



UNIVERSITÀ^{DEGLI STUDI DI}
NAPOLI FEDERICO II

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

Corso di Architetture di sistemi digitali

Elaborato progettuale del gruppo 27 bis

Anno Accademico 2022/2023

Studenti
Vallefouoco Roberto
Michele Savella

Indice

1 Rete di interconnessione	5
1.1 Progettazione in VHDL di un Mux 16:1	5
1.1.1 Schematici	5
1.1.2 Codice VHDL	8
1.1.3 Simulazione	11
1.2 Progettazione in VHDL di una rete di interconnessione 16:4	12
1.2.1 Traccia	12
1.2.2 Schematici	12
1.2.3 Codice VHDL	14
1.2.4 Simulazione	18
1.3 Implementazione su board della rete di interconnessione	19
1.3.1 Traccia	19
1.3.2 Schematici	19
1.3.3 Codice VHDL	21
2 Encoder	25
2.1 Progettazione in VHDL	25
2.1.1 Schematici	25
2.1.2 Codice VHDL	26
2.1.3 Simulazione	26
2.2 Implementazione su board	28
2.2.1 Soluzione	28
2.3 Implementazione su board con display	29
2.3.1 Schematici	29
2.3.2 Codice VHDL	31
3 Riconoscitore di sequenze	36

3.1	Progettazione in VHDL	36
3.1.1	Schematici	36
3.1.2	Codice VHDL	37
3.1.3	Simulazione	40
3.2	Implementazione su board	41
3.2.1	Schematici	41
3.2.2	Codice VHDL	42
4	Shift register	52
4.1	Progettazione in VHDL con approccio comportamentale	52
4.1.1	Schematici	52
4.1.2	Codice VHDL	52
4.1.3	Simulazione	55
4.2	Progettazione in VHDL con approccio strutturale	56
4.2.1	Schematici	56
4.2.2	Codice VHDL	57
4.2.3	Simulazione	61
5	Cronometro	62
5.1	Progettazione in VHDL	62
5.1.1	Schematici	62
5.1.2	Codice VHDL	67
5.1.3	Simulazione	75
5.2	Implementazione su board con display	76
5.2.1	Schematici	76
5.2.2	Codice VHDL	81
5.2.3	Implementazione su board	94
5.3	Memorizzazione di intertempi e visualizzazione	96
5.3.1	Schematici	96
5.3.2	Codice VHDL	99

5.3.3	Implementazione su board	104
6	Sistema di testing	107
6.1	Progettazione in VHDL	107
6.1.1	Schematici	107
6.1.2	Codice VHDL	110
6.1.3	Simulazione	115
6.2	Implementazione su board	115
6.2.1	Schematici	115
6.2.2	Codice VHDL	116
7	Comunicazione con Handshaking	118
7.1	Progettazione in VHDL	119
7.1.1	Schematici	120
7.1.2	Codice VHDL	123
7.1.3	Simulazione	127
8	Processore MIC-1	129
8.1	Studio di due istruzioni del processore	132
8.1.1	Codici operativi analizzati	132
8.2	Modifica di un codice operativo	136
8.2.1	Creazione di un codice operativo: IXOR	136
9	Interfaccia seriale	142
9.1	Progettazione in VHDL e implementazione su board	143
9.1.1	Schematici	143
9.1.2	Codice VHDL	152
9.1.3	Implementazione su scheda	174
9.2	Modifica del progetto utilizzando una ROM	175
9.2.1	Schematici	175
9.2.2	Codice VHDL	180

9.2.3	Implementazione su scheda	189
10	Switch multistadio	191
10.1	Progettazione in VHDL con priorità fissa	193
10.1.1	Schematici	193
10.1.2	Codice VHDL	196
10.1.3	Simulazione	200
10.2	Progettazione in VHDL con gestione dei conflitti	201
10.2.1	Schematici	201
10.2.2	Codice VHDL	204
10.2.3	Simulazione	210
10.3	Progettazione in VHDL con handshaking	211
10.3.1	Schematici	212
10.3.2	Codice VHDL	216
10.3.3	Simulazione	221
11	Macchine aritmetiche: Moltiplicatore di Booth	223
11.1	Progettazione in VHDL ed implementazione su board	225
11.1.1	Traccia	225
11.1.2	Schematici	226
11.1.3	Codice VHDL	229
11.1.4	Simulazione	233
11.2	Implementazione su board	234
12	Esercizio libero: Rete neurale	239
12.1	Progettazione in VHDL	241
12.1.1	Schematici	242
12.1.2	Codice VHDL	250
12.1.3	Simulazione	268

1 Rete di interconnessione

In questo capitolo affronteremo il progetto di un multiplexer 16:1 una rete di interconnessione 16:4.

1.1 Progettazione in VHDL di un Mux 16:1

Traccia

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 Schematici

Per implementare la soluzione al problema il multiplexer [16:1] è stato implementato in modo strutturale. Di seguito procediamo ad illustrare schematiche e codice VHDL delle varie soluzioni implementate.

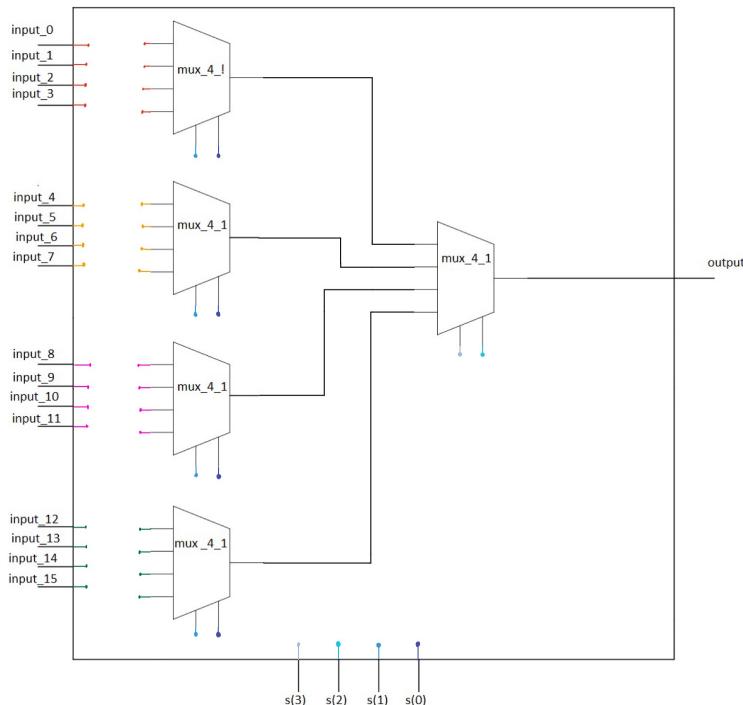


Figura 1: composizione del mux 16:1

Multiplexer [2:1]

Il multiplexer [2:1] é stato usato come componente di partenza per l'implementazione dei multiplexer [4:1]. Il mux [2:1] é dotato di 3 ingressi (2 per i dati ed 1 come selettore) ed 1 uscita (che trasporta i dati sulla linea indicata dal selettore). Il funzionamento del componente é molto semplice, i dati presenti sulla linea *input*₀ (se *s* = '0') o *input*₁ (se *s* = '1'), vengono passati sull'uscita *u*.

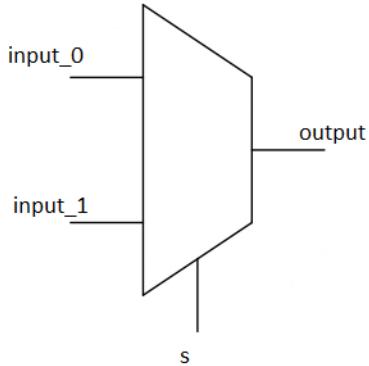


Figura 2: mux 2:1

Multiplexer [4:1]

Il multiplexer [4:1] (come illustrato nel paragrafo 1.1.1) é stato sviluppato attraverso la composizione di 3 multiplexer [2:1]. Il mux [4:1] é formato da un totale di 7 segnali: 6 ingressi (4 ingressi che trasportano dati *input*[0 : 4] e 2 ingressi per selezionare una linea in ingresso *s*[0 : 2]) ed 1 uscita *output* (che trasporta i dati della linea indicata dal selettore). Per ricreare la logica del componente é stato necessario utilizzare 2 multiplexer [2:1] su un livello (per poter selezionare tra le 4 linee di dati le 2 indicate da *s*[0]) e di un multiplexer 2:1 su un altro livello (per poter selezionare tra le linee di dati quella indicata da *s*[1] ed *s*[0]). Il componente é rappresentato nella figura 3

Multiplexer 16:1

Il multiplexer 16:1 é un componente che permette il passaggio dell'informazione da una linea d'ingresso (tra 16 possibili) ad una linea in uscita. É formato da 21 segnali: 20 in ingresso (*x*[0 : 15] linee per i dati, *k*[0 : 3] linee per il selettore) ed 1 uscita (*y*, per portare l'informazione in ingresso all'uscita). Il mux 16:1 é stato implementato (come richiesto dalla

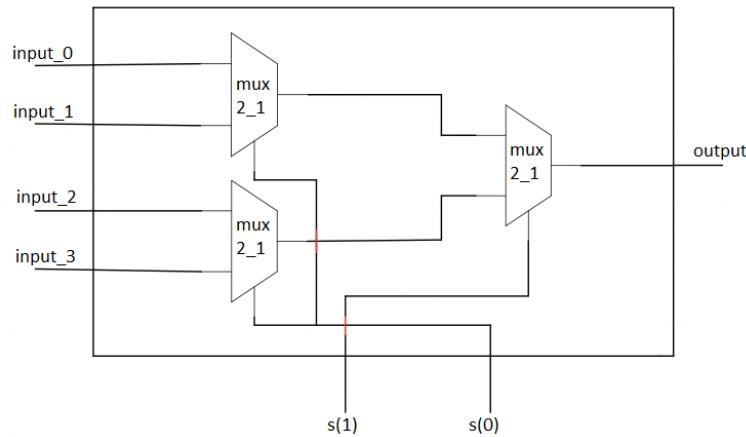


Figura 3: composizione del mux 4:1

traccia) per composizione di 5 multiplexer 4:1. I collegamenti dei vari mux 4:1 (per ricreare la logica del mux 16:1) sono presenti nella figura 4.

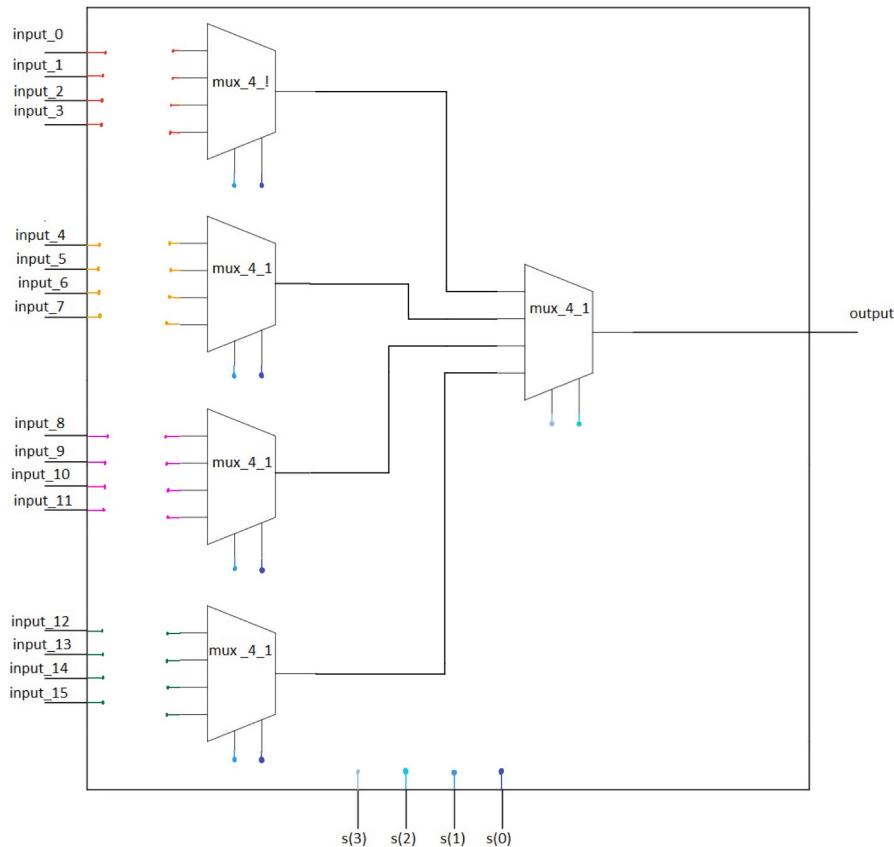


Figura 4: composizione del mux 16:1

1.1.2 Codice VHDL

Passiamo ora a presentare il codice VHDL che ci ha permesso di implementare i componenti visti nella precedente sezione.

Multiplexer [2:1]

Il multiplexer [2:1], come detto precedentemente, è l'elemento base dal quale di è partiti per la progettazione del mux [4:1]. Il mux[2:1] è stato sviluppato utilizzando 3 ingressi (input_0 ed input_1 per i dati, s per la selezione) ed 1 uscita (output) implementati come STD_LOGIC. L'architettura è stata descritta in dataflow, dove è stata data all'uscita la funzione del multiplexer. di seguito il codice:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4
5 entity mux_2_1 is
6   port (
7     input0 : in std_logic;
8     input1 : in std_logic;
9     s : in std_logic;
10    output : out std_logic
11  );
12 end mux_2_1;
13
14 architecture dataflow of mux_2_1 is
15
16 begin
17
18   output <= ((not s) and input0) or (s and input1);
19
20 end dataflow;
```

Multiplexer [4:1]

Il multiplexer [4:1] è stato sviluppato per composizione a partire da tre mux [2:1]. Per descrivere il componente quindi è stata adottata una descrizione del tipo structural (che facilita la descrizione dei collegamenti tra componenti). Il componente ha 5 ingressi (un vettore input[0 to 3] per i dati ed un vettore s[1 downto 0] per la selezione) ed 1 uscita (output). La scelta di una numerazione differente per il vettore del selettore ed il vettore dei dati è stata fatta per poter semplificare la lettura del progetto (visto che in questo modo i bit più significativi dei vettori del componente è uguale a quella del mux visto a lezione). Infine è stato necessario implementare 2 segnali interni al componente per poter collegare i vari mux[2:1].

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux_4_1 is
5   port (
6     input: in std_logic_vector(0 to 3);
7     s: in std_logic_vector(1 downto 0);
8     output: out std_logic
9   );
10 end mux_4_1;
11
12 architecture structural of mux_4_1 is
13
14 component mux_2_1 is
15   port (
16     input0 : in std_logic;
17     input1 : in std_logic;
18     s : in std_logic;
19     output : out std_logic
20   );
21 end component;
22
23 signal c: std_logic_vector (0 to 1);
24
25 begin
26
27 mux0_0: mux_2_1
28   port map (
29     input0 => input(0),
30     input1 => input(1),
31     s => s(0),
32     output => c(0)
33   );
34
35 mux0_1: mux_2_1
36   port map (
37     input0 => input(2),
38     input1 => input(3),
39     s => s(0),
40     output => c(1)
41   );
42
43 mux1: mux_2_1
44   port map (
45     input0 => c(0),
46     input1 => c(1),
47     s => s(1),
48     output => output
49   );
50
51 end structural;

```

Multiplexer [16:1]

Il multiplexer [16:1] è stato sviluppato per composizione di 5 mux [4:1]. Il componente è formato da 20 segnali in ingresso (input[0 to 16] per i dati e s[3 downto 0] per il selettore)

ed un'uscita (output); per descriverlo è stata adottata una descrizione del tipo structural. Infine è stato necessario aggiungere un vettore di 4 segnali interno per poter collegare i vari multiplexer. Il codice VHDL del mux[16:1] è:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux_16_1 is
5     port (
6         input: in std_logic_vector(0 to 15);
7         s: in std_logic_vector(3 downto 0);
8         output: out std_logic
9     );
10 end mux_16_1;
11
12 architecture structural of mux_16_1 is
13
14 component mux_4_1 is
15     port (
16         input: in std_logic_vector(0 to 3);
17         s: in std_logic_vector(1 downto 0);
18         output: out std_logic
19     );
20 end component;
21
22 signal c: std_logic_vector(0 to 3);
23
24 begin
25
26     mux0_0: mux_4_1
27         port map (
28             input(0 to 3) => input(0 to 3),
29             s(1 downto 0) => s(1 downto 0),
30             output => c(0)
31         );
32
33     mux0_1: mux_4_1
34         port map (
35             input(0 to 3) => input(4 to 7),
36             s(1 downto 0) => s(1 downto 0),
37             output => c(1)
38         );
39
40     mux0_2: mux_4_1
41         port map (
42             input(0 to 3) => input(8 to 11),
43             s(1 downto 0) => s(1 downto 0),
44             output => c(2)
45         );
46
47     mux0_3: mux_4_1
48         port map (
49             input(0 to 3) => input(12 to 15),
50             s(1 downto 0) => s(1 downto 0),
51             output => c(3)
52         );
53

```

```

54 mux1: mux_4_1
55     port map (
56         input(0 to 3) => input(0 to 3),
57         s(1 downto 0) => s(3 downto 2),
58         output => output
59     );
60
61 end structural;

```

1.1.3 Simulazione

Come richiesto dalla traccia è stato eseguito anche il testing del componente, dove sono stati testati 3 casi:

- input = ("1100000100000111"), s=("0000"), assert su output = '1'
- input = ("0100000100010000"), s=("0001"), assert su output = '1'
- input = ("0000000100010000"), s=("0001"), assert su output = '0'

Di seguito il codice del testbench.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux_16_1_TB is
5 --  Port ();
6 end mux_16_1_TB;
7
8 architecture Behavioral of mux_16_1_TB is
9
10 component mux_16_1 is
11     port (
12         input: in std_logic_vector(0 to 15);
13         s: in std_logic_vector(3 downto 0);
14         output: out std_logic
15     );
16 end component;
17
18 signal inputs: std_logic_vector(0 to 15) := (others => 'U');
19 signal selections_m: std_logic_vector(3 downto 0):= (others => 'U');
20 signal output: std_logic := 'U';
21
22 begin
23
24 uut: mux_16_1
25     port map (
26         input => inputs,
27         s=> selections_m,
28         output => output
29     );
30
31 stim_proc: process

```

```

32
33 begin
34
35 wait for 100 ns;
36
37 -- input[2]=1 (mux)
38 wait for 10 ns;
39 inputs <= "1100000100000111";
40 selections_m <= "0000";
41 wait for 10 ns;
42 assert output = '1'
43 report "errore caso 0"
44 severity failure;
45
46 -- input[1]=1
47
48 wait for 10 ns;
49 inputs <= "0100000100010000";
50 selections_m <= "0001";
51 wait for 10 ns;
52 assert output = '1'
53 report "errore caso 1"
54 severity failure;
55
56 -- input[1]=0
57
58 wait for 10 ns;
59 inputs <= "0000000100010000";
60 selections_m <= "0001";
61 wait for 10 ns;
62 assert output = '0'
63 report "errore caso 2"
64 severity failure;
65
66 wait;
67 end process;
68 end Behavioral;

```

1.2 Progettazione in VHDL di una rete di interconnessione 16:4

1.2.1 Traccia

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.

1.2.2 Schematici

Per la realizzazione della rete di interconnessione [16:4] necessitiamo di due componenti, un MUX 16:4 e un DEMUX 1:4.

Il MUX [16:1] è stato già realizzato nella sezione 1, quindi useremo lo stesso componente implementato in precedenza, che possiede 16 sorgenti in ingresso, 4 ingressi di selezione e una singola uscita.

Quindi, prima di realizzare la rete di interconnessione, resta da implementare il demultiplexer [1:4]. Per farlo abbiamo deciso di utilizzare tre DEMUX [1:2].

Demultiplexer [1:2]

Il DEMUX [1:2] è una semplice macchina combinatoria, con una sorgente in ingresso, due ingressi di selezione e due uscite. In particolare abbiamo realizzato un DEMUX [1:2] in modo indirizzabile, utilizzando un'unica linea di selezione codificata (cioè r in figura 5). Di seguito lo schematico del DEMUX [1:2].

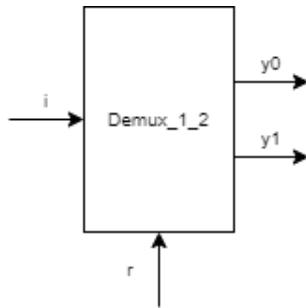


Figura 5: Demultiplexer [1:2]

Demultiplexer [1:4]

Partendo da tre DEMUX [1:2] abbiamo realizzato, tramite composizione, il DEMUX [1:4]. Vi è una sorgente in ingresso (j), due ingressi di selezione codificati ($s(0), s(1)$), e quattro uscite ($u(0), u(1), u(2), u(3)$). In figura 6 sono illustrati i collegamenti effettuati per permettere un corretto funzionamento del componente.

La struttura è divisa su due livelli:

1. Il primo livello è composto dal solo il primo demux [1:2], che ha in ingresso l'unico ingresso del demux [1:4] e la linea di selezione del demux [1:4] corrispondente al bit più significativo
2. Nel secondo livello, invece, vi sono gli altri due demux [1:2], ciascuno ha in ingresso un'uscita del demux del primo livello, mentre l'ingresso di selezione per entrambi e la

linea di selezione del demux [1:4] corrispondente al bit più significativo. Le uscite di entrambi i demux [1:2] sono riportate in uscita al demux [1:4]

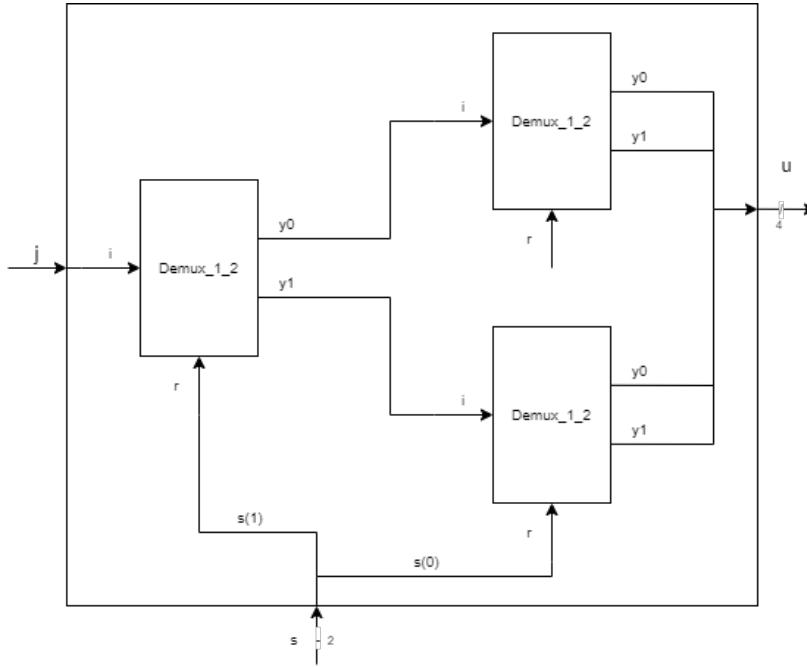


Figura 6: Struttura del demultiplexer [1:4]

Rete di interconnessione [16:4]

Realizzate tutte le componenti necessarie per la nostra rete di interconnessione [16:4] passiamo ora a definirla. In figura 7 è illustrata la struttura della nostra rete, abbiamo 16 sorgenti in ingresso (vettore $a[16]$) che entrano direttamente nel multiplexer e 4 uscite (vettore $y[4]$) che sono le uscite del demultiplexer. Le linee di selezione, invece, sono 6, di cui 4 servono per selezionare la linea di ingresso del multiplexer (vettore $s_mux[4]$) e 2 selezionano l'uscita del demultiplexer (vettore $s_demux[2]$). Internamente l'unico collegamento necessario è quello tra l'uscita del multiplexer e l'entrata del multiplexer, che è necessario per chiudere la linea di comunicazione.

1.2.3 Codice VHDL

Passiamo ora a presentare il codice VHDL che ci ha permesso di implementare le componenti viste nella precedente sezione.

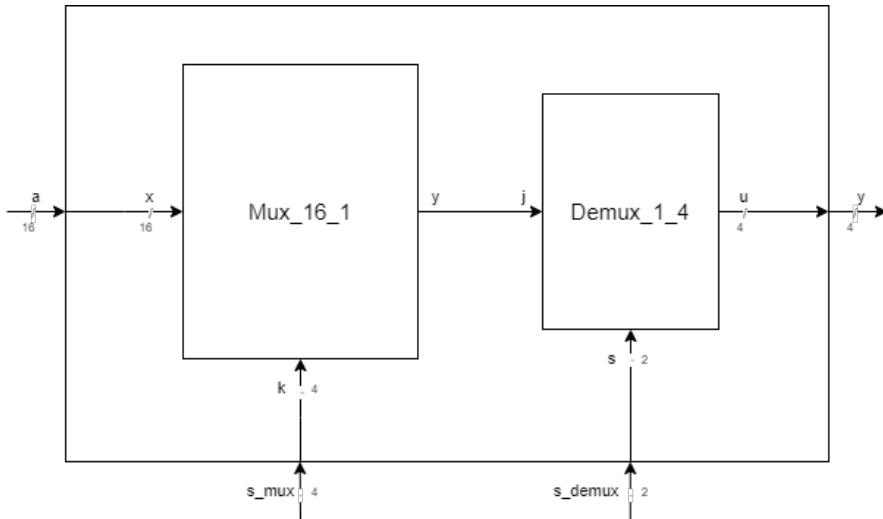


Figura 7: Rete di interconnessione [16:4]

Demux [1:2]

Il codice VHDL del Demux [1:2] è molto semplice. Gli ingressi e le uscite sono state definite come STD_LOGIC ed abbiamo descritto il comportamento del componente utilizzando il costrutto dataflow scrivendo le equazioni per le uscite del demux.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity demux_1_2 is
5     port(
6         i: in std_logic;
7         r: in std_logic;
8         y0: out std_logic;
9         y1: out std_logic
10    );
11 end demux_1_2;
12
13 architecture dataflow of demux_1_2 is
14 begin
15     y0 <= (not r) and i;
16     y1 <= r and i;
17 end dataflow;

```

Demux [1:4]

Per il Demux [1:4] è stato usato un approccio differente, infatti è stato definito attraverso il costrutto structural, specificando quindi la struttura interna del componente usando tre demux [1:2] e specificando i loro collegamenti. Anche gli ingressi e le uscite del Demux [1:4]

sono state definite come STD_LOGIC, in questo caso però gli ingressi di selezione e le uscite sono dei vettori. In particolare le selezioni sono numerate in modo inverso, questo semplifica l'interpretazione del funzionamento in quanto i bit in entrata rappresenterebbero un numero in binario che seleziona una specifica uscita. È di particolare interesse il segnale c , un vettore di due elementi, entrambi interni al Demux [1:4] che sono necessari per collegare i due livelli di Demux [1:2] internamente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity demux_1_4 is
5   port(
6     j: in std_logic;
7     s: in std_logic_vector(1 downto 0);
8     u: out std_logic_vector(0 to 3)
9   );
10 end demux_1_4;
11
12 architecture structural of demux_1_4 is
13 component demux_1_2 is
14   port(
15     i: in std_logic;
16     r: in std_logic;
17     y0: out std_logic;
18     y1: out std_logic
19   );
20 end component;
21
22 signal c: std_logic_vector(0 to 1);
23
24 begin
25   demux_0: demux_1_2
26     port map (
27       i => j,
28       r => s(1),
29       y0 => c(0),
30       y1 => c(1)
31     );
32   demux_1_0: demux_1_2
33     port map (
34       i => c(0),
35       r => s(0),
36       y0 => u(0),
37       y1 => u(1)
38     );
39   demux_1_1: demux_1_2
40     port map (
41       i => c(1),
42       r => s(0),
43       y0 => u(2),
44       y1 => u(3)
45     );
46 end structural;
```

Rete di interconnessione [16:4]

Arriviamo infine al rete di interconnessione in cui semplicemente connettiamo il Mux [16:1] al Demux [1:4] attraverso un segnale interno c . La rete ha quindi 16 sorgenti in ingresso e 4 uscite, inoltre i sei ingressi di selezione sono partizionati in due vettori, quattro per il Mux e due per il Demux. Naturalmente anche qui i vettori delle selezioni

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity conn_16_4 is
5   port (
6     a: in std_logic_vector(0 to 15);
7     s_mux: in std_logic_vector(3 downto 0);
8     s_demux: in std_logic_vector(1 downto 0);
9     y: out std_logic_vector(0 to 3)
10    );
11 end conn_16_4;
12
13 architecture structural of conn_16_4 is
14 component mux_16_1 is
15   port (
16     x: in std_logic_vector(0 to 15);
17     k: in std_logic_vector(3 downto 0);
18     y: out std_logic
19   );
20 end component;
21
22 component demux_1_4 is
23   port (
24     j: in std_logic;
25     s: in std_logic_vector(1 downto 0);
26     u: out std_logic_vector(0 to 3)
27   );
28 end component;
29
30 signal c: std_logic;
31
32 begin
33   mux: mux_16_1
34     port map (
35       x(0 to 15) => a(0 to 15),
36       k(3 downto 0) => s_mux(3 downto 0),
37       y => c
38     );
39   demux: demux_1_4
40     port map (
41       j => c,
42       s(1 downto 0) => s_demux(1 downto 0),
43       u(0 to 3) => y(0 to 3)
44     );
45 end structural;
```

1.2.4 Simulazione

Come richiesto dalla traccia è stato eseguito anche il testing del componente, dove sono stati testati 3 casi:

- input = ("0000000100000000"), s_mux=("0111"), s_demux=("11"), assert su output = "0001"
- input = ("0100000100010000"), s_mux=("0001"), s_demux=("01"), assert su output = "0100"
- input = ("0000000100010000"), s_mux=("0001"), s_demux=("01"), assert su output = "0000"

Di seguito il codice del testbech.

