2 => abilitazione(2),
42             en3 => abilitazione(3),
43             sel_sorgente => temp_sorgente
44         );
45
46         --i primi due bit sono la destinazione, l'ultimo e' il
47         -- payload.
48     uo: unita_operativa
49         port map(
50             ingresso(0) => ing0(2),
51             ingresso(1) => ing1(2),
52             ingresso(2) => ing2(2),

```

```

52         ingresso(3) => ing3(2),
53         selezione(0) => temp_sorgente(0),
54         selezione(1) => temp_destinazione(0),
55         selezione(2) => temp_sorgente(1),
56         selezione(3) => temp_destinazione(1),
57         uscita => uscita_sistema
58     );
59
60     with temp_sorgente select
61
62         temp_destinazione(0) <= ing0(0) when "00",
63                         ing1(0) when "01",
64                         ing2(0) when "10",
65                         ing3(0) when "11",
66                         '-' when others;
67
68     with temp_sorgente select
69         temp_destinazione(1) <= ing0(1) when "00",
70                         ing1(1) when "01",
71                         ing2(1) when "10",
72                         ing3(1) when "11",
73                         '-' when others;
74
75 end structural;

```

Nell’architettura strutturale vengono istanziati come componenti l’unità operativa e il gestore. Il primo segnale d’appoggio è `temp_sorgente`, mediante il quale collegiamo l’uscita del gestore agli ingressi di selezione dell’unità operativa lato sorgente. Il secondo segnale d’appoggio è `temp_destinazione`, il quale viene inizializzato in base ad un costrutto di assegnazione condizionale, dipendente dal valore assunto dalla sorgente, in particolare, assume il valore dei primi due bit del messaggio del nodo sorgente, e viene collegato agli ingressi di selezione dell’unità operativa lato destinazione.

7.2.2 Simulazione

Vediamo il codice per il testbench:

```

,
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

```

5
6 entity omega_network_tb is
7 end omega_network_tb;
8
9 architecture tb of omega_network_tb is
10
11     component omega_network
12         port (
13             abilitazione : in STD_LOGIC_VECTOR(3 downto 0);
14             ing0 : in STD_LOGIC_VECTOR(2 downto 0);
15             ing1 : in STD_LOGIC_VECTOR(2 downto 0);
16             ing2 : in STD_LOGIC_VECTOR(2 downto 0);
17             ing3 : in STD_LOGIC_VECTOR(2 downto 0);
18             uscita_sistema : out STD_LOGIC_VECTOR(3 downto 0)
19         );
20     end component;
21
22     signal abilitazione : STD_LOGIC_VECTOR(3 downto 0);
23     signal ing0, ing1, ing2, ing3 : STD_LOGIC_VECTOR(2 downto
24         0);
25     signal uscita_sistema : STD_LOGIC_VECTOR(3 downto 0);
26
27 begin
28     uut: omega_network
29         port map (
30             abilitazione => abilitazione,
31             ing0 => ing0,
32             ing1 => ing1,
33             ing2 => ing2,
34             ing3 => ing3,
35             uscita_sistema => uscita_sistema
36         );
37
38     stimulus: process
39         begin
40             -- Scenario 1:
41             abilitazione <= "0001";
42             ing0 <= "101";
43             ing1 <= "010";
44             ing2 <= "111";

```

```

45      ing3 <= "001";
46      wait for 10 ns;
47
48      -- Scenario 2:
49      abilitazione <= "1010";
50      ing0 <= "011";
51      ing1 <= "100";
52      ing2 <= "001";
53      ing3 <= "110";
54      wait for 10 ns;
55      wait;
56  end process;
57
58 end tb;

```

L'unità sottoposta al test (*uut*) è la **omega network**, i cui ingressi e uscite vengono mappate sui segnali omonimi `abilitazione`, `ing0`, `ing1`, `ing2`, `ing3` e `uscita_sistema`.

Nel processo abbiamo due scenari, il primo abilita il primo nodo, che invierà come payload il bit '1' al terzo nodo destinazione, mentre il secondo abilita il secondo e il quarto nodo, ma il secondo sarà più prioritario, dunque, invierà come payload il bit '0' al terzo nodo destinazione. Vediamo i risultati della simulazione:



Figure 7.4: Risultato della simulazione del tb dell'omega network.