```

1 entity conn_TB is
2 end conn_TB;
3
4 architecture Behavioral of conn_TB is
5 component conn_16_4 is
6   port (
7     a: in std_logic_vector(0 to 15);
8     s_mux: in std_logic_vector(3 downto 0);
9     s_demux: in std_logic_vector(1 downto 0);
10    y: out std_logic_vector(0 to 3)
11  );
12 end component;
13 signal inputs: std_logic_vector(0 to 15) := (others => 'U');
14 signal selections_m: std_logic_vector(3 downto 0):= (others => 'U');
15 signal selections_d: std_logic_vector(1 downto 0):= (others => 'U');
16 signal outputs: std_logic_vector(0 to 3):= (others => 'U');
17 begin
18 uut: conn_16_4
19   port map (
20     a => inputs,
21     s_mux => selections_m,
22     s_demux => selections_d,
23     y => outputs
24   );
25
26 stim_proc: process
27 begin
28
29 wait for 100 ns;
30
31 -- A7=1 to Y3
32 wait for 10 ns;
33 inputs <= "0000000100000000";
34 selections_m <= "0111";
35 selections_d <= "11";
36 wait for 10 ns;
37 assert outputs = "0001"

```

```

38 report "errore caso 0"
39 severity failure;
40
41 -- A1=1 to Y1
42
43 wait for 10 ns;
44 inputs <= "0100000100010000";
45 selections_m <= "0001";
46 selections_d <= "01";
47 wait for 10 ns;
48 assert outputs = "0100"
49 report "errore caso 1"
50 severity failure;
51
52 -- A1=0 to Y1
53
54 wait for 10 ns;
55 inputs <= "0000000100010000";
56 selections_m <= "0001";
57 selections_d <= "01";
58 wait for 10 ns;
59 assert outputs = "0000"
60 report "errore caso 2"
61 severity failure;
62
63 wait;
64
65 end process;
66 end Behavioral;

```

1.3 Implementazione su board della rete di interconnessione

1.3.1 Traccia

Sintetizzare ed implementare su board il progetto della rete di interconnessione sviluppato al punto 1.2, utilizzando gli switch per fornire gli input di selezione e i led per visualizzare i 4 bit di uscita. Per quanto riguarda i 16 bit dato in input, essi possono essere precaricati nel sistema oppure immessi anch'essi mediante switch, sviluppando in questo secondo caso un'apposita rete di controllo per l'acquisizione.

1.3.2 Schematici

Per la soluzione di questo punto dell'esercizio è stato necessario sviluppare un'unità di controllo in gradi di disciplinare l'acquisizione dei valori di input della connessione e le selezioni. A supporto del meccanismi di acquisizione saranno usati due bottoni, alla pressione del primo acquisiremo gli input dagli switch, mentre al pressione del secondo acquisiremo le selezioni sempre dagli switch.

Unità di controllo

L'unità di controllo è molto semplice, necessita in input i 16 valori degli switch, i due bottoni, oltre a naturalmente il clock e reset perché è una macchina sequenziale. In uscita avremo i valori di input alla connessione e i valori delle selezioni della connessione. In figura 8 lo schematico della rete di controllo.

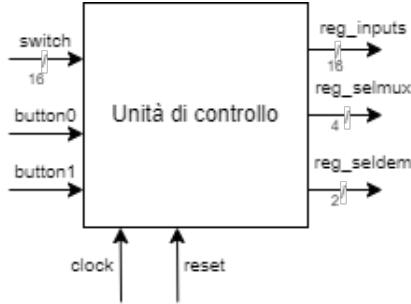


Figura 8: Unità di controllo per l'acquisizione dei valori

Rete di interconnessione su board

Per implementare la rete di interconnessione su board è quindi necessario comporre le due componenti e fare in modo che sia l'unità di controllo a fornire gli ingressi delle reti di interconnessione. Quindi basta unire tramite composizione unità di controllo e rete di interconnessione. In figura 9 lo schematico del componente che verrà implementato su board.

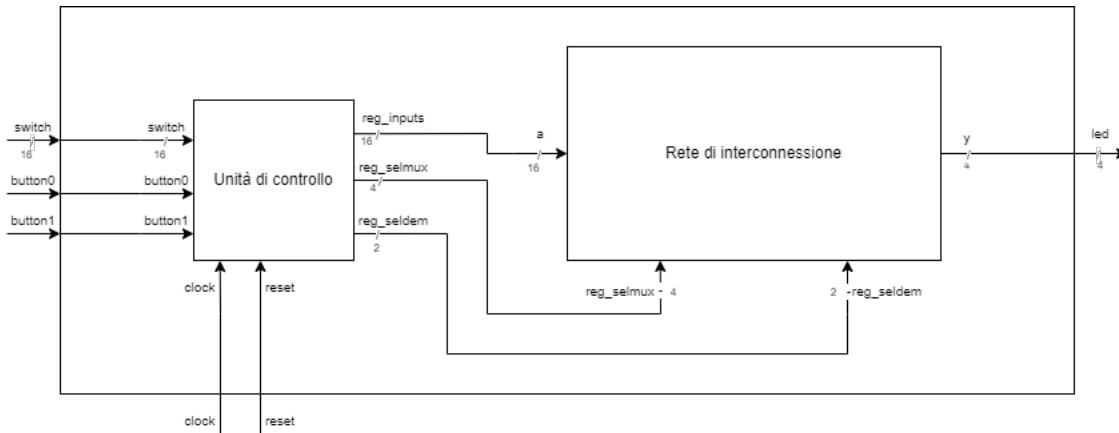


Figura 9: Rete di interconnessione su board

1.3.3 Codice VHDL

Unità di controllo

L'unità di controllo è stata implementata utilizzando il paradigma di progettazione behavioural. Sul fronte di salita del clock, se uno dei buttoni è premuto prelievo i valori degli switch e li metto in uscita. L'uscita su cui metto i valori dipende dal bottone che è stato premuto. Inoltre abbiamo anche inserito un reset sincrono con il clock. Di seguito il codice.

```

1 entity unita_di_controllo is port (
2     switch      : in std_logic_vector (0 to 15);
3     button_0    : in std_logic;      -- button [0] per i selettori button [1] per i dati
4     button_1    : in std_logic;
5     reset       : in std_logic;
6     clock       : in std_logic;
7     reg_inputs  : out std_logic_vector (0 to 15) := (others => '0');
8     reg_selmux : out std_logic_vector (3 downto 0) := "0000";
9     reg_seldem : out std_logic_vector (1 downto 0) := "00"
10 );
11 end unita_di_controllo;
12
13 architecture Behavioral of unita_di_controllo is
14 begin
15 main: process (clock)
16 begin
17
18 if (rising_edge(clock)) then
19     if (reset = '1') then
20         reg_inputs <= (others => '0');
21         reg_selmux <= (others => '0');
22         reg_seldem <= (others => '0');
23     end if;
24     if (button_0= '1') then
25         reg_selmux (3) <= switch (15);
26         reg_selmux (2) <= switch (14);
27         reg_selmux (1) <= switch (13);
28         reg_selmux (0) <= switch (12);
29         reg_seldem (1) <= switch (11);
30         reg_seldem (0) <= switch (10);
31     end if;
32     if (button_1= '1') then
33         reg_inputs <= switch (0 to 15);
34     end if;
35 end if;
36
37 end process;
38 end Behavioral;
```

Rete di interconnessione su board

Per la rete di interconnessione su board è stato usato uno structural per mettere insieme e collegare unità di controllo e rete di interconnessione che figurano, rispettivamente, come

parte di controllo e parte operativa di un sistema. Di seguito il codice.

```

1 entity macchina is port (
2 switch      : in std_logic_vector (0 to 15);
3 button_0    : in std_logic;      -- button [0] per i selettori button [1] per i dati
4 button_1    : in std_logic;
5 reset       : in std_logic;
6 clock       : in std_logic;
7 led         : out std_logic_vector (0 to 3)
8 );
9 end macchina;
10
11 architecture structural of macchina is
12 component conn_16_4 is port (
13 a: in std_logic_vector(0 to 15);
14 s_mux: in std_logic_vector(3 downto 0);
15 s_demux: in std_logic_vector(1 downto 0);
16 y: out std_logic_vector(0 to 3)
17 );
18 end component;
19
20 component unita_di_controllo is port (
21 switch : in std_logic_vector (0 to 15);
22 button_0 : in std_logic;      -- button [0] per i selettori button [1] per i dati
23 button_1 : in std_logic;
24 reset   : in std_logic;
25 clock   : in std_logic;
26 reg_inputs : out std_logic_vector (15 downto 0) := (others => '0');
27 reg_selmux : out std_logic_vector (3 downto 0) := "0000";
28 reg_seldem : out std_logic_vector (1 downto 0) := "00"
29 );
30 end component;
31
32 signal inputs   : std_logic_vector (0 to 15);
33 signal selmux   : std_logic_vector (3 downto 0);
34 signal seldem   : std_logic_vector (1 downto 0);
35 begin
36
37 cu : unita_di_controllo port map (
38 switch      => switch,
39 button_0    => button_0,
40 button_1    => button_1,
41 reset       => reset,
42 clock       => clock,
43 reg_inputs  => inputs,
44 reg_selmux  => selmux,
45 reg_seldem  => seldem
46 );
47
48 po : conn_16_4 port map (
49 a           => inputs,
50 s_mux       => selmux,
51 s_demux    => seldem,
52 y           => led
53 );
54
55 end structural;
```

Successivamente abbiamo scritto dei constraint per poter collegare i nostri ingressi ed uscite agli elementi su board. Oltre a definire un clock di periodo 10 ns, abbiamo collegato gli switch alla porta switch del componente, i led alla porta di uscita led del componente e 3 bottoni, uno per il reset e due per gestore l'acquisizione dei valori degli switch. Di seguito i constraint scritti.

```

1 ## Clock signal
2 set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { clock }]; #
3   IO_L12P_T1_MRCC_35 Sch=clk100mhz
4 create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clock}];
5
6 ##Switches
7 set_property -dict { PACKAGE_PIN J15     IOSTANDARD LVCMOS33 } [get_ports { switch[0] }]; #
8   IO_L24N_T3_RS0_15 Sch=sw[0]
9 set_property -dict { PACKAGE_PIN L16     IOSTANDARD LVCMOS33 } [get_ports { switch[1] }]; #
10  IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
11 set_property -dict { PACKAGE_PIN M13     IOSTANDARD LVCMOS33 } [get_ports { switch[2] }]; #
12  IO_L6N_T0_D08_VREF_14 Sch=sw[2]
13 set_property -dict { PACKAGE_PIN R15     IOSTANDARD LVCMOS33 } [get_ports { switch[3] }]; #
14  IO_L13N_T2_MRCC_14 Sch=sw[3]
15 set_property -dict { PACKAGE_PIN R17     IOSTANDARD LVCMOS33 } [get_ports { switch[4] }]; #
16  IO_L12N_T1_MRCC_14 Sch=sw[4]
17 set_property -dict { PACKAGE_PIN T18     IOSTANDARD LVCMOS33 } [get_ports { switch[5] }]; #
18  IO_L7N_T1_D10_14 Sch=sw[5]
19 set_property -dict { PACKAGE_PIN U18     IOSTANDARD LVCMOS33 } [get_ports { switch[6] }]; #
20  IO_L17N_T2_A13_D29_14 Sch=sw[6]
21 set_property -dict { PACKAGE_PIN R13     IOSTANDARD LVCMOS33 } [get_ports { switch[7] }]; #
22  IO_L5N_T0_D07_14 Sch=sw[7]
23 set_property -dict { PACKAGE_PIN T8      IOSTANDARD LVCMOS18 } [get_ports { switch[8] }]; #
24  IO_L24N_T3_34 Sch=sw[8]
25 set_property -dict { PACKAGE_PIN U8      IOSTANDARD LVCMOS18 } [get_ports { switch[9] }]; #
26  IO_25_34 Sch=sw[9]
27 set_property -dict { PACKAGE_PIN R16     IOSTANDARD LVCMOS33 } [get_ports { switch[10] }]; #
28  IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
29 set_property -dict { PACKAGE_PIN T13     IOSTANDARD LVCMOS33 } [get_ports { switch[11] }]; #
30  IO_L23P_T3_A03_D19_14 Sch=sw[11]
31 set_property -dict { PACKAGE_PIN H6      IOSTANDARD LVCMOS33 } [get_ports { switch[12] }]; #
32  IO_L24P_T3_35 Sch=sw[12]
33 set_property -dict { PACKAGE_PIN U12     IOSTANDARD LVCMOS33 } [get_ports { switch[13] }]; #
34  IO_L20P_T3_A08_D24_14 Sch=sw[13]
35 set_property -dict { PACKAGE_PIN U11     IOSTANDARD LVCMOS33 } [get_ports { switch[14] }]; #
36  IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
37 set_property -dict { PACKAGE_PIN V10    IOSTANDARD LVCMOS33 } [get_ports { switch[15] }]; #
38  IO_L21P_T3_DQS_14 Sch=sw[15]
39
40 ## LEDs
41 set_property -dict { PACKAGE_PIN H17    IOSTANDARD LVCMOS33 } [get_ports { led[0] }]; #
42   IO_L18P_T2_A24_15 Sch=led[0]
43 set_property -dict { PACKAGE_PIN K15    IOSTANDARD LVCMOS33 } [get_ports { led[1] }]; #
44   IO_L24P_T3_RS1_15 Sch=led[1]
45 set_property -dict { PACKAGE_PIN J13    IOSTANDARD LVCMOS33 } [get_ports { led[2] }]; #
46   IO_L17N_T2_A25_15 Sch=led[2]
47 set_property -dict { PACKAGE_PIN N14    IOSTANDARD LVCMOS33 } [get_ports { led[3] }]; #
48   IO_L8P_T1_D11_14 Sch=led[3]
```

```
30 ##Buttons
31 set_property -dict { PACKAGE_PIN N17    IOSTANDARD LVCMOS33 } [get_ports { reset }]; #
32           IO_L9P_T1_DQS_14 Sch=btnc
32 set_property -dict { PACKAGE_PIN P17    IOSTANDARD LVCMOS33 } [get_ports { button_1 }]; #
33           IO_L12P_T1_MRCC_14 Sch=btnl
33 set_property -dict { PACKAGE_PIN M17    IOSTANDARD LVCMOS33 } [get_ports { button_0 }]; #
34           IO_L10N_T1_D15_14 Sch=btnr
```

2 Encoder

In questo capitolo viene affrontato il problema della creazione di un encoder [16:4], l'encoder è stato prima sviluppato in vhdl (parte 1), poi sintetizzato su board (parte 2), infine alla sintesi su board è stata aggiunta la lettura su display.

2.1 Progettazione in VHDL

Traccia

Progettare, implementare in VHDL e testare mediante simulazione una rete che, data in ingresso una stringa binaria X di 10 bit $X_9 X_8 X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0$ che corrisponde alla rappresentazione decodificata di una cifra decimale (cioè, una rappresentazione in cui ogni stringa contiene un solo bit alto), fornisce in uscita la rappresentazione Y della cifra mediante codifica Binary-Coded Decimal (BCD).

2.1.1 Schematici

Per implementare un Encoder BCD è stato prima necessario comprendere il suo funzionamento nel dettaglio. In ingresso all'encoder vi è una stringa di 10 bit composta da tutti 0 ed un solo 1, la posizione di questo 1 all'interno della stringa rappresenta il numero decimale da codificare in binario e porre in uscita all'encoder. Questo comportamento è stato implementato utilizzando un componente che presenta la struttura riportata nella seguente schematica:

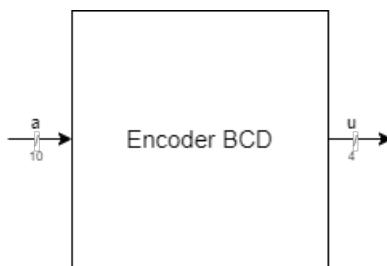


Figura 10: Encoder BCD

con a la stringa di 10 bit in ingresso, mentre u è la codifica binaria corrispondente alla stringa in ingresso.

2.1.2 Codice VHDL

Passiamo ora a presentare il codice VHDL che ci ha permesso di implementare il comportamento del componente visto nella precedente sezione.

Abbiamo deciso di implementare l'Encoder BCD in modo Dataflow, quindi abbiamo descritto la tabella della verità che riproduce il comportamento desiderato tramite il costrutto *when-else* ed utilizzando una particolare istanza del vettore di ingresso per selezionare una specifica sequenza di uscita. Di seguito il codice completo:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Encoder_BCD is
5     port (
6         a : in std_logic_vector(9 downto 0);
7         u : out std_logic_vector(3 downto 0)
8     );
9 end Encoder_BCD;
10
11 architecture Dataflow of Encoder_BCD is
12 begin
13     u <= "0000" when a = "0000000001" else
14         "0001" when a = "0000000010" else
15         "0010" when a = "0000000100" else
16         "0011" when a = "0000001000" else
17         "0100" when a = "0000010000" else
18         "0101" when a = "0000100000" else
19         "0110" when a = "0001000000" else
20         "0111" when a = "0010000000" else
21         "1000" when a = "0100000000" else
22         "1001" when a = "1000000000" else
23         "----";
24 end Dataflow;

```

2.1.3 Simulazione

Come richiesto dalla traccia è stato eseguito anche il testing del componente. In particolare abbiamo deciso di controllare il comportamento dell'Encoder in 3 casi di test:

- inputs = ("0010000000"), assert su outputs = "0111"
- inputs = ("0000000100"), assert su outputs = "0010"
- inputs = ("0000100000"), assert su outputs = "0101"

Di seguito il codice del testbench:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Encoder_BCD_tb is
5 end Encoder_BCD_tb;
6
7 architecture Behavioral of Encoder_BCD_tb is
8 component Encoder_BCD is port(
9     a : in std_logic_vector(9 downto 0);
10    u : out std_logic_vector(3 downto 0)
11 );
12 end component;
13
14 signal inputs : std_logic_vector(9 downto 0) := (others => 'U');
15 signal outputs : std_logic_vector(3 downto 0) := (others => 'U');
16
17 begin
18 uut: Encoder_BCD
19     port map (
20         a => inputs,
21         u => outputs
22     );
23 stim_proc: process
24 begin
25
26 wait for 100 ns;
27 -- a7
28 wait for 10 ns;
29 inputs <= "0010000000";
30 wait for 10 ns;
31 assert outputs = "0111"
32 report "errore caso 1"
33 severity failure;
34
35 -- a2
36 wait for 10 ns;
37 inputs <= "0000000100";
38 wait for 10 ns;
39 assert outputs = "0010"
40 report "errore caso 2"
41 severity failure;
42
43 -- a5
44 wait for 10 ns;
45 inputs <= "0000100000";
46 wait for 10 ns;
47 assert outputs = "0101"
48 report "errore caso 3"
49 severity failure;
50
51 wait;
52 end process;
53 end Behavioral;

```

2.2 Implementazione su board

Traccia

Sintetizzare ed implementare su board il progetto dell'encoder BCD utilizzando gli switch per fornire la stringa X in ingresso, e i led per visualizzare Y. Nel caso in cui si utilizzi una board dotata di soli 8 switch, è possibile sviluppare il progetto considerando X di soli 8 bit (la macchina sarà allora in grado di fornire in uscita la rappresentazione BCD delle cifre decimali da 0 a 7).

2.2.1 Soluzione

Per implementare su board l'Encoder non è stato necessario definire un ulteriore componente wrapper per gestire l'acquisizione degli input. Infatti gli input sono solamente 10, quindi è possibile collegarli direttamente agli switch (che sono 16), mentre l'uscita, che è su 4 bit, si può collegare direttamente a 4 led su board. Quindi, dato il componente visto la sezione precedente, è stato necessario solamente utilizzare i constraint per collegare ingressi ed uscite al elementi sulla board. Di seguito i constraint utilizzati.

```

1 ##Switches
2 set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports { a[0] }]; #
   IO_L24N_T3_RS0_15 Sch=sw[0]
3 set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports { a[1] }]; #
   IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
4 set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 } [get_ports { a[2] }]; #
   IO_L6N_T0_D08_VREF_14 Sch=sw[2]
5 set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 } [get_ports { a[3] }]; #
   IO_L13N_T2_MRCC_14 Sch=sw[3]
6 set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 } [get_ports { a[4] }]; #
   IO_L12N_T1_MRCC_14 Sch=sw[4]
7 set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMOS33 } [get_ports { a[5] }]; #
   IO_L7N_T1_D10_14 Sch=sw[5]
8 set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMOS33 } [get_ports { a[6] }]; #
   IO_L17N_T2_A13_D29_14 Sch=sw[6]
9 set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMOS33 } [get_ports { a[7] }]; #
   IO_L5N_T0_D07_14 Sch=sw[7]
10 set_property -dict { PACKAGE_PIN T8      IOSTANDARD LVCMOS18 } [get_ports { a[8] }]; #
   IO_L24N_T3_34 Sch=sw[8]
11 set_property -dict { PACKAGE_PIN U8      IOSTANDARD LVCMOS18 } [get_ports { a[9] }]; #IO_25_34
   Sch=sw[9]
12
13 ## LEDs
14 set_property -dict { PACKAGE_PIN H17      IOSTANDARD LVCMOS33 } [get_ports { u[0] }]; #
   IO_L18P_T2_A24_15 Sch=led[0]
15 set_property -dict { PACKAGE_PIN K15      IOSTANDARD LVCMOS33 } [get_ports { u[1] }]; #
   IO_L24P_T3_RS1_15 Sch=led[1]
16 set_property -dict { PACKAGE_PIN J13      IOSTANDARD LVCMOS33 } [get_ports { u[2] }]; #
   IO_L17N_T2_A25_15 Sch=led[2]
```

```
17 | set_property -dict { PACKAGE_PIN N14      IO_STANDARD LVCMOS33 } [get_ports { u[3] }]; #
     | IO_L8P_T1_D11_14 Sch=led[3]
```

2.3 Implementazione su board con display

Traccia

Utilizzare un display a 7 segmenti per visualizzare la cifra decimale codificata da Y (pilotare opportunamente i catodi del display per visualizzare la cifra).

2.3.1 Schematici

Come richiesto dalla traccia è stato preso l'esercizio svolto nei punti precedenti ed è stato implementato sul display a 7 segmenti presente sulla scheda FPGA Artyx A7-50T.

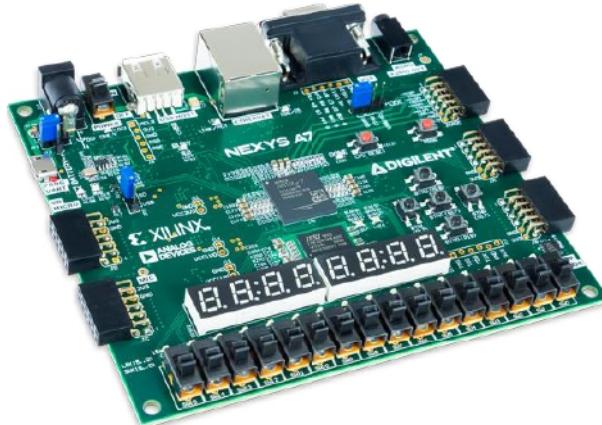


Figura 11: Scheda FPGA Artyx A7-50T

Display

Prima di collegare, tramite composizione, il display all'encoder è stato necessario creare un piccolo blocco che gestisce il suo funzionamento. A causa dei collegamenti fisici all'interno della scheda il display non può mostrare numeri diversi, infatti tutte le cifre del display hanno i catodi in comune e gli anodi separati, tutti funzionanti in logica negata (quindi per accendere una cifra è necessario abbassare il suo anodo e la cifra mostrata è quella indicata dai catodi abbassati). È possibile vedere i collegamenti elettrici del display nella figura 12

Il componente implementato per permettere il funzionamento del display ha in input:

- *punto*, utilizzato per indicare se si vuole accendere il punto o meno

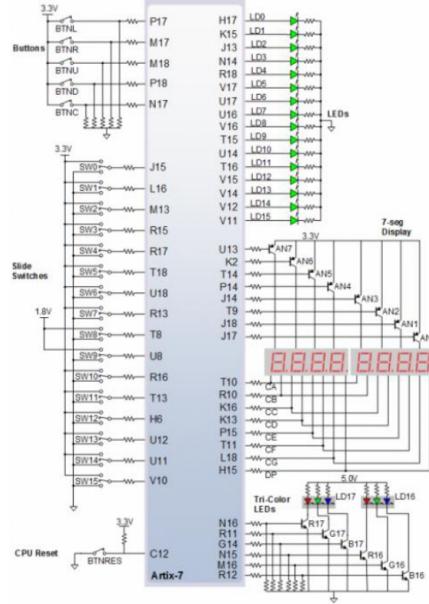


Figura 12: Collegamenti elettrici nel display

- $u[3 \text{ downto } 0]$, utilizzato per indicare il numero da rappresentare ed ha in uscita:
- $catodi[7 \text{ downto } 0]$, che viene utilizzato per modificare il valore dei catodi
- $anodi[7 \text{ downto } 0]$, che viene utilizzato per modificare il valore degli anodi

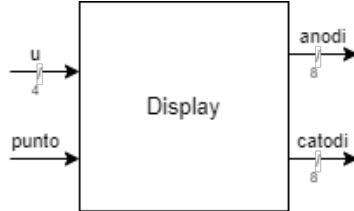


Figura 13: Componente Display

Encoder con display

Dopo aver implementato il componente del display si è passati al collegamento con l'encoder_BCD. Il collegamento è stato fatto tramite l'entity encoder_con_display, che attraverso un'architettura strutturale collega i 2 componenti. Affinché sia possibile permettere il corretto funzionamento della macchina si utilizzano gli ingressi:

- $a[9 \text{ downto } 0]$, che viene utilizzato per poter scegliere il numero da mostrare a display (sempre tramite gli switch)

le uscite:

- $u[3 \text{ downto } 0]$, che viene utilizzato per poter indicare tramite led quale numero è stato inserito (il segnale viene utilizzato anche per indicare quale numero mostrare sul display)
- $catodi[7 \text{ downto } 0]$, che viene utilizzato per poter scegliere lo stato dei catodi del display
- $anodi[7 \text{ downto } 0]$, che viene utilizzato per poter scegliere lo stato degli anodi del display

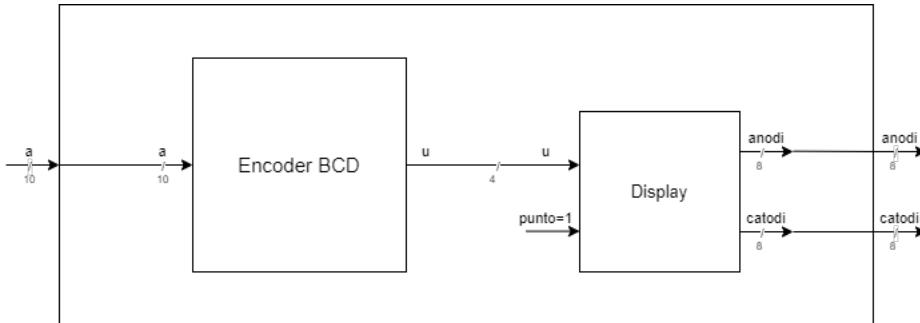


Figura 14: Encoder con display

2.3.2 Codice VHDL

display

Il componente è stato sviluppato in dataflow, si utilizzano delle costanti per poter codificare le cifre in ingressi binari dei catodi e vengono utilizzati 2 segnali:

- $cifra_anodo[7 \text{ downto } 0]$, che viene utilizzato per accendere soli la prima cifra sul display
- $numero[6 \text{ downto } 0]$, che viene utilizzato per poter rappresentare il numero in ingresso nei catodi

Il codice del componente aggiunto è:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity display is port (
5     punto : in std_logic;                                -- bit da mettere a 0 se si vuole
6     displayare_il_punto
7     u      : in std_logic_vector (3 downto 0); -- numero
8     catodi : out std_logic_vector (7 downto 0); -- a partire da: ca a dp
9     anodi  : out std_logic_vector (7 downto 0)  -- a partire da: an7 a an0
10 );
11 end display;
12
13 architecture dataflow of display is
14
15 constant zero   : std_logic_vector(6 downto 0) := "1000000";
16 constant one    : std_logic_vector(6 downto 0) := "1111001";
17 constant two    : std_logic_vector(6 downto 0) := "0100100";
18 constant three  : std_logic_vector(6 downto 0) := "0110000";
19 constant four   : std_logic_vector(6 downto 0) := "0011001";
20 constant five   : std_logic_vector(6 downto 0) := "0010010";
21 constant six    : std_logic_vector(6 downto 0) := "0000010";
22 constant seven  : std_logic_vector(6 downto 0) := "1111000";
23 constant eight  : std_logic_vector(6 downto 0) := "0000000";
24 constant nine   : std_logic_vector(6 downto 0) := "0010000";
25 constant e       : std_logic_vector(6 downto 0) := "0000110";
26
27 signal cifra_anodo : std_logic_vector (7 downto 0) := "01111111";
28 signal numero   : std_logic_vector (6 downto 0);
29
30 begin
31     numero <= zero  when u = "0000" else
32         one   when u = "0001" else
33         two   when u = "0010" else
34         three when u = "0011" else
35         four  when u = "0100" else
36         five  when u = "0101" else
37         six   when u = "0110" else
38         seven when u = "0111" else
39         eight when u = "1000" else
40         nine  when u = "1001" else
41             e;
42     catodi <= (not punto) & numero;
43     anodi  <= cifra_anodo;
44 end dataflow;

```

encoder _ con _ display

Il componente viene utilizzato per collegare il componente "encoder" precedentemente creato con il nuovo componente "display", per questo motivo è stato sviluppato su un'architettura strutturale. Per l'unione dei 2 elementi è stato necessario la creazione di 2 segnali interni:

- *punto*, che viene utilizzato per indicare se si vuole o meno mostrare il punto vicino al numero sul display;

- *cifra[3downto0]*, che viene utilizzato per collegare l'uscita del componente "encoder" con l'ingresso del componente "display".

Il codice di "encoder_con_display" è:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity encoder_con_display is port (
5     a : in std_logic_vector(9 downto 0);
6     u : out std_logic_vector(3 downto 0);
7     catodi : out std_logic_vector (7 downto 0); -- a partire da: ca a dp
8     anodi : out std_logic_vector (7 downto 0) -- a partire da: an7 a an0
9 );
10
11 end encoder_con_display;
12
13 architecture structural of encoder_con_display is
14
15 component display port (
16     punto : in std_logic;                                -- bit da mettere a 0 se si vuole
17     displayare il punto
18     u : in std_logic_vector (3 downto 0); -- numero
19     catodi : out std_logic_vector (7 downto 0); -- a partire da: ca a dp
20     anodi : out std_logic_vector (7 downto 0) -- a partire da: an7 a an0
21 );
22 end component;
23
24 component Encoder_BCD port (
25     a : in std_logic_vector(9 downto 0);
26     u : out std_logic_vector(3 downto 0)
27 );
28 end component;
29
30 signal punto : std_logic := '1';
31 signal cifra : std_logic_vector (3 downto 0);
32
33 begin
34
35 display_0 : display PORT MAP(
36     punto      => punto,
37     u          => cifra,
38     catodi    => catodi,
39     anodi    => anodi
40 );
41
42
43 encoder : Encoder_BCD PORT MAP (
44     a => a,
45     u => cifra
46 );
47 u <= cifra;
48
49 end structural;

```

Infine è stato necessario solamente collegare il componente sviluppato agli elementi su board utilizzando opportuni costrain. Abbiamo mantenuto l'uscita dell'encoder anche sui led, oltre a visualizzarla sul display. Di seguito i costrain utilizzati.

```

1 ##Switches
2 set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports { a[0] }]; #
3   IO_L24N_T3_RS0_15 Sch=sw[0]
4 set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports { a[1] }]; #
5   IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
6 set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 } [get_ports { a[2] }]; #
7   IO_L6N_T0_D08_VREF_14 Sch=sw[2]
8 set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 } [get_ports { a[3] }]; #
9   IO_L13N_T2_MRCC_14 Sch=sw[3]
10 set_property -dict { PACKAGE_PIN R17     IOSTANDARD LVCMOS33 } [get_ports { a[4] }]; #
11   IO_L12N_T1_MRCC_14 Sch=sw[4]
12 set_property -dict { PACKAGE_PIN T18     IOSTANDARD LVCMOS33 } [get_ports { a[5] }]; #
13   IO_L7N_T1_D10_14 Sch=sw[5]
14 set_property -dict { PACKAGE_PIN U18     IOSTANDARD LVCMOS33 } [get_ports { a[6] }]; #
15   IO_L17N_T2_A13_D29_14 Sch=sw[6]
16 set_property -dict { PACKAGE_PIN R13     IOSTANDARD LVCMOS33 } [get_ports { a[7] }]; #
17   IO_L5N_T0_D07_14 Sch=sw[7]
18 set_property -dict { PACKAGE_PIN T8      IOSTANDARD LVCMOS18 } [get_ports { a[8] }]; #
19   IO_L24N_T3_34 Sch=sw[8]
20 set_property -dict { PACKAGE_PIN U8      IOSTANDARD LVCMOS18 } [get_ports { a[9] }]; #IO_25_34
21   Sch=sw[9]
22
23 ## LEDs
24 set_property -dict { PACKAGE_PIN H17     IOSTANDARD LVCMOS33 } [get_ports { u[0] }]; #
25   IO_L18P_T2_A24_15 Sch=led[0]
26 set_property -dict { PACKAGE_PIN K15     IOSTANDARD LVCMOS33 } [get_ports { u[1] }]; #
27   IO_L24P_T3_RS1_15 Sch=led[1]
28 set_property -dict { PACKAGE_PIN J13     IOSTANDARD LVCMOS33 } [get_ports { u[2] }]; #
29   IO_L17N_T2_A25_15 Sch=led[2]
30 set_property -dict { PACKAGE_PIN N14     IOSTANDARD LVCMOS33 } [get_ports { u[3] }]; #
31   IO_L8P_T1_D11_14 Sch=led[3]
32
33 ###7 segment display
34 set_property -dict { PACKAGE_PIN T10     IOSTANDARD LVCMOS33 } [get_ports { catodi[0] }]; #
35   IO_L24N_T3_A00_D16_14 Sch=ca
36 set_property -dict { PACKAGE_PIN R10     IOSTANDARD LVCMOS33 } [get_ports { catodi[1] }]; #
37   IO_25_14 Sch=cb
38 set_property -dict { PACKAGE_PIN K16     IOSTANDARD LVCMOS33 } [get_ports { catodi[2] }]; #
39   IO_25_15 Sch=cc
40 set_property -dict { PACKAGE_PIN K13     IOSTANDARD LVCMOS33 } [get_ports { catodi[3] }]; #
41   IO_L17P_T2_A26_15 Sch=cd
42 set_property -dict { PACKAGE_PIN P15     IOSTANDARD LVCMOS33 } [get_ports { catodi[4] }]; #
43   IO_L13P_T2_MRCC_14 Sch=ce
44 set_property -dict { PACKAGE_PIN T11     IOSTANDARD LVCMOS33 } [get_ports { catodi[5] }]; #
45   IO_L19P_T3_A10_D26_14 Sch=cf
46 set_property -dict { PACKAGE_PIN L18     IOSTANDARD LVCMOS33 } [get_ports { catodi[6] }]; #
47   IO_L4P_T0_D04_14 Sch=cg
48 set_property -dict { PACKAGE_PIN H15     IOSTANDARD LVCMOS33 } [get_ports { catodi[7] }]; #
49   IO_L19N_T3_A21_VREF_15 Sch=dp
50 set_property -dict { PACKAGE_PIN J17     IOSTANDARD LVCMOS33 } [get_ports { anodi[0] }]; #
51   IO_L23P_T3_FOE_B_15 Sch=an[0]
52 set_property -dict { PACKAGE_PIN J18     IOSTANDARD LVCMOS33 } [get_ports { anodi[1] }]; #
53   IO_L23N_T3_FWE_B_15 Sch=an[1]

```

```
30 | set_property -dict { PACKAGE_PIN T9      IOSTANDARD LVCMOS33 } [get_ports { anodi[2] }]; #
  |   IO_L24P_T3_A01_D17_14 Sch=an[2]
31 | set_property -dict { PACKAGE_PIN J14      IOSTANDARD LVCMOS33 } [get_ports { anodi[3] }]; #
  |   IO_L19P_T3_A22_15 Sch=an[3]
32 | set_property -dict { PACKAGE_PIN P14      IOSTANDARD LVCMOS33 } [get_ports { anodi[4] }]; #
  |   IO_L8N_T1_D12_14 Sch=an[4]
33 | set_property -dict { PACKAGE_PIN T14      IOSTANDARD LVCMOS33 } [get_ports { anodi[5] }]; #
  |   IO_L14P_T2_SRCC_14 Sch=an[5]
34 | set_property -dict { PACKAGE_PIN K2      IOSTANDARD LVCMOS33 } [get_ports { anodi[6] }]; #
  |   IO_L23P_T3_35 Sch=an[6]
35 | set_property -dict { PACKAGE_PIN U13      IOSTANDARD LVCMOS33 } [get_ports { anodi[7] }]; #
  |   IO_L23N_T3_A02_D18_14 Sch=an[7]
```

3 Riconoscitore di sequenze

In questa sezione verrà affrontato il problema di riconoscere una sequenza in data in ingresso ad una macchina. Viene sviluppata per la prima volta una macchina a stati finiti in vhdl, utilizzata per poter gestire il riconoscimento, ed infine (nel secondo paragrafo) lo si implementa su board.

3.1 Progettazione in VHDL

Traccia

Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza 1001. La macchina prende in ingresso un segnale binario i che rappresenta il dato, un segnale A di 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 4,
- se $M=1$, la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta.

3.1.1 Schematici

Per lo svolgimento di questa traccia è stato sufficiente l'implementazione di un'automa a stati finiti per poter riconoscere la sequenza, di conseguenza l'unico componente utilizzato è stato: Automa_riconoscitore.

Automa_riconoscitore

Il componente è molto semplice, infatti attraverso gli ingressi (I, M) è possibile scegliere il modo ed il codice in ingresso, l'automa controlla ad ogni rising_edge di clock l'ingresso in attesa di riconoscere la sequenza 1001.

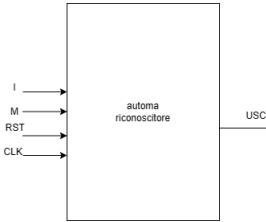


Figura 15: componente_automma_riconoscitore

3.1.2 Codice VHDL

automa_riconoscitore

L'automa è stato sviluppato tramite il costrutto behavioral con 2 process (uno per calcolare lo stato prossimo ed un altro per cambiare stato e tenerlo in memoria). L'automa sviluppato è formato da 11 stati, si parte da uno stato iniziale (in cui è possibile scegliere in che modo la macchina deve riconoscere la sequenza) e da lì si divide in 2 sottoautomi, che gestiscono il funzionamento della macchina. La macchina a stati finiti è presente in figura 16:

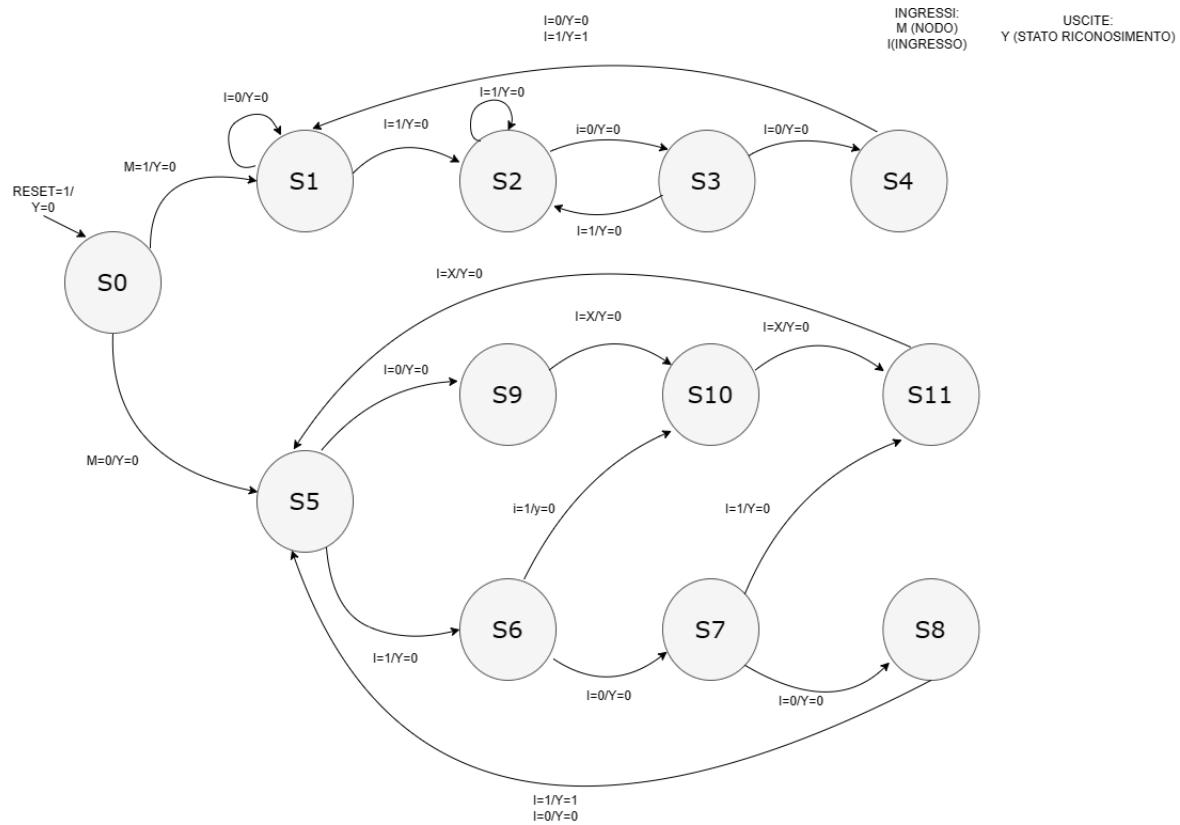


Figura 16: automma_riconoscitore

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Riconoscitore_2modi is
5 port(
6     i : in std_logic;
7     RST: in std_logic;
8     CLK: in std_logic;
9     M : in std_logic;
10    uscita : out std_logic
11 );
12 end Riconoscitore_2modi;
13
14 architecture Behavioral of Riconoscitore_2modi is
15
16 type stato is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11);
17
18 signal stato_corrente : stato := S0;
19 signal stato_prossimo : stato;
20 signal Y : std_logic;
21
22 begin
23
24 stato_uscita: process (i, stato_corrente)
25 begin
26
27 case stato_corrente is
28 when S0 =>
29     if (M = '0') then
30         stato_prossimo <= S5;
31         Y <= '0';
32     else
33         stato_prossimo <= S1;
34         Y <= '0';
35     end if;
36
37 when S1 =>
38     if (i = '1') then
39         stato_prossimo <= S2;
40         Y <= '0';
41     else
42         stato_prossimo <= S1;
43         Y <= '0';
44     end if;
45
46 when S2 =>
47     if (i = '1') then
48         stato_prossimo <= S2;
49         Y <= '0';
50     else
51         stato_prossimo <= S3;
52         Y <= '0';
53     end if;
54
55 when S3 =>
56     if (i = '1') then
57         stato_prossimo <= S2;
58         Y <= '0';
59     else

```

```

60         stato_prossimo <= S4;
61         Y <= '0';
62     end if;

63
64     when S4 =>
65         if (i = '1') then
66             stato_prossimo <= S1;
67             Y <= '1';
68         else
69             stato_prossimo <= S1;
70             Y <= '0';
71         end if;

72
73     when S5 =>
74         if (i = '1') then
75             stato_prossimo <= S6;
76             Y <= '0';
77         else
78             stato_prossimo <= S9;
79             Y <= '0';
80         end if;

81
82     when S6 =>
83         if (i = '0') then
84             stato_prossimo <= S7;
85             Y <= '0';
86         else
87             stato_prossimo <= S10;
88             Y <= '0';
89         end if;

90
91     when S7 =>
92         if (i = '0') then
93             stato_prossimo <= S8;
94             Y <= '0';
95         else
96             stato_prossimo <= S11;
97             Y <= '0';
98         end if;

99
100    when S8 =>
101        if (i = '0') then
102            stato_prossimo <= S5;
103            Y <= '0';
104        else
105            stato_prossimo <= S5;
106            Y <= '1';
107        end if;

108
109    when S9 =>
110        stato_prossimo <= S10;
111        Y <= '0';

112
113    when S10 =>
114        stato_prossimo <= S11;
115        Y <= '0';

116
117    when S11 =>
118        stato_prossimo <= S5;
119        Y <= '0';

```

```

120
121     end case;
122
123 end process;
124
125 registro: process (CLK)
126 begin
127     if(CLK'event and CLK='1') then
128         if(RST='1') then
129             stato_corrente <= S0;
130             uscita <= '0';
131         else
132             stato_corrente <= stato_prossimo;
133             uscita <= Y;
134
135         end if;
136     end if;
137
138 end process;
139
140 end Behavioral;

```

3.1.3 Simulazione

In simulazione sono stati testati entrambi i sottografi (una volta con la sequenza corretta e l'altra con la sequenza sbagliata). Gli ingressi provati sono stati:

- M=1, I=1,1,0,1,0,0,1
- M=1, I=1,0,0,0
- M=0, I=1,1,0,1
- M=0, I=1,0,0,1

Di seguito è presente il comportamento dell'automa osservato sul testbench di vivado.

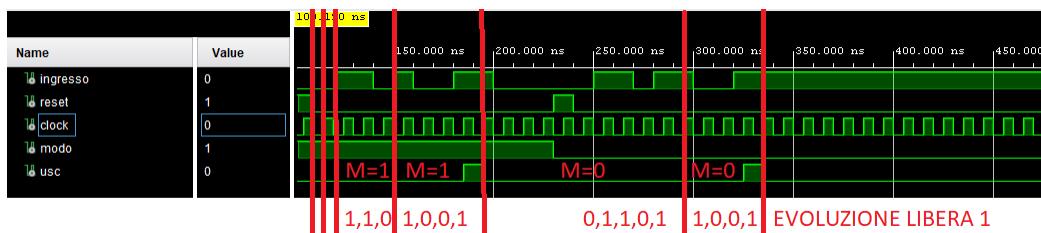


Figura 17: simulazione automa riconoscitore

3.2 Implementazione su board

Traccia

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

3.2.1 Schematici

Per il completamento di questo esercizio è stato necessario aggiungere altri 3 componenti:

- divisore_freq, utilizzato per gestire il sincronismo tra la pressione del tasto ed il segnale di tempificazione e richiesto esplicitamente dalla traccia;
- automa_debouncer, utilizzato per gestire il "rimbalzo" legato al bottone fisico presente su scheda;
- riconoscitore_su_board, che collega tutti i 3 componenti tra di loro.

divisore_freq

Il divisore di frequenza è un componente che rallenta la frequenza del clock, per poter svolgere questa funzione il divisore conta i colpi di clock in ingresso e restituisce in uscita '1' quando arriva ad un numero definito dall'utente, l'uscita rimane alta fino al prossimo rising edge del clock.

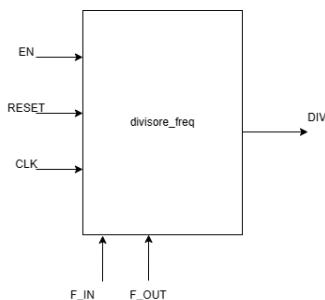


Figura 18: divisore_di_frequenza

automa_debouncer

Il componente `automa_debouncer` gestisce il problema del bottone presente su scheda, infatti quando il bottone viene premuto vengono registrate piú variazioni tra '0' e '1' prima di stabilizzarsi al valore '1'. Per ovviare a questo problema ad ogni pressione del bottone si attende un tempo definito dall'utente (definito attraverso il valore di `btn_noise_time`) e solo dopo questo tempo si vede se il bottone é stato premuto o rilasciato correttamente. In questo esercizio al componente é stato aggiunto un nuovo ingresso (`div`), che gestisce il sincronismo tra la pressione ed il divisore di frequenza (come richiesto dalla traccia).

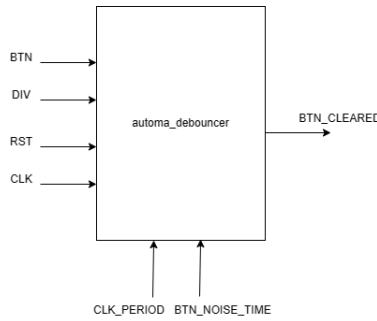


Figura 19: `componente_automa_debouncer`

riconoscitore_su_board

Il componente `riconoscitore_su_board` viene utilizzato per collegare gli input ai componenti, per metterli in comunicazione e permettere il normale funzionamento della macchina.

3.2.2 Codice VHDL

divisore_freq

Per permettere il normale funzionamento del divisore di frequenza, si calcola il numero di colpi di clock da contare, attraverso la funzione:

$$\text{max_value} = \frac{f_{in}}{f_{out}} - 1$$

Il comportamento del componente consiste nel contare quanti colpi di clock ci sono stati in ingresso attraverso l'aggiornamento di una variabile intera (conteggio). Quando si ha:

$$\text{conteggio} = \text{max_value}$$

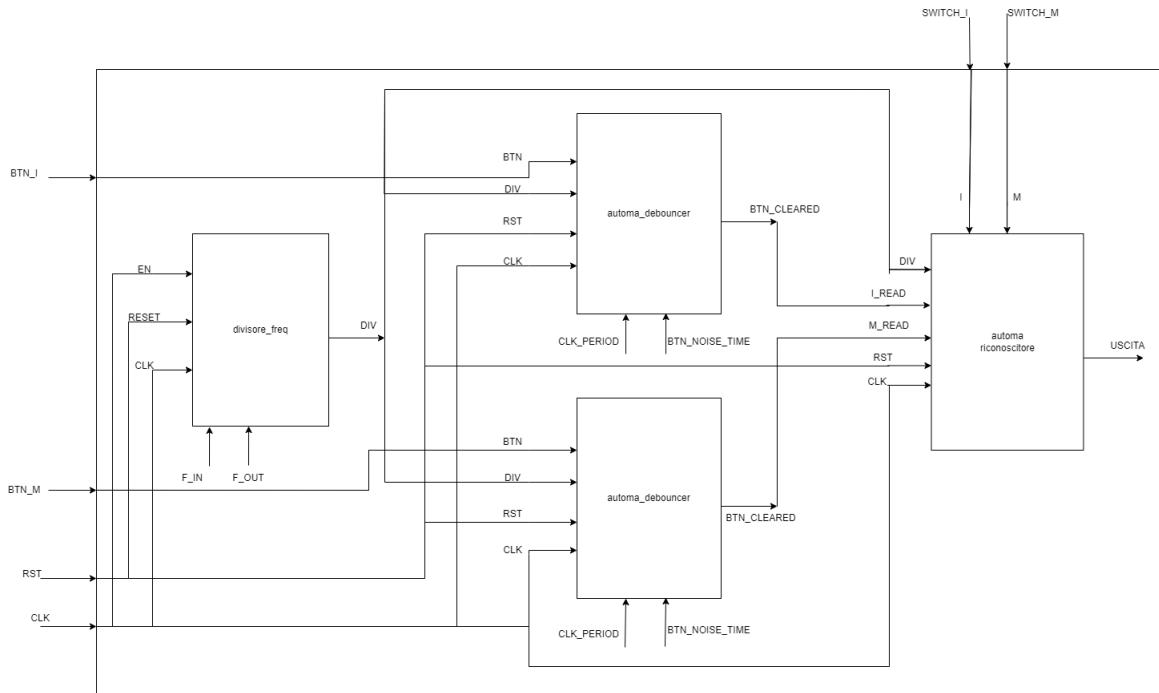


Figura 20: componente_riconoscitore_su_board

si mette 1 in uscita, si riazzera il conteggio e dal prossimo colpo di clock si ripete il comportamento specificato.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Divisore_freq is
5     generic (
6         f_in : integer := 100000000;
7         f_out : integer := 50000000
8     );
9     port (
10        reset : in std_logic;
11        clock : in std_logic;
12        div   : out std_logic
13    );
14 end Divisore_freq;
15
16 architecture Behavioral of Divisore_freq is
17
18 constant max_value : integer := (f_in/f_out)-1;
19
20 begin
21
22 process (clock)
23 variable conteggio : integer range 0 to max_value := 0;
24 begin
25
26 if (rising_edge(clock)) then

```

```

27   if (reset ='1') then
28     conteggio := 0;
29     div <= '0';
30   elsif (conteggio = max_value) then
31     conteggio := 0;
32     div <= '1';
33   else
34     conteggio := conteggio+1;
35     div <= '0';
36   end if;
37
38 end if;
39
40 end process;
41
42 end Behavioral;

```

automa_debouncer

Per permettere il normale funzionamento del componente automa_debouncer è stata sviluppata una macchina a stati finiti (di mealy), formata da 4 stati:

- non_premuto, che è lo stato in cui si attende che il bottone venga premuto;
- controllo_pressione, che è lo stato in cui si controlla se il bottone è stato effettivamente premuto e quindi se alzare o meno l'uscita;
- premuto, che è lo stato in cui si attende che il bottone venga rilasciato;
- controllo_rilascio, che è lo stato in cui si controlla se il bottone è stato correttamente rilasciato.

Il segnale del bottone sarà quindi ripulito dalle oscillazioni e durerà solo un colpo di clock. Per gestire correttamente la sincronizzazione tra il bottone ed il segnale di divisione (in modo che il riconoscitore veda il bottone premuto) è stato aggiunto un ingresso DIV, che è atteso alto prima di alzare il segnale del bottone nello stato "controllo_pressione".

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity automa_debouncer is
6   generic (
7     CLK_period: integer := 10; -- periodo del clock in nanosec
8     btn_noise_time: integer := 10000000 --durata dell'oscillazione in nanosec
9   );
10  Port ( RST : in STD_LOGIC;
11        div : in std_logic;
12        CLK : in STD_LOGIC;

```

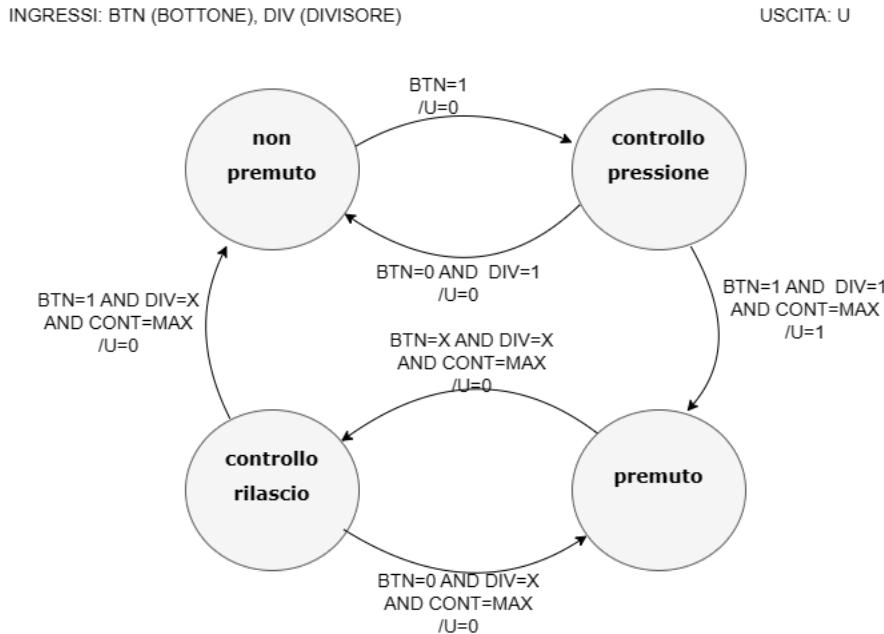


Figura 21: automa_debouncer

```

13      BTN : in STD_LOGIC;
14      CLEARED_BTN : out STD_LOGIC);
15  end automa_debouncer;
16
17 architecture Behavioral of automa_debouncer is
18
19 type stato is (non_premuto,controllo_pressione, premuto, controllo_rilascio );
20 constant max_count: integer :=btn_noise_time/CLK_period;
21 signal btn_state : stato := non_premuto;
22 begin
23 debouncer : process (CLK)
24 variable cont: integer :=0;
25 begin
26 if (falling_edge(CLK)) then
27     if (RST= '1') then
28         btn_state <= non_premuto;
29         CLEARED_BTN<='0';
30
31     else
32         case btn_state is
33             when non_premuto =>
34                 CLEARED_BTN <='0';
35                 cont := 0;
36                 if (BTN = '1') then
37                     btn_state <= controllo_pressione;
38                 end if;
39
40             when controllo_pressione =>
41                 if (cont /= max_count-1) then
42                     cont := cont+1;
43                 else
44                     if (div = '1') then

```

```

45          if (BTN = '1') then
46              btn_state<=premuto;
47              cont:=0;
48              CLEARED_BTN <='1';
49          else
50              btn_state <= non_premuto;
51          end if;
52      end if;
53  end if;

54

55
56      when premuto          =>
57          CLEARED_BTN <='0';
58          if (BTN = '0') then
59              btn_state <= controllo_rilascio;
60          end if;
61
62      when controllo_rilascio  =>
63          if (cont /= max_count-1) then
64              cont := cont+1;
65          else
66              if (BTN = '0') then
67                  btn_state <= non_premuto;
68                  CLEARED_BTN <= '0';
69              else
70                  btn_state <= premuto;
71              end if;
72          end if;
73
74      when others             =>
75          btn_state <= non_premuto;
76
77  end case;
78 end if;
79 end if;
80
81 end process;
82
83 end Behavioral;

```

automa_riconoscitore

Anche il componente automa_riconoscitore ha avuto dei cambiamenti a causa degli ingressi sugli switch, degli enable sui buttoni e della sincronizzazione su segnale di temporizzazione in ingresso (DIV). Questi cambiamenti sono stati gestiti nella parte di controllo, dove sono stati messi controlli sull'ingresso DIV e sui buttoni in ingresso rispetto agli stati (la parte operativa è rimasta invariata).

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Riconoscitore_2modi is
5     port(

```

```

6      i : in std_logic;
7      div : in std_logic;
8      RST: in std_logic;
9      CLK: in std_logic;
10     M : in std_logic;
11     i_read : in std_logic;
12     M_read : in std_logic;
13     uscita : out std_logic;
14     state : out std_logic_vector(2 downto 0)
15   );
16 end Riconoscitore_2modi;
17
18 architecture Behavioral of Riconoscitore_2modi is
19 type stato is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11);
20 signal stato_corrente : stato := S0;
21 signal stato_prossimo : stato;
22 signal Y : std_logic;
23
24 begin
25
26 stato_uscita: process (i, stato_corrente, M, div)
27 begin
28
29   case stato_corrente is
30     when S0 =>
31       if (M = '0') then
32         stato_prossimo <= S5;
33         Y <= '0';
34       else
35         stato_prossimo <= S1;
36         Y <= '0';
37       end if;
38
39     when S1 =>
40       if (i = '1') then
41         stato_prossimo <= S2;
42         Y <= '0';
43       else
44         stato_prossimo <= S1;
45         Y <= '0';
46       end if;
47
48     when S2 =>
49       if (i = '1') then
50         stato_prossimo <= S2;
51         Y <= '0';
52       else
53         stato_prossimo <= S3;
54         Y <= '0';
55       end if;
56
57     when S3 =>
58       if (i = '1') then
59         stato_prossimo <= S2;
60         Y <= '0';
61       else
62         stato_prossimo <= S4;
63         Y <= '0';
64       end if;
65

```

```

66      when S4 =>
67          if (i = '1') then
68              stato_prossimo <= S1;
69              Y <= '1';
70          else
71              stato_prossimo <= S1;
72              Y <= '0';
73          end if;
74
75      when S5 =>
76          if (i = '1') then
77              stato_prossimo <= S6;
78              Y <= '0';
79          else
80              stato_prossimo <= S9;
81              Y <= '0';
82          end if;
83
84      when S6 =>
85          if (i = '0') then
86              stato_prossimo <= S7;
87              Y <= '0';
88          else
89              stato_prossimo <= S10;
90              Y <= '0';
91          end if;
92
93      when S7 =>
94          if (i = '0') then
95              stato_prossimo <= S8;
96              Y <= '0';
97          else
98              stato_prossimo <= S11;
99              Y <= '0';
100         end if;
101
102     when S8 =>
103         if (i = '0') then
104             stato_prossimo <= S5;
105             Y <= '0';
106         else
107             stato_prossimo <= S5;
108             Y <= '1';
109         end if;
110
111     when S9 =>
112         stato_prossimo <= S10;
113         Y <= '0';
114
115     when S10 =>
116         stato_prossimo <= S11;
117         Y <= '0';
118
119     when S11 =>
120         stato_prossimo <= S5;
121         Y <= '0';
122
123     end case;
124
125 end process;

```

```

126 registro: process (CLK)
127 begin
128   if(CLK'event and CLK='1') then
129     if(RST='1') then
130       stato_corrente <= S0;
131       uscita <= '0';
132     else
133       if (div = '1') then
134         if (stato_corrente = S0 and m_read='1') then
135           stato_corrente <= stato_prossimo;
136           uscita <= Y;
137         elsif (stato_corrente /= S0 and i_read ='1') then
138           stato_corrente <= stato_prossimo;
139           uscita <= Y;
140         end if;
141       end if;
142     end if;
143   end if;
144 end process;
145
146 end Behavioral;

```

riconoscitore_su_board

Il componente riconoscitore_su_board gestisce i collegamenti tra i vari componenti della macchina ed aggiunge gli input dall'esterno, di conseguenza è stato sviluppato completamente in modo strutturale.

```

1
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4
5 entity Riconoscitore_su_board is port (
6   clock      : in std_logic;
7   reset      : in std_logic;
8   button1    : in std_logic;
9   button2    : in std_logic;
10  switch1    : in std_logic;
11  switch2    : in std_logic;
12  led        : out std_logic
13 );
14 end Riconoscitore_su_board;
15
16 architecture structural of Riconoscitore_su_board is
17
18 component Riconoscitore_2modi is
19   port(
20     i : in std_logic;
21     div : in std_logic;
22     RST: in std_logic;
23     CLK: in std_logic;
24     M : in std_logic;
25     i_read : in std_logic;

```

```

26      M_read : in std_logic;
27      uscita : out std_logic
28  );
29 end component;
30
31 component automa_debouncer is
32   generic (
33     CLK_period: integer := 10; -- periodo del clock in nanosec
34     btn_noise_time: integer := 30 --durata dell'oscillazione in nanosec
35   );
36   Port ( RST : in STD_LOGIC;
37         div : in std_logic;
38         CLK : in STD_LOGIC;
39         BTN : in STD_LOGIC;
40         CLEARED_BTN : out STD_LOGIC);
41 end component;
42
43 component Divisore_freq is
44   generic (
45     f_in : integer := 100000000;
46     f_out : integer := 50000000
47   );
48   port (
49     reset : in std_logic;
50     clock : in std_logic;
51     div : out std_logic
52   );
53 end component;
54
55 signal sig_div : std_logic;
56 signal sig_in : std_logic;
57 signal sig_m : std_logic;
58
59 begin
60
61 riconoscitore : Riconoscitore_2modi port map (
62   i      => switch1,
63   div    => sig_div,
64   RST    => reset,
65   CLK    => clock,
66   M      => switch2,
67   i_read => sig_in,
68   M_read => sig_m,
69   uscita => led
70 );
71
72 deb_in : automa_debouncer port map (
73   RST      => '0',
74   div      => sig_div,
75   CLK      => clock,
76   BTN      => button1,
77   CLEARED_BTN => sig_in
78 );
79
80 deb_m : automa_debouncer port map (
81   RST      => '0',
82   div      => sig_div,
83   CLK      => clock,
84   BTN      => button2,
85   CLEARED_BTN => sig_m

```

```
86 );
87
88 divisore : Divisore_freq port map (
89     reset    => '0',
90     clock    => clock,
91     div      => sig_div
92 );
93
94 end structural;
```

4 Shift register

In questo esercizio bisogna implementare un registro a scorrimento. Questa è una macchina sequenziale in grado di memorizzare uno o più bit implementando anche un'operazione di shift sui bit memorizzati. Lo shift viene effettuato in presenza di un segnale di abilitazione e può essere implementato in due direzioni (destra o sinistra). Inoltre sia ingresso che uscita del registro possono assumere 4 configurazioni che sono combinazioni di serie e parallelo. Proprio per questa sua caratteristica, il registro a scorrimento, può essere anche utilizzato per la conversione serie/parallelo e parallelo/serie di una stringa di bit.

Traccia

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. Il componente deve essere realizzato utilizzando sia un a) approccio comportamentale sia un b) approccio strutturale. Nota: il numero di bit del registro X e i valori che può assumere il parametro Y possono essere scelti dallo studente (ad es. $X=8$ e $Y=\{1,2\}$).

Per la nostra implementazione abbiamo utilizzato come parametri: $X = 8$ e $Y = \{1, 2\}$

4.1 Progettazione in VHDL con approccio comportamentale

4.1.1 Schematici

Per implementare il registro a scorrimento in maniera comportamentale abbiamo definito un'interfaccia che ci permetta di implementare tutte le operazioni definite dalla traccia. Abbiamo, quindi, utilizzato un ingresso di selezione che permette di selezionare tra shift a destra o sinistra di 1 o 2 bit. Lo shift viene effettivamente eseguito in presenza di un ingresso di enable. Infine, dato che è possibile shiftare di 1 o 2 bit, sia l'ingresso di input dati che l'uscita di output dati sono di 2 bit.

Nella figura 22 vi è lo schematico che riassume le scelte fatte in fase di progettazione.

4.1.2 Codice VHDL

Definiamo prima di tutto l'interfaccia del componente descritto in precedenza. Dato che parliamo di una macchina sequenziale è necessario che la macchina riceva il clock clk in ingresso ed anche un segnale di $reset$, per riportare ad uno stato noto la macchina. Per

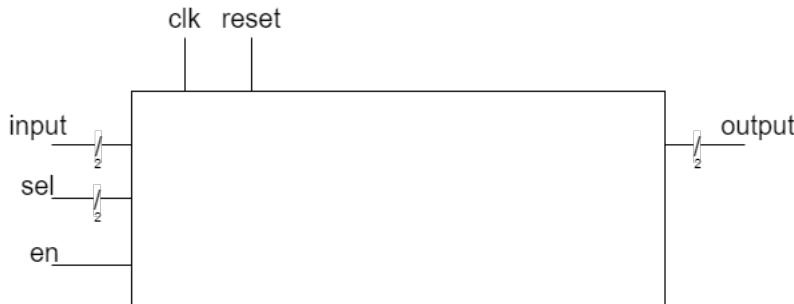


Figura 22: Registro da 8 bit con shift a destra/sinistra di 1/2 bit

selezionare tra i 4 comportamenti del registro a scorrimento bisogna modificare il valore dell’ingresso di selezione *sel*, che modifica la semantica dei segnali di *input* ed *output*:

- *sel = "00"*: shift verso destra di un bit, l’ingresso viene preso da sinistra e viene considerato solo il bit meno significativo dell’ingresso, mentre in uscita vi saranno gli ultimi due bit a destra
- *sel = "01"*: shift verso sinistra di un bit, l’ingresso viene preso da destra e viene considerato solo il bit meno significativo dell’ingresso, mentre in uscita vi saranno gli ultimi due bit a sinistra
- *sel = "10"*: shift verso destra di due bit, l’ingresso viene preso da sinistra e vengono considerati entrambi i bit dell’ingresso, mentre in uscita vi saranno gli ultimi due bit a destra
- *sel = "11"*: shift verso sinistra di due bit, l’ingresso viene preso da destra e vengono considerati entrambi i bit dell’ingresso, mentre in uscita vi saranno gli ultimi due bit a sinistra

Lo shift viene effettuato solamente quando il segnale di enable *en* è alto. Di seguito l’interfaccia appena descritta implementata in VHDL:

```

1 entity Shift_reg is port(
2     clk      : in std_logic;
3     en       : in std_logic;
4     reset    : in std_logic;
5     input    : in std_logic_vector(1 downto 0);
6     sel      : in std_logic_vector(1 downto 0);
7     output   : out std_logic_vector(1 downto 0)
8 );
9 end Shift_reg;
```

Definiti i segnali che compongono l’interfaccia passiamo ora all’implementazione del comportamento del registro a scorrimento. Per farlo è stato utilizzato, come richiesto dalla traccia, un approccio comportamentale. Abbiamo quindi utilizzato il costrutto *process* per svolgere l’operazione di shift sul fronte di salita del clock, solo quando il segnale di enable è alto. Inoltre è stato anche implementato un reset sincrono con il clock. Le uscite del registro sono definite fuori dal process, in quanto così avremo in output sempre i due bit, a destra o sinistra, attualmente presenti nel registro. Inserendo la definizione delle uscite nel process avrei che, i bit in output saranno quelli "usciti" dal registro con l’ultima operazione di shift, con questo approccio il compilatore inferirebbe ulteriori due registri per mantenere i due bit in uscita, quindi il registro da 8 bit diverrebbe un effettivo registro da 10 bit, non rispettando le indicazioni della traccia. Quindi è essenziale che la definizione del segnale di output sia fatta fuori dal process. In particolare è stato usato il costrutto *when – else* per selezionare in output i bit a destra o sinistra in base al valore della selezione. Per selezionare il comportamento in base al segnale *sel* è stato utilizzato il costrutto *case – when* e, nel caso il cui il segnale di sel sia fuori dai casi definiti, si effettua uno shift di un bit a destra. Di seguito il codice VHDL che implementa il comportamento appena descritto:

```

1 architecture Behavioral of Shift_reg is
2
3 signal reg: std_logic_vector(7 downto 0);
4
5 begin
6
7 process (clk)
8 begin
9 if (rising_edge(clk)) then
10
11 if (reset = '1') then
12     reg <= (others => '0');
13 end if;
14
15 if (en = '1') then
16     case sel is
17         when "00" =>
18             reg(7) <= input(0);
19             reg(6 downto 0) <= reg(7 downto 1);
20
21         when "01" =>
22             reg(0) <= input(0);
23             reg(7 downto 1) <= reg(6 downto 0);
24
25         when "10" =>
26             reg(7) <= input(1);
27             reg(6) <= input(0);
28             reg(5 downto 0) <= reg(7 downto 2);
29
30         when "11" =>

```

```

31         reg(0) <= input(1);
32         reg(1) <= input(0);
33         reg(7 downto 2) <= reg(5 downto 0);
34
35     when others =
36         reg(7) <= input(0);
37         reg(6 downto 0) <= reg(7 downto 1);
38     end case;
39
40   end if;
41 end if;
42 end process;
43
44 output(0) <= reg(0) when sel = "00" else
45         reg(7) when sel = "01" else
46         reg(0) when sel = "10" else
47         reg(7) when sel = "11" else
48         reg(0);
49
50 output(1) <= reg(1) when sel = "00" else
51         reg(6) when sel = "01" else
52         reg(1) when sel = "10" else
53         reg(6) when sel = "11" else
54         reg(1);
55
56 end Behavioral;

```

4.1.3 Simulazione

La simulazione del registro a scorrimento è stata fatta testando tutti e 4 le operazioni definite. Prima di tutto però abbiamo riempito il registro con la sequenza di bit "00101110" per poi effettuare, in quest'ordine:

- shift a destra di un bit, con in ingresso il valore '1'
- shift a sinistra di un bit, con in ingresso il valore '0'
- shift a destra di due bit, con in ingresso il valore "11"
- shift a sinistra di due bit, con in ingresso il valore "00"

In figura 23 è presentato il risultato della simulazione. Come si può notare gli shift vengono effettuati sul primo fronte di salita del clock dopo che l'enable si alza. Il campo reg indica i bit memorizzati nel registro ad ogni istante di tempo.

Fino a 180 ns il registro viene riempito con la stringa "00101110", successivamente vengono effettuate le operazioni poco sopra ogni volta che l'enable si alza. Si può notare anche come in uscita vi siano sempre 2 bit, che sono quelli da un lato o l'altro del registro in base alla direzione dell'ultimo shift effettuato.

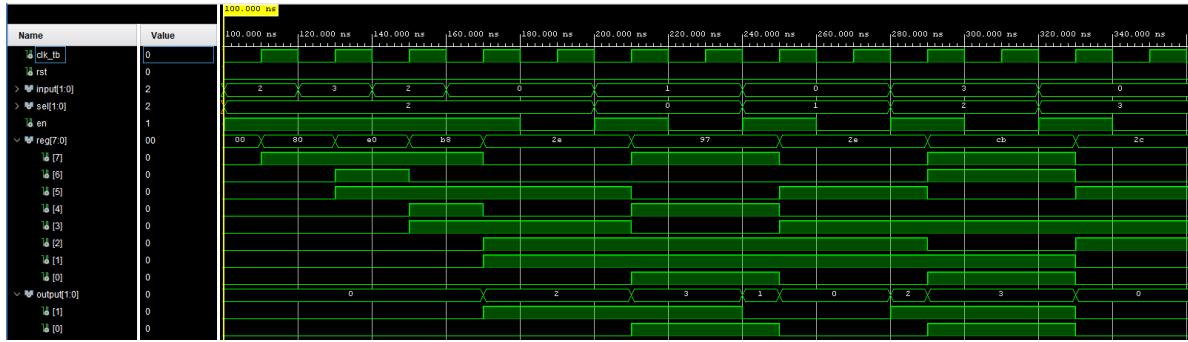


Figura 23: Simulazione del registro realizzato con approccio comportamentale

4.2 Progettazione in VHDL con approccio strutturale

4.2.1 Schematici

Per implementare il registro a scorrimento in maniera strutturale abbiamo utilizzato la stessa interfaccia definita con l'approccio comportamentale. La vera difficoltà nell'approccio strutturale è stata, partendo dal singolo registro di un bit, arrivare a implementare il comportamento desiderato. Per l'intero progetto sono stati necessario solamente due componenti: un Mux [4:1] e un registro ad un singolo bit. Il Mux [4:1] è stato descritto nel paragrafo 1.1.1, di seguito descriviamo il registro che memorizza un solo bit.

Registro con un bit

Il comportamento del registro ad un bit è molto semplice, in presenza di un segnale di abilitazione il valore in input viene memorizzato nel registro, mentre in output viene sempre presentato il valore memorizzato nel registro. Per implementare questo comportamento è necessario che il componente abbia un segnale di input ed uno di output, entrambi da un bit, un segnale di abilitazione, in presenza del quale il valore in input viene memorizzato nel registro ed un segnale di preset, che non è strettamente necessario per questo esercizio (quindi non sarà presente nello schematico), ma che permette di caricare il registro con il valore '1'.

In figura 24 vi è lo schematico del componente appena descritto.

Registro con approccio strutturale

Progettato il registro da un bit ed avendo a disposizione il multiplexer 4:1 diventa semplice definire il registro a scorrimento richiesto dalla traccia. Abbiamo deciso di utilizzare dei mux 4:1 in ingresso a ciascun registro da 1 bit, la selezione dei mux è quella in ingresso al

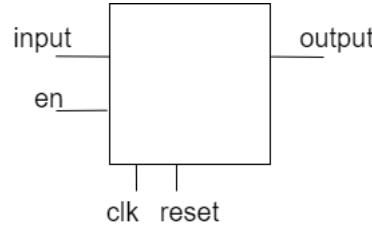


Figura 24: Registro ad un bit

registro a scorrimento, mentre gli ingressi del multiplexer sono gli output di registri specifici che si trovano prima o dopo quelli esaminati. Con opportuni collegamenti si realizza il comportamento richiesto. Inoltre, dato che gli output cambiano in base all'ultima operazione, sono necessari altri due mux 4:1 per selezionare i bit in output.

In figura 25 vi è lo schematico del registro a scorrimento realizzato strutturalmente. Seguendo lo schematico percorriamo i collegamenti del registro 6. La prima linea del mux corrisponde ad uno shift di un bit verso destra, quindi è collegata all'output del registro 7; La seconda linea del mux corrisponde ad uno shift di un bit verso sinistra, quindi è collegata all'output del registro 5; La terza linea del mux corrisponde ad uno shift di due bit verso destra, quindi è collegata al bit più significativo dell'input; La quarta linea del mux corrisponde ad uno shift di due bit verso sinistra, quindi è collegata alle linee 2 e 4 del mux che seleziona il bit più significativo dell'output (che vengono selezionate con shift verso sinistra).

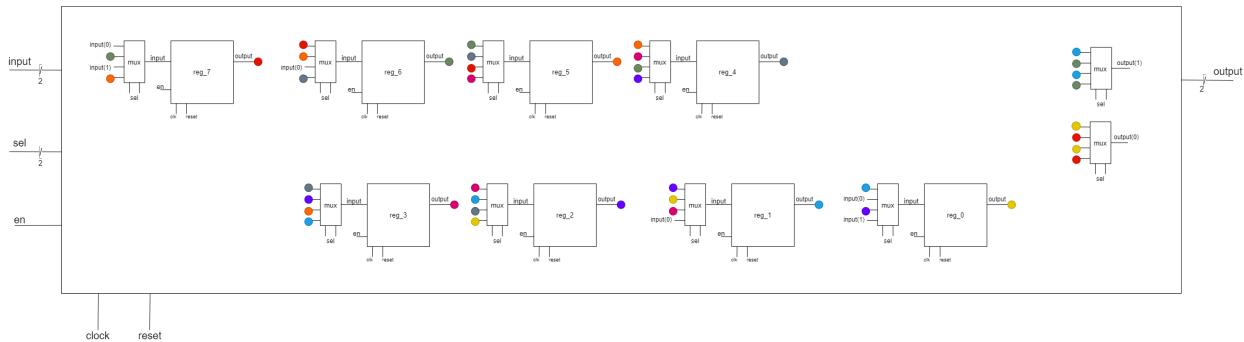


Figura 25: Registro strutturale da 8 bit con shift a destra/sinistra di 1/2 bit

4.2.2 Codice VHDL

L'interfaccia del registro a scorrimento con approccio comportamentale è la stessa di quello con approccio strutturale.

```

1 entity Shift_reg is port(
2     clk      : in std_logic;
3     en       : in std_logic;
4     reset    : in std_logic;
5     input    : in std_logic_vector(1 downto 0);
6     sel      : in std_logic_vector(1 downto 0);
7     output   : out std_logic_vector(1 downto 0)
8 );
9 end Shift_reg;

```

Quello che varia è l'implementazione del comportamento, che in questo caso è fatta tramite una composizione di componenti. Il codice che realizza il mux 4:1 è quello visto nel paragrafo 1.1.2, vediamo quindi il codice del registro da un bit.

Registro con un bit

Per implementare in VHDL il registro da un bit abbiamo utilizzato un approccio comportamentale sfruttando il costrutto *process* per memorizzare nel registro il valore in input sul fronte di salita del clock, solo quando anche l'abilitazione è alta. Inoltre è stato implementato anche un reset sincrono e un preset che viene fatto solo in presenza dell'abilitazione. Di seguito il codice VHDL:

```

1 entity Registrer is port (
2     clk      : in std_logic;
3     en       : in std_logic;
4     reset    : in std_logic;
5     preset   : in std_logic;
6     input    : in std_logic;
7     output   : out std_logic
8 );
9 end Registrer;
10
11 architecture Behavioral of Registrer is
12 begin
13
14 process (clk)
15 begin
16 if (rising_edge(clk)) then
17     if (reset = '1') then
18         output <= '0';
19     end if;
20     if (en = '1') then
21         if (preset = '1') then
22             output <= '1';
23         else
24             output <= input;
25         end if;
26     end if;
27 end if;
28 end process;
29
30 end Behavioral;

```

Registro con approccio strutturale

Per implementare in VHDL il registro da un bit abbiamo utilizzato un approccio strutturale, cioè componendo opportunamente i mux 4:1 e il registri seguendo lo schematico in figura 25. In particolare è stato necessario definire dei segnali per le interconnessioni tra le varie componenti:

- *d_tmp*: mantiene i valori di output dei registri che saranno
- *mux_out*: mantiene i valori di output dei mux, che andranno in input ai registri
- *mux_in*: mantiene i valori che saranno poi in ingresso ai mux

Come vedremo tra poco nel codice sono stati utilizzati dei *for – generate* per collegare facilmente e velocemente le uscite dei registri in *d_tmp* con gli ingressi dei mux *mux_in*. Questo è stato necessari anche perchè i due segnali sono definiti uno tramite *downto* e l'altro tramite un *to*.

Di seguito il codice che implementa il registro a scorrimento strutturale:

```

1 architecture Structural of Shift_reg_struct is
2 signal d_tmp      : std_logic_vector (7 downto 0);
3 signal mux_out    : std_logic_vector (7 downto 0);
4 signal mux_in     : std_logic_vector (0 to 31);
5
6 begin
7 mux_7 : mux_4_1 port map (
8     input(0 to 3)    => mux_in(0 to 3),
9     s                  => sel,
10    output            => mux_out(7)
11 );
12
13 reg_7 : Registrer port map (
14     clk      => clk,
15     en       => en,
16     reset    => reset,
17     preset   => '0',
18     input    => mux_out(7),
19     output   => d_tmp(7)
20 );
21
22 mux_6 : mux_4_1 port map (
23     input(0 to 3)    => mux_in(4 to 7),
24     s                  => sel,
25     output            => mux_out(6)
26 );
27
28 mux_intermedi : for j in 2 to 5 generate
29     mux : mux_4_1 port map (
30         input(0 to 3)    => mux_in(4*j to (4*j)+3),
31         s                  => sel,
32         output            => mux_out(7-j)

```

```

33 );
34 end generate mux_intermedi;
35
36 registri_intermedi : for i in 6 downto 1 generate
37     reg : Registrer port map (
38         clk      => clk,
39         en       => en,
40         reset    => reset,
41         preset   => '0',
42         input    => mux_out(i),
43         output   => d_tmp(i)
44 );
45 end generate registri_intermedi;
46
47 mux_1 : mux_4_1 port map (
48     input(0 to 3)    => mux_in(24 to 27),
49     s                  => sel,
50     output            => mux_out(1)
51 );
52
53 mux_0 : mux_4_1 port map (
54     input(0 to 3)    => mux_in(28 to 31),
55     s                  => sel,
56     output            => mux_out(0)
57 );
58
59 reg_0 : Registrer port map (
60     clk      => clk,
61     en       => en,
62     reset    => reset,
63     preset   => '0',
64     input    => mux_out(0),
65     output   => d_tmp(0)
66 );
67
68 m_output0 : mux_4_1 port map (
69     input(0)    => d_tmp(0),
70     input(1)    => d_tmp(7),
71     input(2)    => d_tmp(0),
72     input(3)    => d_tmp(7),
73     s          => sel,
74     output     => output(0)
75 );
76
77 m_output1 : mux_4_1 port map (
78     input(0)    => d_tmp(1),
79     input(1)    => d_tmp(6),
80     input(2)    => d_tmp(1),
81     input(3)    => d_tmp(6),
82     s          => sel,
83     output     => output(1)
84 );
85
86 mux_line_1: for k in 1 to 7 generate
87     mux_in(4*k) <= d_tmp(8-k);
88 end generate mux_line_1;
89
90 mux_line_2: for k in 6 downto 0 generate
91     mux_in((4*k)+1) <= d_tmp(6-k);
92 end generate mux_line_2;

```

```

93
94 mux_line_3: for k in 1 to 6 generate
95   mux_in((4*k)+6) <= d_tmp(8-k);
96 end generate mux_line_3;
97
98 mux_line_4: for k in 6 downto 1 generate
99   mux_in((4*k)-1) <= d_tmp(6-k);
100 end generate mux_line_4;
101
102 mux_in(0) <= input(0);
103 mux_in(29) <= input(0);
104 mux_in(2) <= input(1);
105 mux_in(6) <= input(0);
106 mux_in(27) <= input(0);
107 mux_in(31) <= input(1);
108
109 end Structural;

```

4.2.3 Simulazione

La simulazione del registro a scorrimento è stato fatta testando tutti e 4 le operazioni definite. Dato che l'interfaccia e il funzionamento sono gli stessi che abbiamo visto per il registro a scorrimento comportamentale abbiamo utilizzato lo stesso codice per la simulazione del componente.

In figura 26 è presentato il risultato della simulazione. Come si può notare gli shift vengono effettuati sul primo fronte di salita del clock dopo che l'enable si alza. Il campo *d_tmp* indica i bit memorizzati nel registro ad ogni istante di tempo.

Come si può notare dalla figura la simulazione di questo componente è perfettamente sovrapponibile con quella che è stata fatta per il registro a scorrimento comportamentale. Quindi valgono le stesse considerazioni fatte in precedenza.



Figura 26: Simulazione del registro realizzato con approccio strutturale

5 Cronometro

In questo esercizio é stato progettato un cronometro in vhdl. Per lo sviluppo dei punti della traccia successivi al primo il progetto é stato sviluppato partendo dai componenti definiti durante la trattazzione dei punti precedenti (infatti il cronometro creato prima parte dell'esercizio verrá mantenuto fino alla fine). Infine é importante tener conto che per una preferenza personale abbiamo preferito testare le nostre capacità di ragionamento cercando di realizzare anche li parti di controllo utilizzando componenti notevoli, quindi senza usare il costrutto behavioural.

5.1 Progettazione in VHDL

Traccia

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.

5.1.1 Schematici

Per la creazione del cronometro in vhdl é stato necessario definire dei contatori modulo 60 e modulo 24 (per poter contare i secondi, i minuti e le ore) ed é stato necessario avere un divisore di frequenza che ricava un segnale di conteggio ad 1hz dal clock da 100mhz presente su scheda. I componenti creati sono dunque:

- FF_D, utilizzato come contatore modulo 2, è la base del nostro contatore;
- counter_mod32_struct, utilizzato per ottenere il contatore delle ore;
- counter_mod24_struct, utilizzato per rendere il contatore modulo 32 un contatore modulo 24 utilizzando l'ingresso di reset;
- counter_mod64, utilizzato per ottenere il contatore dei i secondi e minuti;

- counter_mod60, utilizzato per rendere il contatore modulo 64 un contatore modulo 60 utilizzando l'ingresso di reset;
- divisore_freq, utilizzato per dare il segnale di enable ai secondi;
- cronometro, utilizzato per poter collegare i vari contatori ed il divisore (ed avere il comportamento richiesto dalla traccia).

Abbiamo deciso di realizzare il contatore delle ore in modo strutturale, mentre i contatori di ore e minuti saranno realizzati in modo comportamentale.

FF_D

Il componente FF_D è stato implementato come base del contatore delle ore. Il componente è sincrono con il rising_edge del clock e svolge la funzione di salvare l'ingresso (input) nel momento in cui è dato un segnale di abilitazione (en). Il valore salvato dal flip-flop viene messo sull'uscita (output), mentre sull'uscita (not_out) è presente il valore salvato negato.

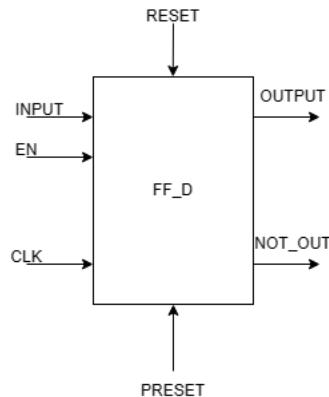


Figura 27: componente FF_D

counter_mod32_struct

Il contatore modulo 32 è stato realizzato in modo strutturale ed ha i seguenti ingressi:

- clk, indispensabile in quanto una macchina sincrona;
- en, utilizzato come segnale d'incremento;
- reset, utilizzato per riazzzerare il contatore;

- set, per impostare il contatore ad un valore scelto dall'utente;
- v_set, utilizzato per indicare quale valore deve essere settato in caso di segnale di set.

E le seguenti uscite:

- counter, un vettore a 5 bit utilizzato per indicare il conteggio del contatore;
- count, un bit che si alza nel momento in cui il contatore é arrivato al massimo numero contabile (32 in questo caso).

Il contatore modulo 32 é stato sviluppato a partire dai flip-flop tipo D. Per permettere il funzionamento corretto del singolo contatore modulo 2 ricavato dal FFD si é dovuto collegare al flip-flop col bit meno significativo la sua uscita negata. I flip-flop successivi hanno in input la xor tra la loro stessa uscita e le uscite dei flip-flop precedenti messe in "AND". Inoltre per avere un corretto funzionamento del preset, la macchina resetta un flip_flop se riceve un set in ingresso ed il bit riguardante quel flip-flop vale '0'. Di seguito é presente la figura rappresentante il contatore modulo 32.

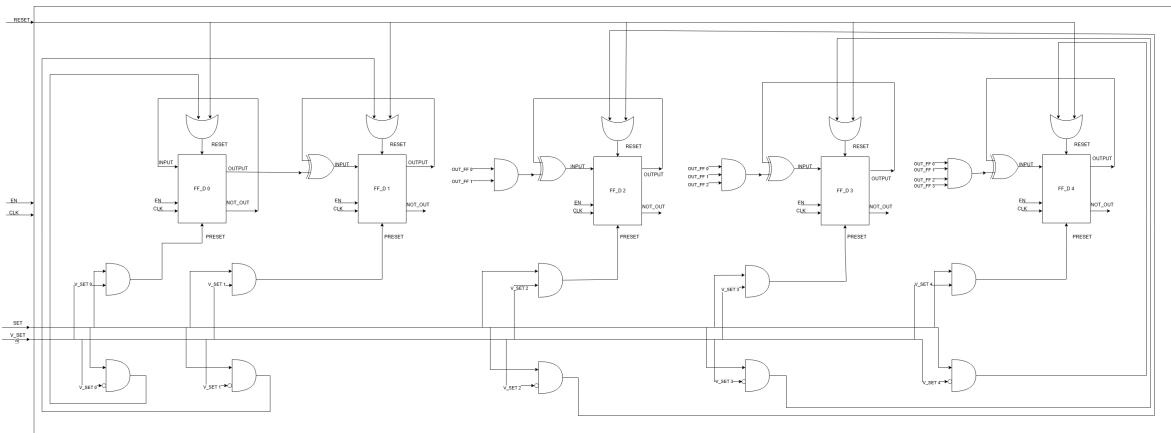


Figura 28: contatore modulo 32 strutturale

counter_mod24_struct

Il componente é utilizzato come contatore delle ore, controlla il valore di conteggio in uscita dal contatore modulo 32 e, una volta arrivato a 24, lo resetta. Di seguito é presenta la figura del componente: 29.

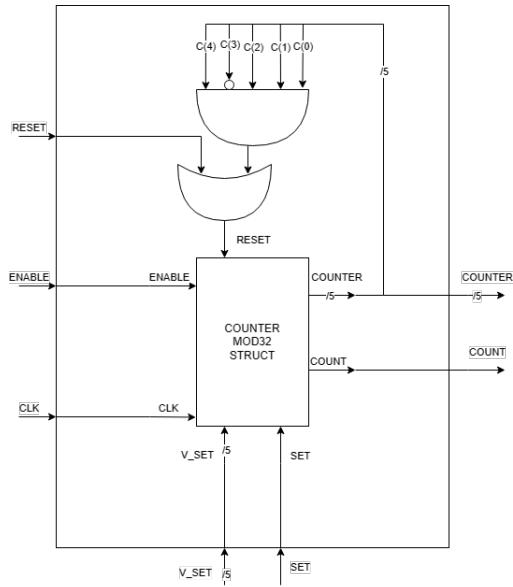


Figura 29: contatore modulo 24 strutturale

counter_mod64

Il componente é utilizzato come base per creare il contatore dei minuti e dei secondi. Questo componente si poteva sviluppare sia in modo comportamentale che in modo strutturale, ma avendo già sviluppato il contatore modulo 24 in strutturale é stata preferita un'implementazione comportamentale. Il componente ha lo stesso comportamento del contatore modulo 32, con la differenza che il conteggio arriva fino a 64. Di seguito é presente il contatore in figura:30

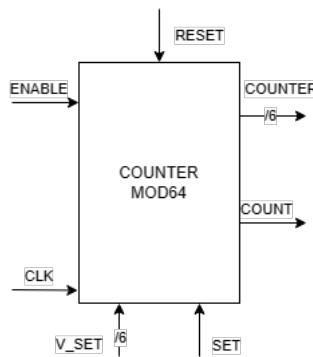


Figura 30: contatore modulo 64

counter_mod60

Il componente é utilizzato come contatore dei minuti e dei secondi, controlla il valore di conteggio in uscita dal contatore modulo 64 per poter poi resettarlo una volta arrivato a 60. Di seguito é presente il contatore in figura: 31.

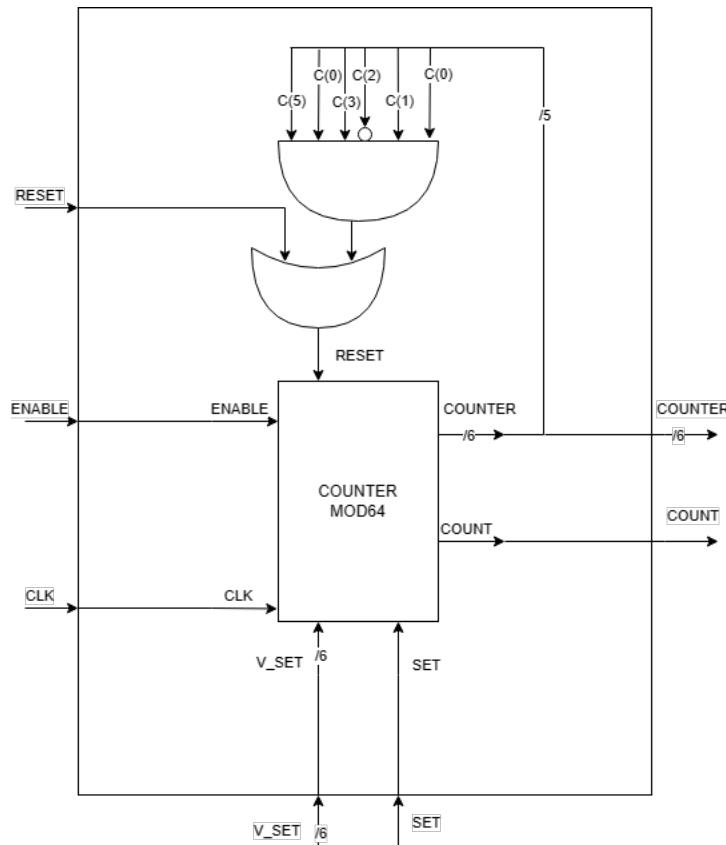


Figura 31: contatore modulo 60

divisore_freq

Il divisore di frequenza utilizzato funziona come quello dell'esercizio 3, nella figura: 18. Viene utilizzato per dare il segnale di enable al contatore dei secondi, quindi é utilizzato per trasformare il clock da 100mhz ad 1hz.

5.1.2 Codice VHDL

FF_D

Il componente FF_D (come specificato in precedenza) funziona come un flip-flop tipo D. È stato descritto seguendo un'architettura Behavioral ed utilizzando il costrutto process. Durante il normale funzionamento il componente quando riceve il segnale di enable (registrato sul rising_edge del clock), salva il valore in input su un segnale (u). Infine all'uscita output è collegato il segnale u, mentre all'uscita not_out è collegato il segnale not (u).

```

1
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4
5 entity FF_D is port (
6     clk      : in std_logic;
7     en       : in std_logic;
8     reset    : in std_logic;
9     preset   : in std_logic;
10    input    : in std_logic;
11    output   : out std_logic;
12    not_out  : out std_logic
13);
14 end FF_D;
15
16 architecture Behavioral of FF_D is
17
18 signal u : std_logic;
19
20 begin
21
22 process (clk)
23
24 begin
25
26 if (rising_edge(clk)) then
27     if (en = '1') then
28         if (preset = '1') then
29             u <= '1';
30
31         elsif (reset = '1') then
32             u <= '0';
33
34         else
35             u <= input;
36         end if;
37     end if;
38 end if;
39
40 end process;
41
42 output <= u;
43 not_out <= not u;
44
45 end Behavioral;
```

counter_mod32_struct

Il contatore modulo 32 é un componente scritto in modo strutturale a partire dal FF_D. I collegamenti sono stati effettuati come indicati nell'immagine: 28. Di seguito é presente il codice vhdl:

```

1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity Counter_mod32_struct is port (
5     clk : in std_logic;
6     en : in std_logic;
7     reset : in std_logic;
8     set : in std_logic;
9     v_set : in std_logic_vector(4 downto 0);
10    counter : out std_logic_vector(4 downto 0);
11    count : out std_logic
12  );
13 end Counter_mod32_struct;
14
15 architecture Structural of Counter_mod32_struct is
16
17 component FF_D is port (
18   clk : in std_logic;
19   en : in std_logic;
20   reset : in std_logic;
21   preset : in std_logic;
22   input : in std_logic;
23   output : out std_logic;
24   not_out : out std_logic
25 );
26 end component;
27
28 signal conteggio : std_logic_vector(4 downto 0) := (others => '0'); -- segnale utilizzato
29   per collegare le uscite di un flip flop all'and di quello successivo
30 signal back : std_logic;      -- segnale utilizzato per permettere il conteggio
31
32 begin
33
34   counter_0 : FF_D
35     port map (
36       clk => clk,
37       en => en,
38       reset => (set and not v_set(0)) or reset,
39       preset => set and v_set(0),
40       input => back,
41       output => conteggio(0),
42       not_out => back
43 );
44
45   counter_1 : FF_D
46     port map (
47       clk => clk,

```

```

47     en => en,
48     reset => (set and not v_set(1)) or reset,
49     preset => set and v_set(1),
50     input => conteggio(0) xor conteggio(1),
51     output => conteggio(1)
52 );
53
54 counter_2 : FF_D
55   port map (
56     clk => clk,
57     en => en,
58     reset => (set and not v_set(2)) or reset,
59     preset => set and v_set(2),
60     input => (conteggio(1) and conteggio(0)) xor conteggio(2),
61     output => conteggio(2)
62 );
63
64 counter_3 : FF_D
65   port map (
66     clk => clk,
67     en => en,
68     reset => (set and not v_set(3)) or reset,
69     preset => set and v_set(3),
70     input => (conteggio(2) and conteggio(1) and conteggio(0)) xor conteggio(3),
71     output => conteggio(3)
72 );
73
74 counter_4 : FF_D
75   port map (
76     clk => clk,
77     en => en,
78     reset => (set and not v_set(4)) or reset,
79     preset => set and v_set(4),
80     input => (conteggio(3) and conteggio(2) and conteggio(1) and conteggio(0)) xor
81     conteggio(4),
82     output => conteggio(4)
83 );
84
85 counter <= conteggio;
86 count <= conteggio(4);
87 end Structural;

```

counter_mod24_struct

Il contatore modulo 24 é un componente sviluppato in modo strutturale per poter resettare il componente counter_mod32_struct una volta che ha contato fino a 24 (ed avere quindi il reset del conteggio). Per svolgere questa funzione é stato necessario creare un segnale interno (conteggio) che si lega all'uscita counter del contatore modulo 32 ed uno (rs) che riconosce il numero 24 attraverso la funzione:

`rs <= conteggio(4) and not conteggio(3) and conteggio(2) and conteggio(1) and conteggio(0);`
 Infine é stato necessario collegare conteggio ed rs con l'esterno, inoltre rs é stato anche

collegato con il reset del contatore modulo 32 per poter avere il componente correttamente funzionante. Di seguito é presente il codice vhdl del componente:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Counter_mod24_struct is port (
5     clk : in std_logic;
6     en : in std_logic;
7     reset : in std_logic;
8     set : in std_logic;
9     v_set : in std_logic_vector(4 downto 0);
10    counter : out std_logic_vector(4 downto 0);
11    count : out std_logic
12 );
13 end Counter_mod24_struct;
14
15 architecture Structural of Counter_mod24_struct is
16
17 component Counter_mod32_struct is port (
18     clk : in std_logic;
19     en : in std_logic;
20     reset : in std_logic;
21     set : in std_logic;
22     v_set : in std_logic_vector(4 downto 0);
23     counter : out std_logic_vector(4 downto 0);
24     count : out std_logic
25 );
26 end component;
27
28 signal rs : std_logic;
29 signal conteggio : std_logic_vector(4 downto 0);
30
31 begin
32
33 rs <= conteggio(4) and not conteggio(3) and conteggio(2) and conteggio(1) and conteggio
34 (0);
35
36 contatore : Counter_mod32_struct
37     port map (
38         clk => clk,
39         en => en,
40         reset => reset or rs,
41         set => set,
42         v_set => v_set,
43         counter => conteggio
44 );
45
46 counter <= conteggio;
47 count <= rs;
48
49 end Structural;

```

counter_mod64

Il componente contatore modulo 64 é stato sviluppato in modo comportamentale tramite il costrutto process. Il componente é molto semplice, normalmente preso un enable in ingresso incrementa il segnale di conteggio (che viene posto in uscita) ed una volta arrivato al valore massimo pone ad '1' l'uscita count, in caso di segnale di set in ingresso ed enable aggiorna la variabile di conteggio con v_set in ingresso ed infine in caso di reset alto resetta il contatore. Di seguito é presente il codice vhdl:

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity Counter_mod64 is port (
6    clk      : in std_logic;
7    en       : in std_logic;
8    reset    : in std_logic;
9    set      : in std_logic;
10   v_set   : in std_logic_vector(5 downto 0);
11   counter : out std_logic_vector(5 downto 0);
12   count   : out std_logic
13 );
14 end Counter_mod64;
15
16 architecture Behavioral of Counter_mod64 is
17
18 signal conteggio : std_logic_vector(5 downto 0) := (others => '0');
19
20 begin
21
22 counter <= conteggio;
23
24 counter_process: process(clk)
25 begin
26
27 if (rising_edge(clk)) then
28
29   if (en = '1') then
30     if (set='1') then
31       conteggio <= v_set;
32       count <= '0';
33     elsif (reset = '1') then
34       count <= '0';
35       conteggio <= (others => '0');
36     elsif (conteggio = "111111") then
37       count <= '1';
38       conteggio <= (OTHERS => '0');
39     else
40       conteggio <= std_logic_vector(unsigned(conteggio) + 1);
41       count <= '0';
42     end if;
43   end if;
44 end if;
45

```

```

46 end process;
47
48 end Behavioral;
```

counter_mod60

Il componente poteva essere evitato, modificando la versione comportamentale del componente counter_mod64 facendolo contare fino a 60, ma visto che la traccia richiedeva l'utilizzo di componenti strutturali abbiamo preferito trasformare il contatore modulo 64 in un contatore modulo 60 con un componente strutturale. Il componente funziona esattamente come il Counter_mod24_struct, con la differenza che la funzione per calcolare il reset e l'uscita count è:

`rs <= conteggio(5) and conteggio(4) and conteggio(3) and not conteggio(2) and conteggio(1)
and conteggio(0);`

Di seguito è presente il codice vhdl del componente:

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Counter_mod60 is port (
5     clk : in std_logic;
6     en : in std_logic;
7     reset : in std_logic;
8     set : in std_logic;
9     v_set : in std_logic_vector(5 downto 0);
10    counter : out std_logic_vector(5 downto 0);
11    count : out std_logic
12 );
13 end Counter_mod60;
14
15 architecture Structural of Counter_mod60 is
16
17 component Counter_mod64 is port (
18     clk : in std_logic;
19     en : in std_logic;
20     reset : in std_logic;
21     set : in std_logic;
22     v_set : in std_logic_vector(5 downto 0);
23     counter : out std_logic_vector(5 downto 0);
24     count : out std_logic
25 );
26 end component;
27
28 signal rs : std_logic;
29 signal conteggio : std_logic_vector(5 downto 0);
30
31 begin
32
33 rs <= conteggio(5) and conteggio(4) and conteggio(3) and not conteggio(2) and conteggio(1)
      and conteggio(0);
```

```

34
35 contatore : Counter_mod64
36     port map (
37         clk => clk,
38         en => en or set or reset,
39         reset => reset or rs,
40         set => set,
41         v_set => v_set,
42         counter => conteggio
43 );
44
45 counter <= conteggio;
46 count <= rs;
47
48 end Structural;

```

divisore_freq

Come detto in precedenza il divisore di frequenza utilizzato funziona come quello dell'esercizio 3. Viene utilizzato per trasformare il clock da 100mhz ad 1hz e permette quindi di contare i secondi. Il codice del divisore di frequenza: 3.2.2

cronometro

Il componente é utilizzato come top entity del progetto e viene quindi utilizzato per unire tutti i componenti creati ed avere la funzione del cronometro. Il componente é stato ovviamente sviluppato in modo strutturale e collega tutti i vari elementi come nella figura: 32. Di seguito il codice vhdl del componente:

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Cronometro is port (
5      clk          : in  std_logic;
6      reset        : in  std_logic;
7      set          : in  std_logic;
8      sel_minuti   : in  std_logic_vector(5 downto 0);
9      sel_secondi  : in  std_logic_vector(5 downto 0);
10     sel_ore       : in  std_logic_vector(4 downto 0);
11     secondi      : out std_logic_vector(5 downto 0);
12     minuti       : out std_logic_vector(5 downto 0);
13     ore          : out std_logic_vector(4 downto 0)
14 );
15 end Cronometro;
16
17 -----
18 -----
19
20 architecture Structural of Cronometro is
21
22 component Counter_mod60 is port (

```

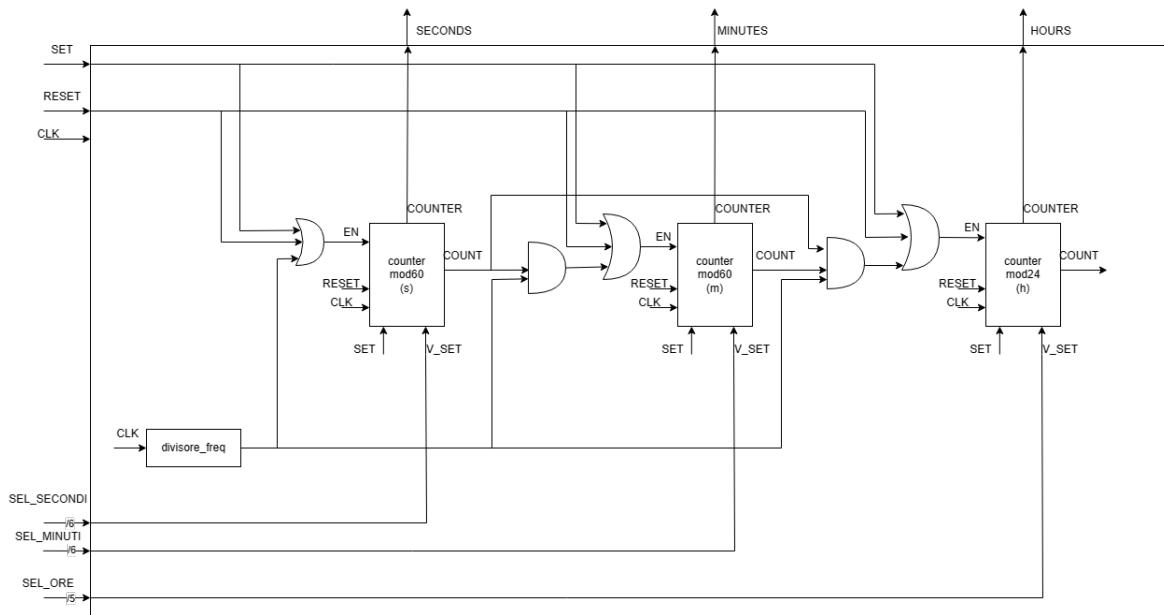


Figura 32: componente cronometro

```

23      clk      : in  std_logic;
24      en       : in  std_logic;
25      reset    : in  std_logic;
26      set      : in  std_logic;
27      v_set   : in  std_logic_vector(5 downto 0);
28      counter : out std_logic_vector(5 downto 0);
29      count   : out std_logic
30  );
31 end component;
32
33 component Counter_mod24_struct is port (
34     clk : in  std_logic;
35     en  : in  std_logic;
36     reset : in  std_logic;
37     set   : in  std_logic;
38     v_set : in  std_logic_vector(4 downto 0);
39     counter : out std_logic_vector(4 downto 0);
40     count  : out std_logic
41  );
42 end component;
43
44 component Divisore_freq is
45     generic (
46         f_in  : integer := 100000000;
47         f_out : integer := 1
48     );
49     port (
50         reset : in  std_logic;
51         clock : in  std_logic;
52         div   : out std_logic
53     );
54 end component;
55

```

```

56 -----
57 -----
58
59 signal cont      : std_logic_vector(1 downto 0);
60 signal en        : std_logic;
61
62 begin
63
64 div : Divisore_freq
65   port map (
66     reset => reset or set,
67     clock => clk,
68     div => en
69 );
70
71 cont_secondi : Counter_mod60
72   port map (
73     clk => clk,
74     en => en or reset or set,
75     reset => reset,
76     set => set,
77     v_set => sel_secondi,
78     counter => secondi,
79     count => cont(0)
80 );
81 cont_minuti : Counter_mod60
82   port map (
83     clk => clk,
84     en => (en and cont(0)) or (reset) or (set),
85     reset => reset,
86     set => set,
87     v_set => sel_minuti,
88     counter => minuti,
89     count => cont(1)
90 );
91
92 cont_ore : Counter_mod24_struct
93   port map (
94     clk => clk,
95     en => (en and cont(1) and cont(0)) or (reset) or (set),
96     reset => reset,
97     set => set,
98     v_set => sel_ore,
99     counter => ore
100);
101
102 end Structural;

```

5.1.3 Simulazione

In fase di simulazione è stato modificato il valore della frequenza in uscita del divisore, infatti adesso il divisore si alza ogni 3 rising_edge del clock. Nel test si può vedere il comportamento della macchina, dopo il reset di 100ns, si hanno 100ns di evoluzione libera e poi il set ad un valore definito dall'utente (01:48:58), infine si è lasciato il componente in evoluzione libera.

È importante notare come nel testbench si veda che il passaggio 59 secondi ad 1 minuto funziona correttamente. Il testbench è presente in figura: 33

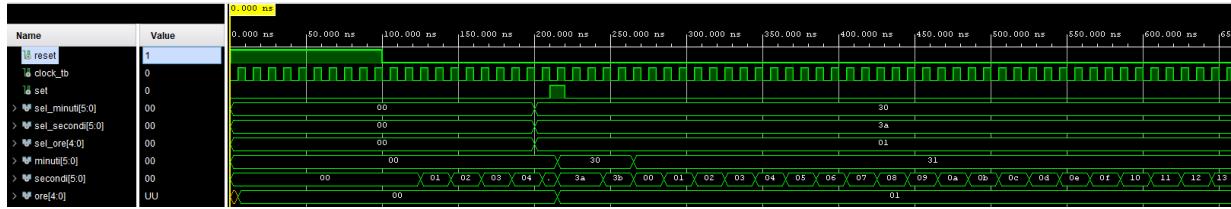


Figura 33: testbench cronometro

5.2 Implementazione su board con display

Traccia

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 buttoni, 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).

5.2.1 Schematici

Per lo svolgimento di questo esercizio è stato utilizzato lo stesso cronometro visto nella scorsa sezione, lo stesso display dell’esercizio 2 (Vedi 2.3.2), ma a cui abbiamo aggiunto un gestore degli anodi, un divisore di frequenza per scegliere quale cifra illuminare, di un contatore per scegliere la cifra da mostrare e di uno splitter che divide le decine dalle unità di un segnale, infine è stato necessario anche un button debouncer (in particolare è stato usato quello dell’esercizio 3, vedi 3.2.1).

Schematici relativi al funzionamento del cronometro

CU_set Il componente CU_set è stato sviluppato a causa di un limite della scheda, infatti gli switch a disposizione della scheda sono 16, mentre gli ingressi per selezionare un orario a scelta dall’utente sono 17. Per non dover rinunciare ad un bit d’ingresso si è preferito prendere solo 12 switch in ingresso, divisi in sel_switch_1 e sel_switch_2 e grazie ad una FSM è possibile inserire un orario tramite i seguenti passaggi:

- inserisci sui 12 switch piú a destra della scheda il valore dell'ingresso per i secondi e per i minuti; (S0)
- premi il tasto set; (S0)
- inserisci sui 5 bit piú a destra il valore delle ore; (S1)
- premi il tasto set; (S1)
- invio del segnale di set al cronometro. (S2)

In figura 35 è presente il componente CU_set ed il suo automa:

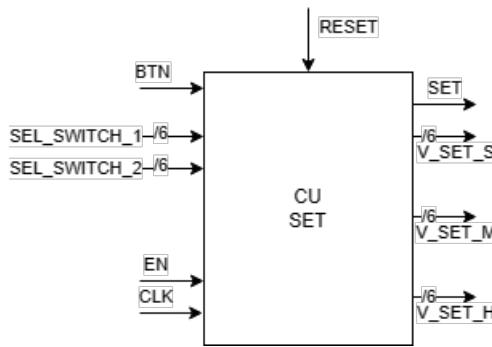


Figura 34: componente CU_set

cronometro Come detto precedentemente il cronometro utilizzato é lo stesso di quello nel punto precedente, ma con l'aggiunta del CU_set, che permette al cronometro di acquisire correttamente il preset dagli switch della scheda. il nuovo componente del cronometro lo si ritrova in figura 36.

Schematici relativi al funzionamento del Display

Per il funzionamento del display é stato necessario creare un'entitá leggermente complessa, infatti per far funzionare il display si ha bisogno di:

- gestire il numero da mostrare; (gestore_catodi)
- gestire quale cifra illuminare; (gestore_anodi, divisore_freq)
- dividere le unitá dalle decine; (splitter)

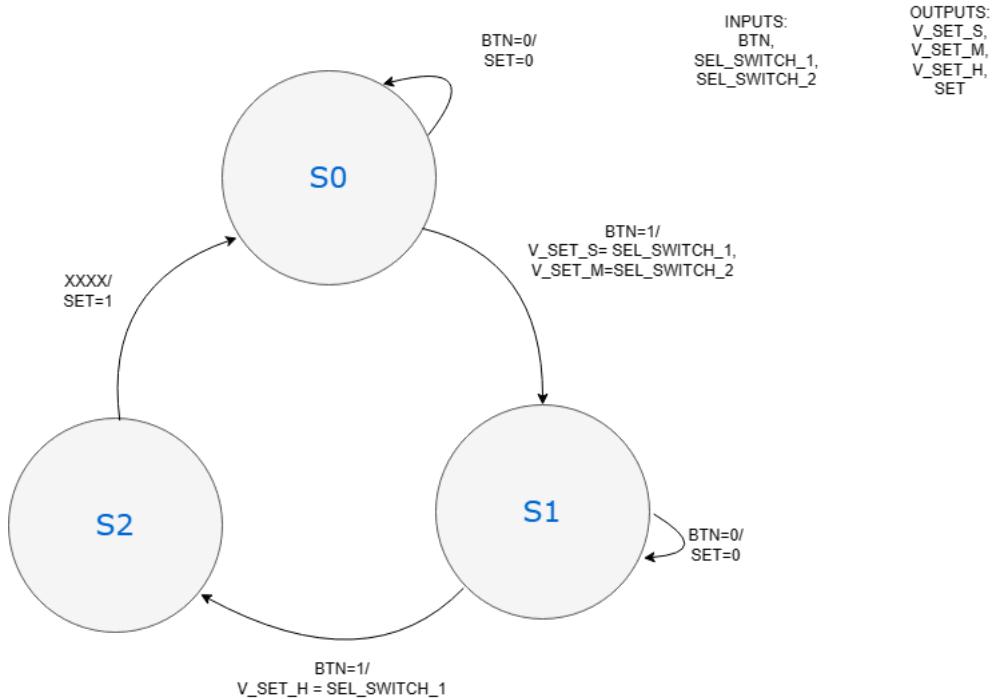


Figura 35: automa CU_set

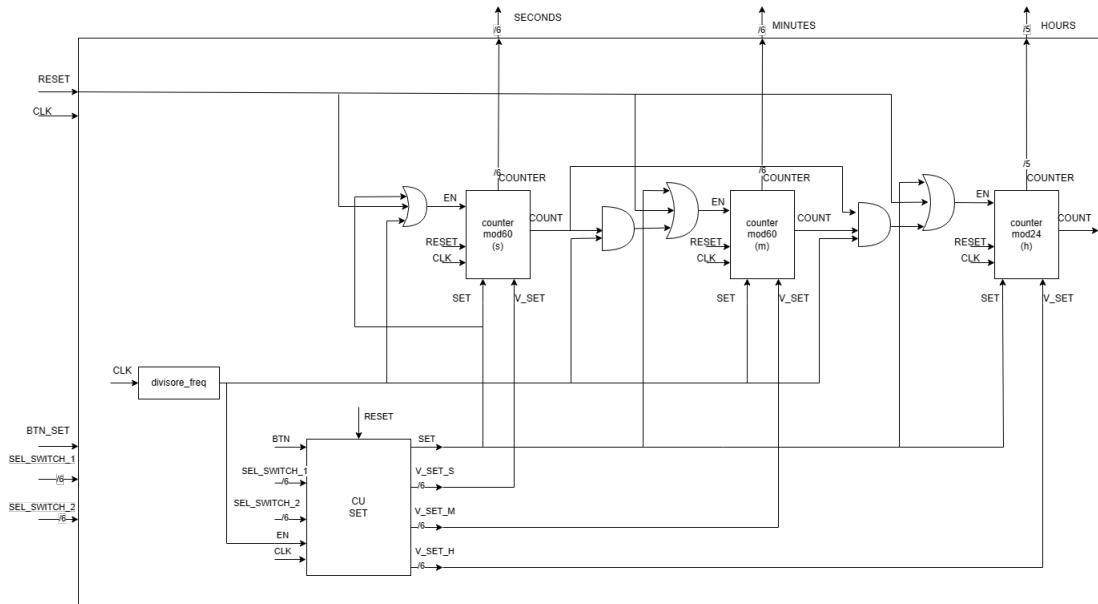


Figura 36: componente cronometro es.2

- gestire il numero da mandare in ingresso. (contatore_modulo_8_display, selettore_display)

Quindi per permettere il normale funzionamento del display, ad ogni enable del divisore di

frequenza si incrementa il contatore modulo 8, che va in ingresso al gestore_anodi ed al selettore display, inoltre il selettore display in base al conteggio del cronometro sceglie il valore da mostrare e lo manda in uscita verso il gestore dei catodi.

NOTA: Avendo già trattato il divisore di frequenza nello stesso esercizio, nella sezione degli schematici verrà omessa la sua descrizione. È possibile vedere in modo strutturale il componente riguardante il display nella figura: 40

splitter Il componente splitter gestisce il vettore in ingresso da 6 bit e attraverso un'operazione di divisione ed una di modulo divide le decine dalle unità e le inserisce in uscita.

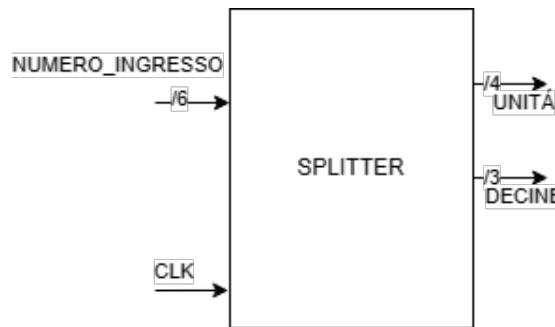


Figura 37: componente splitter

contatore_modulo_8_display Il contatore modulo 8 è stato sviluppato come il contatore modulo 64, con la differenza che il conteggio si ferma ad 8 e quindi si hanno meno fili in uscita (essendo stato già trattato si rimanda alla figura 30). L'uscita del contatore come detto in precedenza viene inviata sia al selettore_display sia al gestore_anodi.

gestore_catodi Il componente è simile a quello sviluppato nell'esercizio 2 (vedi 2.3.2), ma è reso più semplice, infatti si gestiscono solamente i catodi, mentre la gestione degli anodi è fatta da un altro componente.

gestore_anodi Il componente gestore_anodi permette di gestire quale cifra illuminare infatti, in base al valore dell'ingresso sel ha in uscita un vettore anodi con un valore diverso del tipo: "11011111" (dove gli 1 indicano la cifra da spegnere e lo 0 indica la cifra da illuminare). Il componente è mostrato in figura: 38

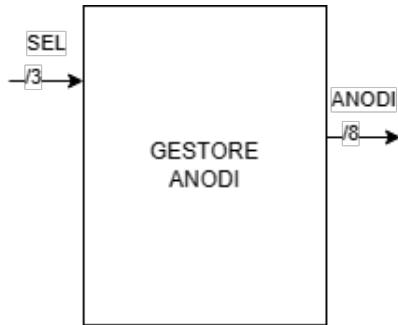


Figura 38: componente gestore anodi

selettore_display Il selettore_display é un componente che prende in ingresso il segnale di conteggio del contatore modulo 8 ed i 6 valori (unità e decine, divise dagli splitter) dei minuti, delle ore e dei secondi, in base al valore di conteggio manda in uscita su un vettore di 4 bit il numero da mostrare su display (segnale che viene inviato al gestore_catodi). Il componente é in figura 39.

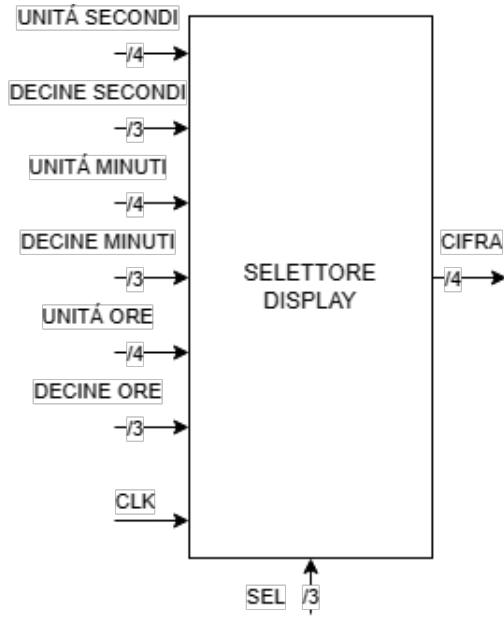


Figura 39: componente selettore display

cronometro_con_display

L'ultimo schema fondamentale consiste nel collegamento tra il display, i bottoni ed il cronometro. Il collegamento tra i vari componenti é molto semplice, infatti gli input di set e reset vanno direttamente nei rispettivi button_debouncer ed il loro segnale "filtrato" viene inviato

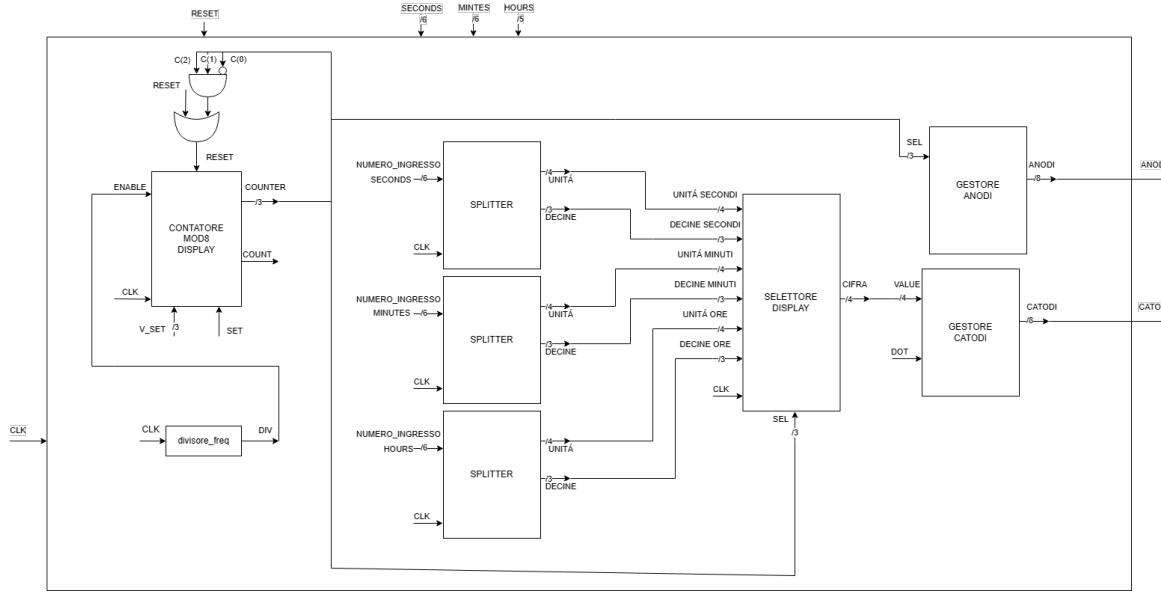


Figura 40: componente display

sia nel componente del cronometro (set e reset) che nel componente del display (reset). Gli inputs degli switch (per i valori del set) sono collegati direttamente al cronometro, infine l'orario in uscita dal cronometro è collegato direttamente col componente display, in modo da avere le combinazioni di catodi ed anodi per il corretto funzionamento. I collegamenti tra tutti i componenti e I/O sono presenti in figura 41.

NOTA: Il componente button_debouncer non è stato descritto in questo paragrafo in quanto è possibile trovarlo nell'esercizio 3 3.2.1

5.2.2 Codice VHDL

Come per gli schematici anche per il codice VHDL dei componenti interni del cronometro e del divisore_freq si rimanda alla sezione 5.1.2 per controllare il codice di tutti gli elementi del cronometro. Inoltre si rimanda alla sezione 3.2.2 per il codice del debouncer.

Codice VHDL relativo al funzionamento del cronometro

CU_set Il CU_set è stato implementato in modo comportamentale grazie al costrutto del process e l'implementazione di una macchina a stati finiti. Gli stati della macchina sono:

- S0, che è lo stato in cui la macchina attende l'immissione dei minuti e dei secondi e la pressione del set;

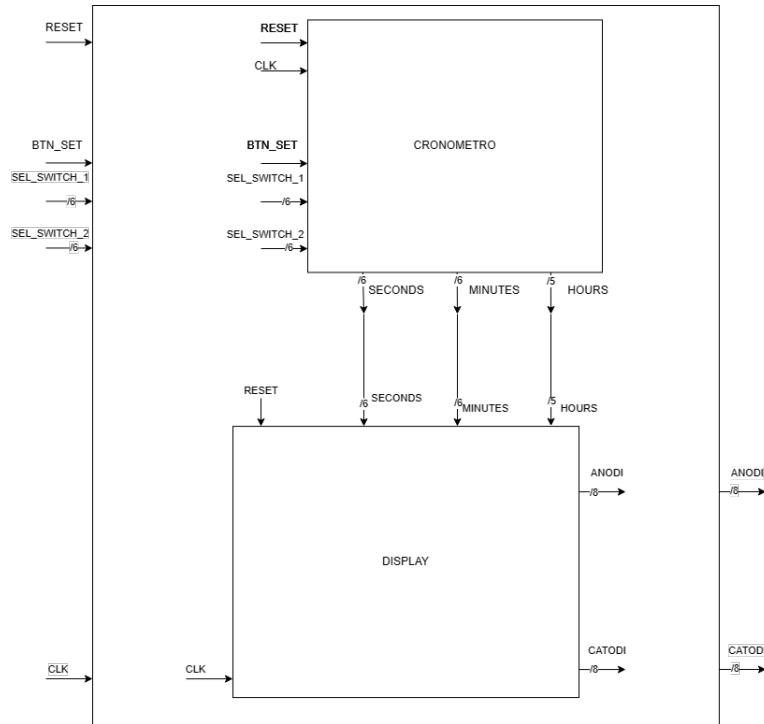


Figura 41: cronometro_con_display

- S1, che è lo stato in cui la macchina attende l'immissione delle ore e la pressione del tasto set;
- S2, che è lo stato che invia il comando di set ai contatori del cronometro.

Di seguito è presente il codice VHDL del componente.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4
5 entity CU_set is port (
6   reset      : in std_logic;
7   en         : in std_logic;
8   btn        : in std_logic;
9   clk        : in std_logic;
10  set_switch_1    : in std_logic_vector(5 downto 0);
11  set_switch_2    : in std_logic_vector(5 downto 0);
12  set_out       : out std_logic;
13  v_set_s       : out std_logic_vector(5 downto 0);
14  v_set_m       : out std_logic_vector(5 downto 0);
15  v_set_h       : out std_logic_vector(4 downto 0)
16 );
17 end CU_set;
18
19 architecture Behavioral of CU_set is
20 
```

```

21 -- definizione macchina di mealy
22 type stato is (S0, S1, S2);
23 signal stato_corrente : stato := S0;
24 signal stato_prossimo : stato;
25 -----
26
27 -----
28 -- segnale per le uscite
29 signal temp_v_set_s : std_logic_vector(5 downto 0) := (others => '0');
30 signal temp_v_set_m : std_logic_vector(5 downto 0) := (others => '0');
31 signal temp_v_set_h : std_logic_vector(4 downto 0) := (others => '0');
32 signal temp_set_out : std_logic := '0';
33 -----
34 begin
35 -----
36 stato_uscita: process (btn, stato_corrente)
37 variable bottone_premuto :std_logic := '0';
38 begin
39
40 case stato_corrente is
41 when S0 =>
42     temp_set_out <= '0';
43     if (btn = '1') then
44         stato_prossimo <= S1;
45         temp_v_set_s <= set_switch_1;
46         temp_v_set_m <= set_switch_2;
47         bottone_premuto := '1';
48     end if;
49     if bottone_premuto = '0' and btn = '0' then
50         stato_prossimo <= S0;
51     end if;
52
53 when S1 =>
54     if (btn = '1') then
55         stato_prossimo <= S2;
56         temp_v_set_h <= set_switch_1 (4 downto 0);
57     end if;
58     bottone_premuto := '0';
59
60 when S2 =>
61     temp_set_out <= '1';
62     stato_prossimo <= S0;
63     bottone_premuto := '0';
64 end case;
65
66
67 end process;
68
69 registro: process (clk)
70 begin
71     if(rising_edge(clk)) then
72         if(reset ='1') then
73             stato_corrente <= S0;
74         elsif (en ='1') then
75             set_out <= temp_set_out;
76             stato_corrente <= stato_prossimo;
77         end if;
78
79     end if;
80

```

```

81 end process;
82
83 v_set_s <= temp_v_set_s;
84 v_set_m <= temp_v_set_m;
85 v_set_h <= temp_v_set_h;
86
87 end Behavioral;

```

cronometro Il componente cronometro continua ad essere strutturale e ad avere la stessa funziona che aveva nel punto precedente, ma in questo caso é stato aggiunto di nuovo il codice in quanto deve essere presente anche il componente CU_set descritto al paragrafo precedente. Di seguito é presente il nuovo codice VHDL del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Cronometro is port (
5     clk          : in  std_logic;
6     reset        : in  std_logic;
7     set          : in  std_logic;
8     set_switch_1 : in  std_logic_vector(5 downto 0);
9     set_switch_2 : in  std_logic_vector(5 downto 0);
10    seconds      : out std_logic_vector(5 downto 0);
11    minutes      : out std_logic_vector(5 downto 0);
12    hours        : out std_logic_vector(4 downto 0)
13 );
14 end Cronometro;
15
16 -----
17 -----
18
19 architecture Structural of Cronometro is
20
21 component Counter_mod60 is port (
22     clk      : in  std_logic;
23     en       : in  std_logic;
24     reset    : in  std_logic;
25     set      : in  std_logic;
26     v_set   : in  std_logic_vector(5 downto 0);
27     counter : out std_logic_vector(5 downto 0);
28     count   : out std_logic
29 );
30 end component;
31
32 component Counter_mod24_struct is port (
33     clk : in  std_logic;
34     en  : in  std_logic;
35     reset : in  std_logic;
36     set   : in  std_logic;
37     v_set : in  std_logic_vector(4 downto 0);
38     counter : out std_logic_vector(4 downto 0);
39     count  : out std_logic
40 );
41 end component;

```

```

42
43 component Divisore_freq is
44   generic (
45     f_in : integer := 100000000;
46     f_out : integer := 1
47   );
48   port (
49     reset : in std_logic;
50     clock : in std_logic;
51     div   : out std_logic
52   );
53 end component;
54
55 component CU_set is port (
56   reset      : in std_logic;
57   en         : in std_logic;
58   btn        : in std_logic;
59   clk         : in std_logic;
60   set_switch_1 : in std_logic_vector(5 downto 0);
61   set_switch_2 : in std_logic_vector(5 downto 0);
62   set_out     : out std_logic;
63   v_set_s    : out std_logic_vector(5 downto 0);
64   v_set_m    : out std_logic_vector(5 downto 0);
65   v_set_h    : out std_logic_vector(4 downto 0)
66 );
67 end component;
68
69 signal en, temp_set      : std_logic;
70 signal conn_m, conn_s    : std_logic_vector(5 downto 0);
71 signal conn_h           : std_logic_vector(4 downto 0);
72 signal cont              : std_logic_vector(1 downto 0);
73
74 begin
75
76 div : Divisore_freq
77   port map (
78     reset => '0',
79     clock => clk,
80     div => en
81 );
82
83 cont_secondi : Counter_mod60
84   port map (
85     clk => clk,
86     en => en or reset or temp_set,
87     reset => reset,
88     set => temp_set,
89     v_set => conn_s,
90     counter => seconds,
91     count => cont(0)
92 );
93 cont_minuti : Counter_mod60
94   port map (
95     clk => clk,
96     en => (en and cont(0)) or (reset) or (temp_set),
97     reset => reset,
98     set => temp_set,
99     v_set => conn_m,
100    counter => minutes,
101    count => cont(1)

```

```

102 );
103
104 cont_ore : Counter_mod24_struct
105   port map (
106     clk => clk,
107     en => (en and cont(1) and cont(0)) or (reset) or (temp_set),
108     reset => reset,
109     set => temp_set,
110     v_set => conn_h,
111     counter => hours
112 );
113
114
115 funzione_set : CU_set  port map (
116   reset      => reset,
117   en         => en,
118   btn        => set,
119   clk         => clk,
120   set_switch_1  => set_switch_1,
121   set_switch_2  => set_switch_2,
122   set_out      => temp_set,
123   v_set_s     => conn_s,
124   v_set_m     => conn_m,
125   v_set_h     => conn_h
126 );
127
128 end Structural;

```

Codice VHDL relativo al funzionamento del Display

splitter Il componente splitter é stato sviluppato in modo comportamentale sfruttando il process. Per poter permettere di eseguire sia l'operazione di modulo (per avere le unitá del segnale in ingresso) che l'operazione di divisione (per avere le decine del segnale in ingresso) é stato necessario utilizzare il pacchetto IEEE.NUMERIC_STD_ALL, dal quale abbiamo usato le funzioni:

- `unsigned()`, per indicare che il vettore in ingresso doveva essere trattato come un numero naturale senza segno;
- `to_integer()`, per trasformare il vettore in ingresso in un intero (per poter poi eseguire le operazioni di modulo e divisione per 10);
- `to_unsigned`, per trasformare il numero intero (risultato sia del modulo che della divisione) in un vettore.

I risultati della divisione e del modulo vengono salvati in 2 segnali implementati come interi che sono: `temp_decine` e `temp_unitá`. Di seguito é presente il codice VHDL del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity splitter is port(
6     numero_ingresso : in std_logic_vector (5 downto 0);
7     clk             : in std_logic;
8     unita          : out std_logic_vector (3 downto 0);
9     decine         : out std_logic_vector (2 downto 0)
10 );
11 end splitter;
12
13 architecture Behavioral of splitter is
14 signal temp_decine, temp_unita : integer;
15 begin
16
17 calcolo : process(clk)
18 begin
19
20 temp_decine <= (to_integer(unsigned(numero_ingresso))) / 10;
21 temp_unita <= (to_integer(unsigned(numero_ingresso))) mod 10;
22
23 end process;
24
25 unita <= std_logic_vector(to_unsigned(temp_unita, 4));
26 decine <= std_logic_vector(to_unsigned(temp_decine, 3));
27
28 end Behavioral;

```

contatore_modulo_8_display Il contatore modulo 8 é stato sviluppato come il contatore modulo 32 in questo esercizio, ma con la differenza che ha un vettore di conteggio (in uscita) ed un segnale di conteggio a 3 bit. Visto che la base del codice VHDL é possibile trovare il codice del contatore al paragrafo 5.1.2

gestore_catodi Nonostante il codice VHDL del gestore_catodi sia lo stesso di quello usato nel capitolo 3, visto che non vengono piú gestiti gli anodi, viene riproposto in questo esercizio.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5
6 entity gestore_catodi is
7     Port ( value : in STD_LOGIC_VECTOR (3 downto 0);
8            dot : in STD_LOGIC;
9            cathodes_dot : out STD_LOGIC_VECTOR (7 downto 0));
10 end gestore_catodi;
11
12 architecture Behavioral of gestore_catodi is
13

```

```

14 constant zero   : std_logic_vector(6 downto 0) := "1000000";
15 constant one    : std_logic_vector(6 downto 0) := "1111001";
16 constant two    : std_logic_vector(6 downto 0) := "0100100";
17 constant three  : std_logic_vector(6 downto 0) := "0110000";
18 constant four   : std_logic_vector(6 downto 0) := "0011001";
19 constant five   : std_logic_vector(6 downto 0) := "0010010";
20 constant six    : std_logic_vector(6 downto 0) := "0000010";
21 constant seven  : std_logic_vector(6 downto 0) := "1111000";
22 constant eight  : std_logic_vector(6 downto 0) := "0000000";
23 constant nine   : std_logic_vector(6 downto 0) := "0010000";
24 constant a      : std_logic_vector(6 downto 0) := "0001000";
25 constant b      : std_logic_vector(6 downto 0) := "0000011";
26 constant c      : std_logic_vector(6 downto 0) := "1000110";
27 constant d      : std_logic_vector(6 downto 0) := "0100001";
28 constant e      : std_logic_vector(6 downto 0) := "0000110";
29 constant f      : std_logic_vector(6 downto 0) := "0001110";
30
31 signal cathodes : std_logic_vector(6 downto 0);
32
33 begin
34
35 seven_segment_decoder_process: process
36 begin
37     case value is
38         when "0000" => cathodes <= zero;
39         when "0001" => cathodes <= one;
40         when "0010" => cathodes <= two;
41         when "0011" => cathodes <= three;
42         when "0100" => cathodes <= four;
43         when "0101" => cathodes <= five;
44         when "0110" => cathodes <= six;
45         when "0111" => cathodes <= seven;
46         when "1000" => cathodes <= eight;
47         when "1001" => cathodes <= nine;
48         when "1010" => cathodes <= a;
49         when "1011" => cathodes <= b;
50         when "1100" => cathodes <= c;
51         when "1101" => cathodes <= d;
52         when "1110" => cathodes <= e;
53         when "1111" => cathodes <= f;
54         when others => cathodes <= (others => '0');
55     end case;
56 end process seven_segment_decoder_process;
57
58 cathodes_dot <= (not dot)&cathodes;
59
60 end Behavioral;

```

gestore_anodi Il componente gestore_anodi è stato sviluppato tramite una descrizione dataflow, utilizzando il costrutto "WITH...SELECT", il componente in base ad un segnale di selezione in ingresso accende un solo numero del display alla volta mettendo a '0' il bit corrispondente al numero. Di seguito è presente il codice VHDL del componente.

```

1 library IEEE;
```

```

2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity gestore_anodi is port (
5     sel      : in    std_logic_vector (2 downto 0);
6     anodi   : out   std_logic_vector (7 downto 0)
7 );
8 end gestore_anodi;
9
10 architecture Dataflow of gestore_anodi is
11
12 begin
13 with sel select
14     anodi  <= "01111111" when "000",
15             "10111111" when "001",
16             "11011111" when "010",
17             "11101111" when "011",
18             "11110111" when "100",
19             "11111011" when "101",
20             "11111101" when "110",
21             "11111110" when "111",
22             "11111111" when others;
23
24 end Dataflow;

```

selettore_display Il componente selettore_display è stato sviluppato in modo comportamentale utilizzando sia il costrutto process che il costrutto "CASE...WHEN", il componente sincronizzandosi sia con i rising_edge del clock che con i falling_edge del clock seleziona il numero da inviare al gestore_catodi e nel caso abbia meno di 4 bits aggiunge i bit necessari al vettore posti a '0'. Di seguito è presente il codice VHDL del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity selettore_display is port(
5     unita_secondi  : in  std_logic_vector (3 downto 0);
6     decine_secondi : in  std_logic_vector (2 downto 0);
7     unita_minuti  : in  std_logic_vector (3 downto 0);
8     decine_minuti : in  std_logic_vector (2 downto 0);
9     unita_ore     : in  std_logic_vector (3 downto 0);
10    decine_ore    : in  std_logic_vector (1 downto 0);
11    sel          : in  std_logic_vector (2 downto 0);
12    clk          : in  std_logic;
13    cifra        : out std_logic_vector (3 downto 0)
14 );
15 end selettore_display;
16
17 architecture behavioral of selettore_display is
18
19 begin
20
21 selettore : process (clk)
22
23 begin

```

```

25 case sel is
26   when "000" => cifra <= "00" & decine_ore;
27   when "001" => cifra <= unita_ore;
28   when "010" => cifra <= '0' & decine_minuti;
29   when "011" => cifra <= unita_minuti;
30   when "100" => cifra <= '0' & decine_secondi;
31   when "101" => cifra <= unita_secondi;
32   when others => cifra <= "0000";
33 end case;
34
35 end process;
36 end behavioral;

```

display Il componente display è stato sviluppato in modo strutturale ed è utilizzato come top entity di tutti gli elementi che permettono il funzionamento del display. La figura del componente è presente nel paragrafo schematici di questa sezione (vedi figura 40), mentre di seguito è presente il codice VHDL del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity display is port(
5   seconds      : in  std_logic_vector(5 downto 0);
6   minutes      : in  std_logic_vector(5 downto 0);
7   hours        : in  std_logic_vector(4 downto 0);
8   clk          : in  std_logic;
9   reset        : in  std_logic;
10  anodi        : out std_logic_vector(7 downto 0);
11  catodi       : out std_logic_vector(7 downto 0)
12 );
13 end display;
14
15 architecture structural of display is
16
17 component Divisore_freq is
18   generic (
19     f_in  : integer := 100000000;
20     f_out : integer := 1
21   );
22   port (
23     reset : in std_logic;
24     clock : in std_logic;
25     div   : out std_logic
26   );
27 end component;
28
29
30 component gestore_anodi is port (
31   sel      : in  std_logic_vector (2 downto 0);
32   anodi    : out std_logic_vector (7 downto 0)
33 );
34 end component;
35
36 component gestore_catodi is Port (

```

```

37         value : in STD_LOGIC_VECTOR (3 downto 0);
38         dot : in STD_LOGIC;
39         cathodes_dot : out STD_LOGIC_VECTOR (7 downto 0));
40 end component;
41
42 component selettore_display is port(
43     unita_secondi : in std_logic_vector (3 downto 0);
44     decine_secondi : in std_logic_vector (2 downto 0);
45     unita_minuti : in std_logic_vector (3 downto 0);
46     decine_minuti : in std_logic_vector (2 downto 0);
47     unita_ore : in std_logic_vector (3 downto 0);
48     decine_ore : in std_logic_vector (1 downto 0);
49     sel : in std_logic_vector (2 downto 0);
50     clk : in std_logic;
51     cifra : out std_logic_vector (3 downto 0)
52 );
53 end component;
54
55 component contatore_modulo_8_display is port(
56     clk : in std_logic;
57     en : in std_logic;
58     reset : in std_logic;
59     set : in std_logic;
60     v_set : in std_logic_vector(2 downto 0);
61     counter : out std_logic_vector(2 downto 0);
62     count : out std_logic);
63 end component;
64
65 component splitter is port(
66     numero_ingresso : in std_logic_vector (5 downto 0);
67     clk : in std_logic;
68     unita : out std_logic_vector (3 downto 0);
69     decine : out std_logic_vector (2 downto 0)
70 );
71 end component;
72
73
74 signal uscita_divisore, reset_contatore : std_logic;
75 signal uscita_contatore : std_logic_vector (2 downto 0); ---
76 signal uscita del contatore modulo 8
77 signal uscita_selettore_display : std_logic_vector (3 downto 0);
78 signal unita_secondi, unita_minuti, unita_ore : std_logic_vector (3 downto 0); ---
79 signal decine_secondi, decine_minuti, decine_ore : std_logic_vector (2 downto 0); ---
80 signal uscite degli splitter
81 constant frequenza_divisore : integer :=65000;
82
83 begin
84 -----
85 -----in questa sezione del componente inizializzo la frequenza in base alla quale cambio cifra
86     ed il contatore che indica quale cifra mostrare sul display
87 freq_div : Divisore_freq
88 generic map(
89     f_out =>frequenza_divisore
90 )
91 port map(
92     reset => '0',
93     clock => clk,
94     div    => uscita_divisore
95 );

```

```

93 reset_contatore <= uscita_contatore(2) and uscita_contatore(1) and not uscita_contatore(0);
94 contatore : contatore_modulo_8_display port map(
95     clk      => clk,
96     en       => uscita_divisore,
97     reset    => reset or reset_contatore,
98     set      => '0',
99     v_set    => "000",
100    counter => uscita_contatore);
101
102 -----
103 --in questa sezione del componente mi preoccupo di dividere le decine dai secondi dai
104 numerosi dei flip-flops
105 splitter_secondi : splitter port map(
106     numero_ingresso => seconds,
107     clk              => clk,
108     unita            => unita_secondi,
109     decine           => decine_secondi
110 );
111
112 splitter_minuti : splitter port map(
113     numero_ingresso => minutes,
114     clk              => clk,
115     unita            => unita_minuti,
116     decine           => decine_minuti
117 );
118
119 splitter_ore : splitter port map(
120     numero_ingresso (5) => '0',
121     numero_ingresso (4 downto 0) => hours,
122     clk                  => clk,
123     unita                => unita_ore,
124     decine               => decine_ore
125 );
126 -----
127 --in questa sezione del componente scelgo quale numero deve essere mostrato (in base alla
128 selettore_numero : selettore_display port map(
129     unita_secondi      => unita_secondi,
130     decine_secondi     => decine_secondi,
131     unita_minuti       => unita_minuti,
132     decine_minuti      => decine_minuti,
133     unita_ore          => unita_ore,
134     decine_ore          => decine_ore(1 downto 0),
135     sel                => uscita_contatore,
136     clk                => clk,
137     cifra              => uscita_selettore_display
138 );
139
140 gestore_catodi_display : gestore_catodi Port map (
141     value => uscita_selettore_display,
142     dot   => '0',
143     cathodes_dot => catodi
144 );
145
146 -----
147 --In questa sezione del componente scelgo quale cifra deve essere illuminata (in base al
148     valore del contatore)
149 gestore_anodi_display : gestore_anodi port map (
150     sel      => uscita_contatore,

```

```

150     anodi    => anodi
151 );
152 end structural;

```

cronometro_con_display

Il componente cronometro_con_display è stato sviluppato in modo strutturale e permette di collegare il display ed i debouncer tra di loro ed agli inputs della scheda. Di seguito è presente il codice VHDL del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity cronometro_con_display is port(
5     clk          : in  std_logic;
6     reset        : in  std_logic;
7     set          : in  std_logic;
8     set_switch_1 : in  std_logic_vector(5 downto 0);
9     set_switch_2 : in  std_logic_vector(5 downto 0);
10    anodi       : out std_logic_vector(7 downto 0);
11    catodi      : out std_logic_vector(7 downto 0)
12
13 );
14 end cronometro_con_display;
15
16 architecture structural of cronometro_con_display is
17
18 component Cronometro is port (
19     clk          : in  std_logic;
20     reset        : in  std_logic;
21     set          : in  std_logic;
22     set_switch_1 : in  std_logic_vector(5 downto 0);
23     set_switch_2 : in  std_logic_vector(5 downto 0);
24     seconds      : out std_logic_vector(5 downto 0);
25     minutes      : out std_logic_vector(5 downto 0);
26     hours        : out std_logic_vector(4 downto 0)
27 );
28 end component;
29
30 component display is port(
31     seconds      : in  std_logic_vector(5 downto 0);
32     minutes      : in  std_logic_vector(5 downto 0);
33     hours        : in  std_logic_vector(4 downto 0);
34     clk          : in  std_logic;
35     reset        : in  std_logic;
36     anodi       : out std_logic_vector(7 downto 0);
37     catodi      : out std_logic_vector(7 downto 0)
38 );
39 end component;
40
41 component ButtonDebouncer is
42     generic (
43         CLK_period: integer := 10; -- periodo del clock in nanosec
44         btn_noise_time: integer := 100000000 --durata dell'oscillazione in nanosec
45     );

```

```

46     Port ( RST : in STD_LOGIC;
47             CLK : in STD_LOGIC;
48             BTN : in STD_LOGIC;
49             CLEARED_BTN : out STD_LOGIC);
50 end component;
51
52 signal seconds, minutes      : std_logic_vector (5 downto 0);
53 signal hours                 : std_logic_vector (4 downto 0);
54 signal conn_set, conn_res    : std_logic;
55 signal cancellami, owo       : std_logic;
56 signal conn_save, conn_show : std_logic;
57
58 begin
59 Cron : Cronometro port map(
60     clk          => clk,
61     reset        => conn_res,
62     set          => conn_set,
63     set_switch_1 => set_switch_1,
64     set_switch_2 => set_switch_2,
65     seconds      => seconds,
66     minutes      => minutes,
67     hours        => hours
68 );
69
70 dis : Display Port map(
71     seconds => seconds,
72     minutes => minutes,
73     hours   => hours,
74     clk     => clk,
75     reset   => conn_res,
76     anodi   => anodi,
77     catodi  => catodi
78 );
79
80 debouncer_set : ButtonDebouncer port map(
81     rst => '0',
82     clk => clk,
83     btn => set,
84     cleared_btn => conn_set
85 );
86
87 debouncer_reset : ButtonDebouncer port map(
88     rst => '0',
89     clk => clk,
90     btn => reset,
91     cleared_btn => conn_res
92 );
93
94 end structural;

```

5.2.3 Implementazione su board

In questa sezione é presente il codice del file di constraint usato per la scheda.

```
1 ## Clock signal
```

```

2 | set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { clk }]; #
3 |   IO_L12P_T1_MRCC_35 Sch=clk100mhz
4 |
5 | ##Switches
6 | set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[0] }
7 |   ]; #IO_L24N_T3_RSO_15 Sch=sw[0]
8 | set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[1] }
9 |   ]; #IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
10 | set_property -dict { PACKAGE_PIN M13     IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[2] }
11 |   ]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
12 | set_property -dict { PACKAGE_PIN R15     IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[3] }
13 |   ]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
14 | set_property -dict { PACKAGE_PIN R17     IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[4] }
15 |   ]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
16 | set_property -dict { PACKAGE_PIN T18     IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[5] }
17 |   ]; #IO_L7N_T1_D10_14 Sch=sw[5]
18 | set_property -dict { PACKAGE_PIN U18     IOSTANDARD LVCMOS33 } [get_ports { set_switch_2[0] }
19 |   ]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
20 | set_property -dict { PACKAGE_PIN R13     IOSTANDARD LVCMOS33 } [get_ports { set_switch_2[1] }
21 |   ]; #IO_L5N_T0_D07_14 Sch=sw[7]
22 | set_property -dict { PACKAGE_PIN T8      IOSTANDARD LVCMOS18 } [get_ports { set_switch_2[2] }
23 |   ]; #IO_L24N_T3_34 Sch=sw[8]
24 | set_property -dict { PACKAGE_PIN U8      IOSTANDARD LVCMOS18 } [get_ports { set_switch_2[3] }
25 |   ]; #IO_25_34 Sch=sw[9]
26 | set_property -dict { PACKAGE_PIN R16     IOSTANDARD LVCMOS33 } [get_ports { set_switch_2[4] }
27 |   ]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
28 | set_property -dict { PACKAGE_PIN T13     IOSTANDARD LVCMOS33 } [get_ports { set_switch_2[5] }
29 |   ]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
30 |
31 | ##7 segment display
32 | set_property -dict { PACKAGE_PIN T10     IOSTANDARD LVCMOS33 } [get_ports { catodi[0] }]; #
33 |   IO_L24N_T3_A00_D16_14 Sch=ca
34 | set_property -dict { PACKAGE_PIN R10     IOSTANDARD LVCMOS33 } [get_ports { catodi[1] }]; #
35 |   IO_25_14 Sch=cb
36 | set_property -dict { PACKAGE_PIN K16     IOSTANDARD LVCMOS33 } [get_ports { catodi[2] }]; #
37 |   IO_25_15 Sch=cc
38 | set_property -dict { PACKAGE_PIN K13     IOSTANDARD LVCMOS33 } [get_ports { catodi[3] }]; #
39 |   IO_L17P_T2_A26_15 Sch=cd
40 | set_property -dict { PACKAGE_PIN P15     IOSTANDARD LVCMOS33 } [get_ports { catodi[4] }]; #
41 |   IO_L13P_T2_MRCC_14 Sch=ce
42 | set_property -dict { PACKAGE_PIN T11     IOSTANDARD LVCMOS33 } [get_ports { catodi[5] }]; #
43 |   IO_L19P_T3_A10_D26_14 Sch=cf
44 | set_property -dict { PACKAGE_PIN L18     IOSTANDARD LVCMOS33 } [get_ports { catodi[6] }]; #
45 |   IO_L4P_T0_D04_14 Sch=cg
46 | set_property -dict { PACKAGE_PIN H15     IOSTANDARD LVCMOS33 } [get_ports { catodi[7] }]; #
47 |   IO_L19N_T3_A21_VREF_15 Sch=dp
48 | set_property -dict { PACKAGE_PIN J17     IOSTANDARD LVCMOS33 } [get_ports { anodi[0] }]; #
49 |   IO_L23P_T3_FOE_B_15 Sch=an[0]
50 | set_property -dict { PACKAGE_PIN J18     IOSTANDARD LVCMOS33 } [get_ports { anodi[1] }]; #
51 |   IO_L23N_T3_FWE_B_15 Sch=an[1]
52 | set_property -dict { PACKAGE_PIN T9      IOSTANDARD LVCMOS33 } [get_ports { anodi[2] }]; #
53 |   IO_L24P_T3_A01_D17_14 Sch=an[2]
54 | set_property -dict { PACKAGE_PIN J14     IOSTANDARD LVCMOS33 } [get_ports { anodi[3] }]; #
55 |   IO_L19P_T3_A22_15 Sch=an[3]
56 | set_property -dict { PACKAGE_PIN P14     IOSTANDARD LVCMOS33 } [get_ports { anodi[4] }]; #
57 |   IO_L8N_T1_D12_14 Sch=an[4]
58 | set_property -dict { PACKAGE_PIN T14     IOSTANDARD LVCMOS33 } [get_ports { anodi[5] }]; #
59 |   IO_L14P_T2_SRCC_14 Sch=an[5]
60 | set_property -dict { PACKAGE_PIN K2      IOSTANDARD LVCMOS33 } [get_ports { anodi[6] }]; #

```

```

    IO_L23P_T3_35 Sch=an[6]
35 set_property -dict { PACKAGE_PIN U13      IOSTANDARD LVCMOS33 } [get_ports { anodi[7] }]; #
    IO_L23N_T3_A02_D18_14 Sch=an[7]

36
37 ##Buttons
38 set_property -dict { PACKAGE_PIN C12      IOSTANDARD LVCMOS33 } [get_ports { reset }]; #
    IO_L3P_TO_DQS_AD1P_15 Sch=cpu_resetn
39 set_property -dict { PACKAGE_PIN N17      IOSTANDARD LVCMOS33 } [get_ports { set }]; #
    IO_L9P_T1_DQS_14 Sch=btnc

```

5.3 Memorizzazione di intertempi e visualizzazione

In questa parte dell'esercizio ci si preoccupa di aggiungere la funzione di poter salvare più intertempi all'interno del componente e di poterli mostrare uno alla volta sul display della scheda tramite la pressione ripetuta di un pulsante. Anche questo punto come il precedente è stato sviluppato in modo graduale, quindi prima della lettura di questo punto si rimanda alla lettura del punto precedente 5.2.

Traccia

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).

5.3.1 Schematici

Per poter comprendere completamente il funzionamento dei componenti che vengono aggiunti si rimanda alla visione degli schematici della parte 1:5.1.1 e 5.2.1 (per vedere la struttura base del cronometro) e della parte: 5.2.1 (per vedere la struttura del display).

Shift_register_3

Lo shift_register_3 è un componente che permette di salvare 3 bits d'informazione tramite un segnale di input ed un segnale di enable in ingresso. Le informazioni del registro sono mostrate in parallelo su un vettore (output) in uscita al componente. In figura 42 lo schematico del componente.

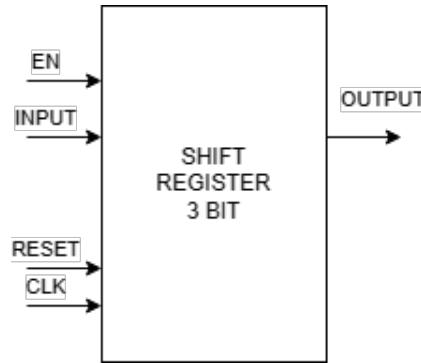


Figura 42: componente shift register 3

mux_4_1

Il mux_4_1 utilizzato è lo stesso di quello sviluppato nell'esercizio 1, è presente la descrizione del componente al paragrafo: 1.1.1

contatore_mod3_cronometro

Il contatore modulo 3 è stato sviluppato come il contatore modulo 64, con la differenza che il conteggio si ferma a 3 e quindi si hanno solo 2 bit in uscita (essendo stato già trattato si rimanda alla figura: 30. L'uscita del contatore viene utilizzata come selezione per il mux_4_1.

cronometro

Per implementare la nuova richiesta della traccia è stato modificato il componente del cronometro, infatti sono state aggiunte 2 funzioni:

- memorizzazione dei dati degli intertempi per un numero N (N=3); (shift_register_3)
- gestione del valore da visualizzare sul display; (mux_4_1, cont_mod4_cronometro)

memorizzazione dei dati degli intertempi per un numero N (N=3) Per memorizzare i valori degli intertempi sono stati aggiunti degli shift_register (SIPO) di 3 bits, per poter salvare un intertempo è stato necessario avere uno shift_register_3 per ogni filo di conteggio in uscita ai contatori (secondi, minuti ed ore). A tutti gli shift_register_3 è stato inserito come enable il segnale btn_save. In questo modo si ha una fila di shift register che ad ogni pressione del bottone btn_save salvano il valore in uscita dai contatori del cronometro e si hanno sempre disponibili le uscite degli shift_register.

gestione del valore da visualizzare sul display Per gestire il valore da visualizzare sul display è stato utilizzato il mux_4_1, che sceglie tra 4 possibili ingressi un'uscita da mandare in uscita al componente. Gli ingressi sono:

- le 3 uscite dello shift_register_3;
- l'ingresso allo shift_register_3, che rappresenta il valore attuale del cronometro.

Il segnale di selezione in ingresso è dato dal cont_mod4_cronometro, che permette di scegliere quale ingresso ai mux [4:1] deve essere mandato in uscita. Infine per permettere all'utente di visualizzare gli intertempi (o il valore del cronometro) al cont_mod4_cronometro è stato inserito il btn_show come enable. Il componente modificato è in figura 43.

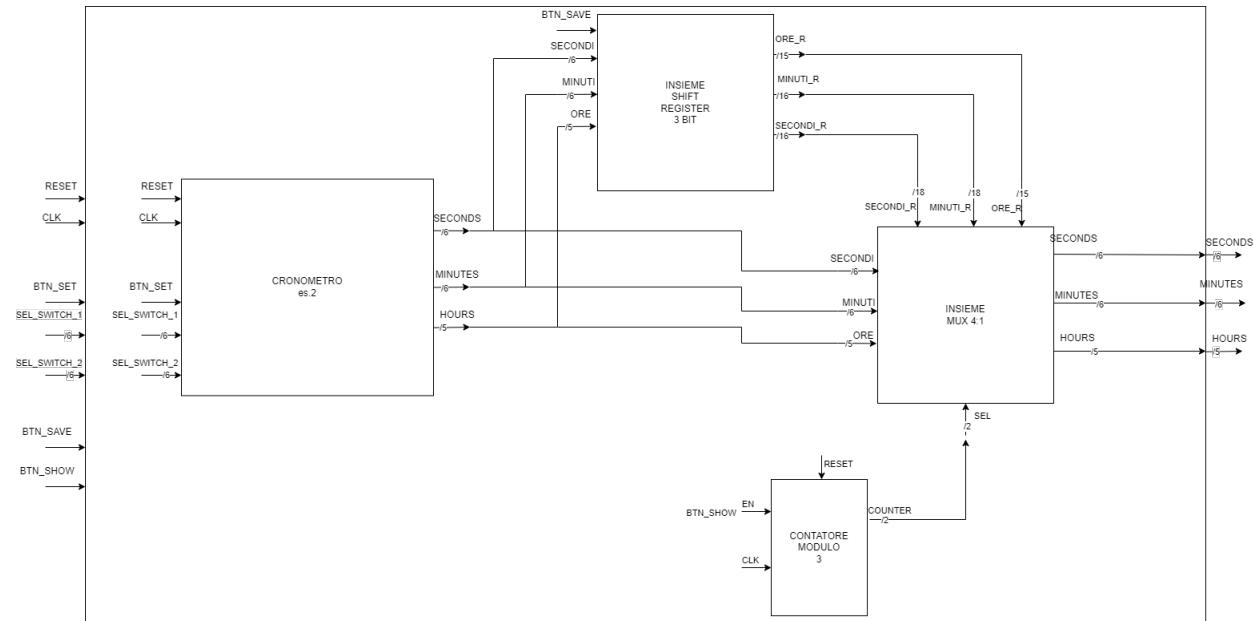


Figura 43: componente cronometro

Cronometro con display su board

Una volta integrati i registri per memorizzare dei valori e i multiplexer per la selezione di un valore nei registri nella struttura del cronometro abbiamo collegato questa nuova struttura del cronometro al display. Inoltre per utilizzare le funzioni di salvataggio e visualizzazione degli intertempi e di set e reset del cronometro abbiamo utilizzati dei bottoni. Quindi sono stati necessari anche dei debouncer per pulire il segnale proveniente dai bottoni. In figura 44 lo schematico del sistema complessivo.

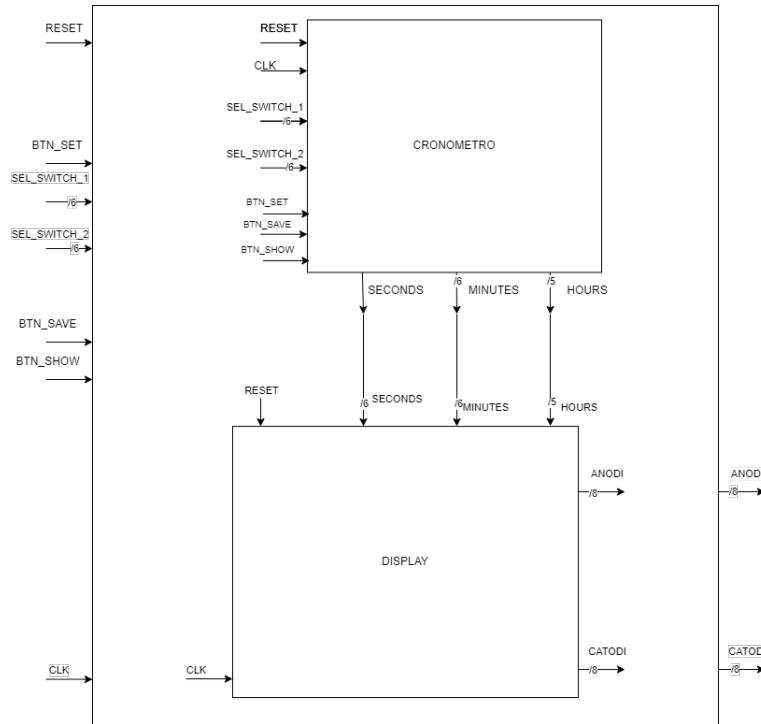


Figura 44: cronometro_con_display

5.3.2 Codice VHDL

Per poter comprendere completamente il funzionamento dei componenti che vengono aggiunti si rimanda alla descrizione del componente cronometro e display visti nei paragrafi precedenti. Inoltre avendo già trattato sia il mux_4_1 (nel paragrafo 1.1.2) sia il codice di un contatore (nel paragrafo 5.1.2), viene omesso il reinserimento del codice dei 2 componenti.

Shift_register_3

Il componente dello shift_register_3 è stato descritto in modo comportamentale attraverso il costrutto del process. Il registro ha 2 segnali in ingresso: un enable ed un input (che permettono di salvare un ingresso e di eseguire lo shift dei valori precedenti) ed 1 segnale in uscita: output (un vettore che permette di vedere costantemente il valore del registro). I valori che vengono acquisiti in input, vengono salvati all'interno di un segnale interno al componente (che viene sintetizzato come una serie di flip-flop), che viene successivamente mandato in uscita. Di seguito è presente il codice VHDL del componente.

```

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

```

4 entity shift_register_3 is port(
5     clk      : in std_logic;
6     en       : in std_logic;
7     reset    : in std_logic;
8     input    : in std_logic;
9     output   : out std_logic_vector(0 to 2)
10);
11 end shift_register_3;
12
13 architecture Behavioral of shift_register_3 is
14 signal reg: std_logic_vector(0 to 2);
15 begin
16
17 process (clk)
18 begin
19
20 if (rising_edge(clk)) then
21
22     if (reset = '1') then
23         reg <= (others => '0');
24     end if;
25
26     if (en = '1') then
27         reg(0) <= input;
28         reg(1) <= reg(0);
29         reg(2) <= reg(1);
30
31     end if;
32 end if;
33
34 end process;
35
36 output (0 to 2) <= reg (0 to 2);
37
38 end Behavioral;

```

cronometro

Il componente cronometro é stato sviluppato in modo strutturale e descrive la nuova struttura del componente (presente in figura 43). Di seguito é presente il codice VHDL del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Cronometro is port (
5     clk      : in std_logic;
6     reset    : in std_logic;
7     set      : in std_logic;
8     btn_save : in std_logic;
9     btn_show : in std_logic;
10    set_switch_1 : in std_logic_vector(5 downto 0);
11    set_switch_2 : in std_logic_vector(5 downto 0);
12    seconds   : out std_logic_vector(5 downto 0);
13    minutes   : out std_logic_vector(5 downto 0);
14    hours     : out std_logic_vector(4 downto 0)

```

```

15 );
16 end Cronometro;
17
18
19 architecture Structural of Cronometro is
20
21 component Counter_mod60 is port (
22     clk      : in std_logic;
23     en       : in std_logic;
24     reset    : in std_logic;
25     set      : in std_logic;
26     v_set    : in std_logic_vector(5 downto 0);
27     counter : out std_logic_vector(5 downto 0);
28     count   : out std_logic
29 );
30 end component;
31
32 component Counter_mod24_struct is port (
33     clk : in std_logic;
34     en : in std_logic;
35     reset : in std_logic;
36     set : in std_logic;
37     v_set : in std_logic_vector(4 downto 0);
38     counter : out std_logic_vector(4 downto 0);
39     count : out std_logic
40 );
41 end component;
42
43 component Divisore_freq is
44     generic (
45         f_in : integer := 100000000;
46         f_out : integer := 1
47     );
48     port (
49         reset : in std_logic;
50         clock : in std_logic;
51         div   : out std_logic
52     );
53 end component;
54
55 component CU_set is port (
56     reset      : in std_logic;
57     en        : in std_logic;
58     btn       : in std_logic;
59     clk        : in std_logic;
60     set_switch_1 : in std_logic_vector(5 downto 0);
61     set_switch_2 : in std_logic_vector(5 downto 0);
62     set_out    : out std_logic;
63     v_set_s   : out std_logic_vector(5 downto 0);
64     v_set_m   : out std_logic_vector(5 downto 0);
65     v_set_h   : out std_logic_vector(4 downto 0)
66 );
67 end component;
68
69 component shift_register_3 is port(
70     clk      : in std_logic;
71     en       : in std_logic;
72     reset    : in std_logic;
73     input    : in std_logic;
74     output   : out std_logic_vector(0 to 2)

```

```

75 );
76 end component;
77
78 component mux_4_1 is port (
79     input: in std_logic_vector(0 to 3);
80     s: in std_logic_vector(1 downto 0);
81     output: out std_logic
82 );
83 end component;
84
85 component cont_mod4_cronometro is port(
86     clk      : in std_logic;
87     en       : in std_logic;
88     reset    : in std_logic;
89     counter : out std_logic_vector(1 downto 0)
90 );
91 end component;
92
93 signal cont          : std_logic_vector(1 downto 0);
94 signal en, temp_set   : std_logic;
95 signal conn_m, conn_s  : std_logic_vector(5 downto 0);
96 signal conn_h         : std_logic_vector(4 downto 0);
97
98 signal secondi, minuti : std_logic_vector(5 downto 0);
99 signal ore             : std_logic_vector(4 downto 0);
100
101 signal selezione_mux  : std_logic_vector(1 downto 0);
102
103 type reg_out_sm is array (0 to 5) of std_logic_vector(0 to 2);
104 type reg_out_h is array (0 to 4) of std_logic_vector(0 to 2);
105
106 signal reg_out_s : reg_out_sm;
107 signal reg_out_m : reg_out_sm;
108 signal reg_out_o : reg_out_h;
109
110
111 begin
112
113 div : Divisore_freq
114     port map (
115         reset => '0',
116         clock => clk,
117         div => en
118 );
119
120 cont_secondi : Counter_mod60
121     port map (
122         clk => clk,
123         en => en or reset or temp_set,
124         reset => reset,
125         set => temp_set,
126         v_set => conn_s,
127         counter => secondi,
128         count => cont(0)
129 );
130 cont_minuti : Counter_mod60
131     port map (
132         clk => clk,
133         en => (en and cont(0)) or (reset) or (temp_set),
134         reset => reset,

```

```

135     set => temp_set,
136     v_set => conn_m,
137     counter => minuti,
138     count => cont(1)
139 );
140
141 cont_ore : Counter_mod24_struct
142   port map (
143     clk => clk,
144     en => (en and cont(1) and cont(0)) or (reset) or (temp_set),
145     reset => reset,
146     set => temp_set,
147     v_set => conn_h,
148     counter => ore
149 );
150
151 reg_seconds : for i in 0 to 5 generate
152   reg_s : shift_register_3
153   port map (
154     clk      => clk,
155     en       => btn_save,
156     reset    => reset,
157     input    => secondi(5-i),
158     output   => reg_out_s(i)
159 );
160 end generate reg_seconds;
161
162 reg_minutes : for i in 0 to 5 generate
163   reg_m : shift_register_3
164   port map (
165     clk      => clk,
166     en       => btn_save,
167     reset    => reset,
168     input    => minuti(5-i),
169     output   => reg_out_m(i)
170 );
171 end generate reg_minutes;
172
173 reg_hours : for i in 0 to 4 generate
174   reg_o : shift_register_3
175   port map (
176     clk      => clk,
177     en       => btn_save,
178     reset    => reset,
179     input    => ore(4-i),
180     output   => reg_out_o(i)
181 );
182 end generate reg_hours;
183
184 -----
185 -----
186
187 cont_mod4 : cont_mod4_cronometro port map (
188   clk      => clk,
189   en       => btn_show,
190   reset    => reset,
191   counter  => selezione_mux
192 );
193
194 -----

```

```

195 -----
196
197 mux_seconds : for i in 0 to 5 generate
198     mux_s : mux_4_1
199     port map (
200         input (0)      => secondi(5-i),
201         input (1 to 3) => reg_out_s(i),
202         s              => selezione_mux,
203         output         => seconds(5-i)
204 );
205 end generate mux_seconds;
206
207 mux_minutes : for i in 0 to 5 generate
208     mux_m : mux_4_1
209     port map (
210         input (0)      => minuti(5-i),
211         input (1 to 3) => reg_out_m(i),
212         s              => selezione_mux,
213         output         => minutes(5-i)
214 );
215 end generate mux_minutes;
216
217 mux_ore : for i in 0 to 4 generate
218     mux_o : mux_4_1
219     port map (
220         input (0)      => ore(4-i),
221         input (1 to 3) => reg_out_o(i),
222         s              => selezione_mux,
223         output         => hours(4-i)
224 );
225 end generate mux_ore;
226
227 funzione_set : CU_set port map (
228     reset          => reset,
229     en             => en,
230     btn            => set,
231     clk            => clk,
232     set_switch_1   => set_switch_1,
233     set_switch_2   => set_switch_2,
234     set_out         => temp_set,
235     v_set_s        => conn_s,
236     v_set_m        => conn_m,
237     v_set_h        => conn_h
238 );
239
240 end Structural;

```

5.3.3 Implementazione su board

In questa sezione è presente il codice del file di constraint usato per la scheda.

```

1
2 ## Clock signal
3 set_property -dict { PACKAGE_PIN E3      IO_STANDARD LVCMOS33 } [get_ports { clk }]; #
4     IO_L12P_T1_MRCC_35 Sch=clk100mhz
4 create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];

```

```

5
6 ##Switches
7 set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[0]
8     }]; #IO_L24N_T3_RSO_15 Sch=sw[0]
9 set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[1]
10    }]; #IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
11 set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[2]
12    }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
13 set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[3]
14    }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
15 set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[4]
16    }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
17 set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMOS33 } [get_ports { set_switch_1[5]
18    }]; #IO_L7N_T1_D10_14 Sch=sw[5]
19 set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMOS33 } [get_ports { set_switch_2[0]
20    }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
21 set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMOS33 } [get_ports { set_switch_2[1]
22    }]; #IO_L5N_T0_D07_14 Sch=sw[7]
23 set_property -dict { PACKAGE_PIN T8       IOSTANDARD LVCMOS18 } [get_ports { set_switch_2[2]
24    }]; #IO_L24N_T3_34 Sch=sw[8]
25 set_property -dict { PACKAGE_PIN U8       IOSTANDARD LVCMOS18 } [get_ports { set_switch_2[3]
26    }]; #IO_25_34 Sch=sw[9]
27 set_property -dict { PACKAGE_PIN R16      IOSTANDARD LVCMOS33 } [get_ports { set_switch_2[4]
28    }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
29 set_property -dict { PACKAGE_PIN T13      IOSTANDARD LVCMOS33 } [get_ports { set_switch_2[5]
30    }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
31
32 ##7 segment display
33 set_property -dict { PACKAGE_PIN T10      IOSTANDARD LVCMOS33 } [get_ports { catodi[0] }]; #
34     IO_L24N_T3_A00_D16_14 Sch=ca
35 set_property -dict { PACKAGE_PIN R10      IOSTANDARD LVCMOS33 } [get_ports { catodi[1] }]; #
36     IO_25_14 Sch=cb
37 set_property -dict { PACKAGE_PIN K16      IOSTANDARD LVCMOS33 } [get_ports { catodi[2] }]; #
38     IO_25_15 Sch=cc
39 set_property -dict { PACKAGE_PIN K13      IOSTANDARD LVCMOS33 } [get_ports { catodi[3] }]; #
40     IO_L17P_T2_A26_15 Sch=cd
41 set_property -dict { PACKAGE_PIN P15      IOSTANDARD LVCMOS33 } [get_ports { catodi[4] }]; #
42     IO_L13P_T2_MRCC_14 Sch=ce
43 set_property -dict { PACKAGE_PIN T11      IOSTANDARD LVCMOS33 } [get_ports { catodi[5] }]; #
44     IO_L19P_T3_A10_D26_14 Sch=cf
45 set_property -dict { PACKAGE_PIN L18      IOSTANDARD LVCMOS33 } [get_ports { catodi[6] }]; #
46     IO_L4P_T0_D04_14 Sch=cg
47 set_property -dict { PACKAGE_PIN H15      IOSTANDARD LVCMOS33 } [get_ports { catodi[7] }]; #
48     IO_L19N_T3_A21_VREF_15 Sch=dp
49 set_property -dict { PACKAGE_PIN J17      IOSTANDARD LVCMOS33 } [get_ports { anodi[0] }]; #
50     IO_L23P_T3_FOE_B_15 Sch=an[0]
51 set_property -dict { PACKAGE_PIN J18      IOSTANDARD LVCMOS33 } [get_ports { anodi[1] }]; #
52     IO_L23N_T3_FWE_B_15 Sch=an[1]
53 set_property -dict { PACKAGE_PIN T9       IOSTANDARD LVCMOS33 } [get_ports { anodi[2] }]; #
54     IO_L24P_T3_A01_D17_14 Sch=an[2]
55 set_property -dict { PACKAGE_PIN J14      IOSTANDARD LVCMOS33 } [get_ports { anodi[3] }]; #
56     IO_L19P_T3_A22_15 Sch=an[3]
57 set_property -dict { PACKAGE_PIN P14      IOSTANDARD LVCMOS33 } [get_ports { anodi[4] }]; #
58     IO_L8N_T1_D12_14 Sch=an[4]
59 set_property -dict { PACKAGE_PIN T14      IOSTANDARD LVCMOS33 } [get_ports { anodi[5] }]; #
60     IO_L14P_T2_SRCC_14 Sch=an[5]
61 set_property -dict { PACKAGE_PIN K2       IOSTANDARD LVCMOS33 } [get_ports { anodi[6] }]; #
62     IO_L23P_T3_35 Sch=an[6]
63 set_property -dict { PACKAGE_PIN U13      IOSTANDARD LVCMOS33 } [get_ports { anodi[7] }]; #
64     IO_L23N_T3_A02_D18_14 Sch=an[7]

```

```
37
38 ##Buttons
39 set_property -dict { PACKAGE_PIN C12    IO_STANDARD LVCMS33 } [get_ports { reset }]; #
      IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
40 set_property -dict { PACKAGE_PIN N17    IO_STANDARD LVCMS33 } [get_ports { set }]; #
      IO_L9P_T1_DQS_14 Sch=btnc
41 set_property -dict { PACKAGE_PIN M18    IO_STANDARD LVCMS33 } [get_ports { btn_save }]; #
      IO_L4N_T0_D05_14 Sch=btnu
42 set_property -dict { PACKAGE_PIN P17    IO_STANDARD LVCMS33 } [get_ports { btn_show }]; #
      IO_L12P_T1_MRCC_14 Sch=btln
```

6 Sistema di testing

In questo esercizio progetteremo un sistema in grado di testare il funzionamento di una semplice macchina combinatoria ponendo in ingresso una serie di stringhe memorizzata in memoria e osservando le uscite della stessa, che saranno anche memorizzate in un'altra memoria.

6.1 Progettazione in VHDL

Traccia

Progettare, implementare in VHDL e verificare mediante simulazione un sistema in grado di testare in maniera automatica una macchina combinatoria M avente 4 ingressi e 3 uscite binarie sottoponendole N ingressi diversi (si considerino una macchina M e un numero di input N a scelta dello studente).

Gli N valori di input per il test devono essere letti da una ROM, in cui essi sono precaricati, in corrispondenza di un segnale read. Le N uscite fornite della macchina in corrispondenza di ciascuno degli input devono essere memorizzate in una memoria interna, che deve poter essere svuotata in qualsiasi momento in presenza di un segnale di reset.

6.1.1 Schematici

Per progettare il sistema definito dalla traccia sarà necessario, prima di tutto, introdurre dei componenti che non abbiamo ancora trattato. Saranno necessari:

- una ROM, per mantenere le stringhe da porre in ingresso alla macchina combinatoria
- un contatore, per contare gli N valori in ingresso alla macchina combinatoria
- una memoria, per memorizzare le stringhe in uscita dalla macchina combinatoria
- una macchina combinatoria, che trasformerà la stringa in input in una stringa in output

Procediamo all'analisi di ciascun componente per, infine, arrivare all'architettura completa del sistema richiesto dalla traccia

ROM

Una ROM (Read Only Memory) è una memoria a sola lettura e non volatile. Essendo una memoria a sola lettura i valori contenuti al suo interno sono fissati e non modificabili durante il funzionamento del sistema.

Quindi per il funzionamento della ROM sono necessari: un ingresso con l'indirizzo da cui leggere il valore ed un segnale di read in ingresso, in presenza del quale viene letto il valore presente all'indirizzo specificato. In uscita avremo il valore effettivamente presente in memoria. Nel nostro caso il segnale di read non è necessario, in quanto la ROM deve presentare in uscita il valore memorizzato presente all'indirizzo specificato in ingresso, e deve mantenere quel valore in uscita finché non varia l'indirizzo, in modo che la macchina combinatoria possa svolgere le sue operazioni. Abbiamo scelto di implementare una ROM con 16 locazioni di memoria di 4 bit ciascuna (i 4 bit che entreranno nella macchina combinatoria), quindi sia il dato in output alla rom che l'indirizzo saranno su 4 bit.

Quindi la nostra ROM avrà uno schematico di questo tipo:

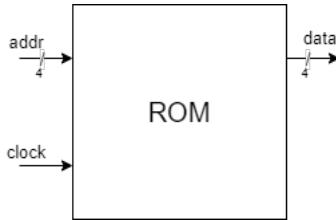


Figura 45: ROM

Contatore

Il contatore è un componente che abbiamo già progettato ed utilizzato nella sezione precedente, quindi utilizzeremo l'implementazione comportamentale vista nella sezione 5.1.1. Riportiamo qui lo schematico del componente utilizzato:

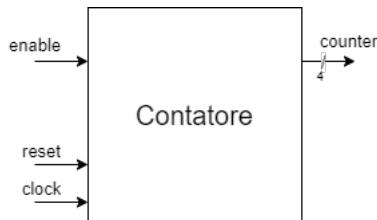


Figura 46: Contatore

Memoria

La memoria che viene richiesta nella traccia è una memoria su cui si può scrivere, oltre poter leggere, e che abbia un segnale di reset in grado di resettare tutte le celle di memoria.

Quindi è necessario un segnale di ingresso per l'indirizzo su cui leggere e uno per quello su cui scrivere, un segnale di scrittura in presenza del quale la cella di memoria viene scritta, due segnali di input e output dati e un segnale di reset che permette di resettare la memoria. In particolare, nello nostra implementazione in uscita abbiamo sempre il valore memorizzato nella cella di memoria con indirizzo `read_addr`, mentre l'ingresso viene memorizzato all'indirizzo `write_addr` quando si alza il segnale `write_en`. La nostra memoria deve contenere 16 celle di memoria di 3 bit ciascuno (i 3 bit in uscita dalla macchina combinatoria), per questo gli indirizzi saranno a 4 bit.

Quanto detto è riassunto nello schematico sottostante:

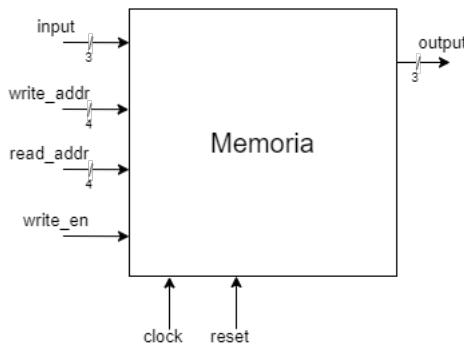


Figura 47: Memoria

Macchina combinatoria

L'unico vincolo che è stato imposto sulla macchina combinatoria è che abbia 4 ingressi e 3 uscite, quindi lo schematico sarà di questo tipo:

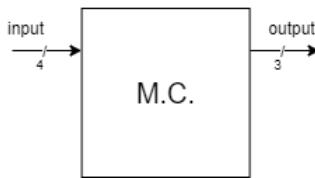


Figura 48: Memoria

Sistema completo

Definiti i componenti necessari non resta che connetterli per ottenere il sistema richiesto nella traccia. Il segnale di read è l'enable di un contatore, l'uscita di conteggio del contatore viene usata direttamente collegata all'indirizzo della ROM, che quindi cambierà valore in uscita ogni volta che il segnale di read si alza. L'uscita della ROM è collegata alla macchina combinatoria e l'uscita di questa è collegata come input della rom. Per il segnale di write_en abbiamo supposto che la macchina combinatoria presenti la sua uscita quasi istantaneamente, quindi possano usare lo stesso segnale di read come segnale di write_en per la memoria. In output al sistema complessivo vi sarà l'uscita della macchina combinatoria, mentre l'unico ingresso, oltre clock e reset, vi sarà il segnale di read.

Di seguito lo schematico del sistema:

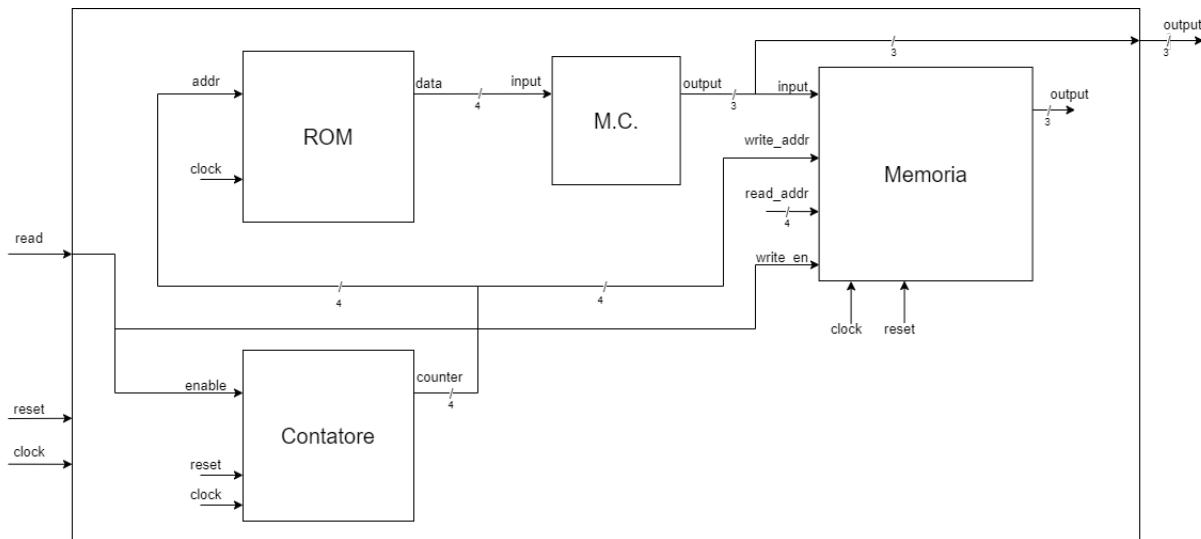


Figura 49: Sistema totale

6.1.2 Codice VHDL

Esaminiamo ora il codice VHDL, prima dei componenti singoli, ed infine del sistema complessivo.

ROM

Come presentato precedentemente l'interfaccia della ROM prevede in ingresso il clock *clk* e l'indirizzo *addr* e in uscita i dati memorizzati all'indirizzo *addr*. La particolarità di questo componente è la definizione di un tipo ad hoc per creare un segnale che possa contenere i dati

della ROM. Il nuovo tipo è un array di 16 array di 4 bit e memorizzati nella rom abbiamo deciso di inserire i valori esadecimale in ordine.

L'implementazione della ROM è di tipo comportamentale, si utilizza il costrutto process con il clock nella sensitivity list e, se ci si trova sul fronte di salita del clock, viene messo in uscita la stringa memorizzata all'indirizzo in ingresso. E' stata utilizzata la funzione conv_integer per convertire il vettore di bit *addr* in un intero da poter usare per recuperare un array nel segnale *ROM*. Di seguito il codice della ROM:

```

1 entity ROM is port(
2     clk : in std_logic;
3     addr : in std_logic_vector(3 downto 0);
4     data : out std_logic_vector(3 downto 0)
5 );
6 end ROM;
7
8 architecture behavioral of ROM is
9 type rom_type is array (0 to 15) of std_logic_vector(3 downto 0);
10 signal ROM : rom_type := (
11 X"0", X"1", X"2", X"3", X"4", X"5", X"6", X"7",
12 X"8", X"9", X"A", X"B", X"C", X"D", X"E", X"F"
13 );
14
15 begin
16 process(CLK)
17 begin
18     if rising_edge(CLK) then
19         DATA <= ROM(conv_integer(ADDR));
20     end if;
21 end process;
22 end behavioral;
```

Contatore

Il codice VHDL del contatore utilizzato non presenta nulla di nuovo rispetto quello visto nel paragrafo 5.1.2, anzi, in questo caso, è molto semplificato, in quanto è necessario solo il segnale di conteggio per i nostri scopi. Di seguito il codice del componente:

```

1 entity counter_mod16 is port (
2     clock : in STD_LOGIC;
3     reset : in STD_LOGIC;
4     enable : in STD_LOGIC;
5     counter : out STD_LOGIC_VECTOR (3 downto 0)
6 );
7 end counter_mod16;
8
9 architecture Behavioral of counter_mod16 is
10 signal c : std_logic_vector (3 downto 0) := (others => '0');
11
12 begin
```

```

13 counter <= c;
14 counter_process: process(clock)
15 begin
16   if (clock'event AND clock = '1') then
17     if (reset = '1') then
18       c <= (others => '0');
19     else
20       if (enable='1') then
21         c <= std_logic_vector(unsigned(c) + 1);
22       end if;
23     end if;
24   end if;
25 end process;
26 end Behavioral;

```

Memoria

Come presentato precedentemente l’interfaccia della memoria prevede in ingresso il clock *clock*, gli indirizzi *write_addr* e *read_addr*, l’abilitazione per la scrittura *write_en* e la stringa di bit *input*, mentre in uscita *output* i dati memorizzati all’indirizzo *read_addr*. Anche in questo caso è stato utilizzato un tipo definito dall’utente creare il segnale che manterrà i valori memorizzati.

L’implementazione della memoria è molto simile a quella della ROM. Presenta solamente un ulteriore controllo sul segnale *write_en*, se questo segnale è alto allora viene posizionato il segnale in *input* all’indirizzo *write_en* della memoria. Di seguito il codice della memoria:

```

1 entity memoria is port (
2   clock      : in std_logic;
3   reset      : in std_logic;
4   input       : in std_logic_vector(2 downto 0);
5   write_addr : in std_logic_vector(3 downto 0);
6   read_addr  : in std_logic_vector(3 downto 0);
7   write_en   : in std_logic;
8   output      : out std_logic_vector(2 downto 0)
9 );
10 end memoria;
11
12 architecture Behavioral of memoria is
13 type ram_type is array(0 to 15) of std_logic_vector(2 downto 0);
14 signal ram : ram_type := (others => "000");
15
16 begin
17 process(clock)
18 begin
19   if (rising_edge(clock)) then
20     if (write_en = '1') then
21       ram(conv_integer(write_addr)) <= input;
22     end if;
23     output <= ram(conv_integer(read_addr));
24   end if;
25 end process;

```

26 | end Behavioral;

Macchina combinatoria

La macchina combinatoria che abbiamo implementato è molto semplice, infatti si occupa solo di eliminare il bit più significativo dalla stringa di bit in ingresso. Di seguito il semplice codice:

```

1 entity combinatorius is port (
2     input    : in std_logic_vector(3 downto 0);
3     output   : out std_logic_vector(2 downto 0)
4 );
5 end combinatorius;
6
7 architecture Dataflow of combinatorius is
8 begin
9     output <= input(2 downto 0);
10    end Dataflow;

```

Sistema completo

Il sistema complessivo è una composizione dei componenti precedentemente illustrati. Quindi è stato utilizzato il paradigma di progettazione strutturale. Infatti tramite l'utilizzo di segnali interni sono stati connessi i vari componenti come illustrato nello schematico precedentemente visto. Di seguito il codice del sistema:

```

1 entity sistema_testing is port (
2     clock    : in std_logic;
3     read     : in std_logic;
4     reset    : in std_logic;
5     output   : out std_logic_vector(2 downto 0)
6 );
7 end sistema_testing;
8
9 architecture structural of sistema_testing is
10
11 component counter_mod16 is port (
12     clock    : in STD_LOGIC;
13     reset    : in STD_LOGIC;
14     enable   : in STD_LOGIC;
15     counter  : out STD_LOGIC_VECTOR (3 downto 0)
16 );
17 end component;
18 component ROM is port(
19     clk : in std_logic;
20     addr : in std_logic_vector(3 downto 0);
21     data : out std_logic_vector(3 downto 0)
22 );
23 end component;

```

```

24 component combinatorius is port (
25     input    : in std_logic_vector(3 downto 0);
26     output   : out std_logic_vector(2 downto 0)
27 );
28 end component;
29 component ButtonDebouncer is port (
30     RST      : in STD_LOGIC;
31     CLK      : in STD_LOGIC;
32     BTN      : in STD_LOGIC;
33     CLEARED_BTN : out STD_LOGIC
34 );
35 end component;
36 component memoria is port (
37     clock    : in std_logic;
38     reset    : in std_logic;
39     input     : in std_logic_vector(2 downto 0);
40     write_addr : in std_logic_vector(3 downto 0);
41     read_addr  : in std_logic_vector(3 downto 0);
42     write_en   : in std_logic;
43     output    : out std_logic_vector(2 downto 0)
44 );
45 end component;
46
47 signal m_input : std_logic_vector(3 downto 0):= (others => '0');
48 signal m_output : std_logic_vector(2 downto 0):= (others => '0');
49 signal address : std_logic_vector(3 downto 0);
50
51 begin
52 counter : counter_mod16 port map (
53     clock    => clock,
54     reset    => reset,
55     enable   => read,
56     counter  => address
57 );
58
59 rom_data : ROM port map (
60     clk      => clock,
61     addr     => address,
62     data     => m_input
63 );
64
65 macchina : combinatorius port map (
66     input    => m_input,
67     output   => m_output
68 );
69
70 ram : memoria port map (
71     clock    => clock,
72     reset    => reset,
73     input     => m_output,
74     write_addr => address,
75     read_addr  => (others => '0'),
76     write_en   => read
77 );
78 output <= m_output;
79 end structural;

```

6.1.3 Simulazione

La simulazione del sistema è stato fatto alzando il segnale di read 3 volte, poi la macchina viene resettata, per poi alzare di nuovo read.

In figura 50 è presentato il risultato della simulazione. Come vediamo il funzionamento è quello che ci si aspetta. In particolare è da notare che la ROM presenta in output il nuovo valore un clock dopo rispetto quello che si aspetta, questo perché il valore del contatore campionato a 135 ns è ancora 0, quindi il valore della ROM cambierà al rising edge successivo. Per la simulazione è stato posizionato nella ROM, come primo elemento il valore esadecimale "7" (invece dello "0" visto nel codice), in modo da rendere più evidente il funzionamento del sistema. La memoria funziona correttamente, infatti vengono memorizzati i primi 3 valori in uscita dalla macchina combinatoria, che sono, per come la abbiamo definita, proprio i valori in uscita della ROM, quindi 7, 1 e 2.

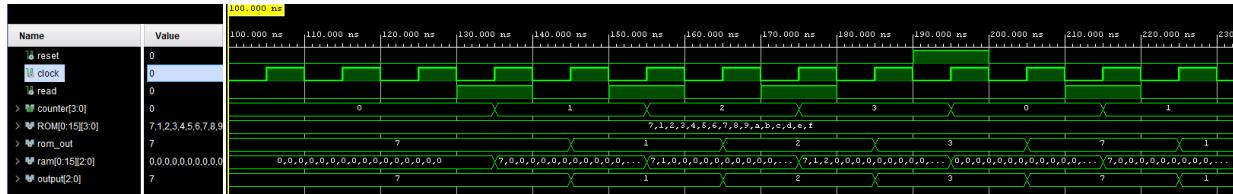


Figura 50: Simulazione del sistema dell'esercizio 6

6.2 Implementazione su board

Traccia

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.

6.2.1 Schematici

Per implementare il sistema di testing su board è stato solamente necessario utilizzare due debouncer per pulire i segnali di read e reset provenienti dai bottoni e collegare gli output del sistema di testing ai led della board. Il debouncer è stato già introdotto nel paragrafo 3.2.1. Di seguito lo schematico del componente che verrà poi implementato su board:

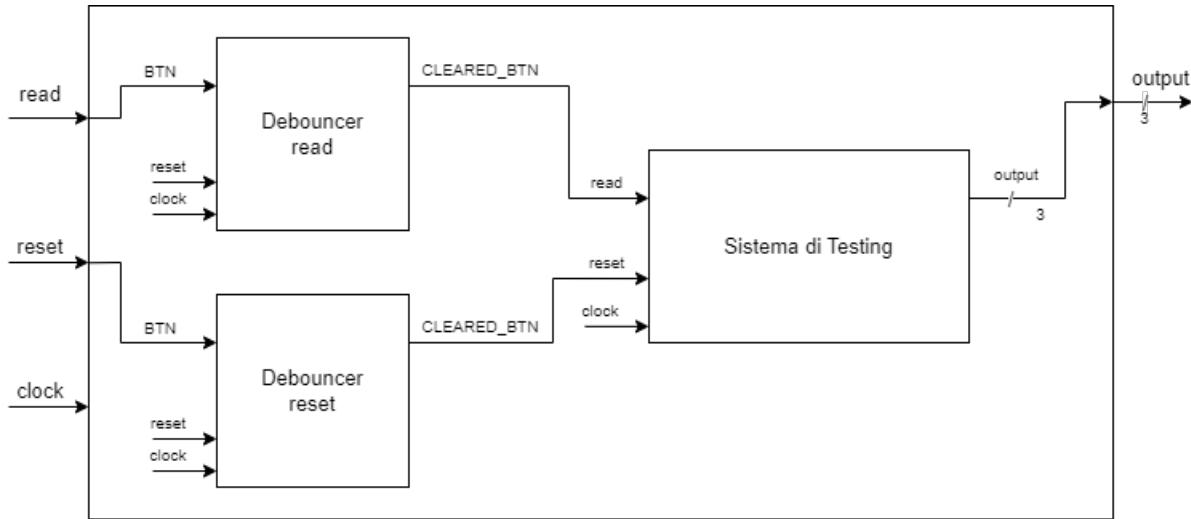


Figura 51: Sistema di testing su board

6.2.2 Codice VHDL

Il sistema su board è stato implementato su board tramite composizione di elementi. Sono stati utilizzati due segnali interni per collegare le uscite dei debouncer al sistema di testing. Per collegare i due bottoni e i led al sistema sviluppato sono stati usati dei constraint. Di seguito il codice dell'implementazione su board:

```

1  entity sistema_testing_on_board is port (
2      clock      : in std_logic;
3      read       : in std_logic;
4      reset      : in std_logic;
5      output     : out std_logic_vector(2 downto 0)
6  );
7  end sistema_testing_on_board;
8
9  architecture structural of sistema_testing_on_board is
10 component ButtonDebouncer is port (
11     RST         : in STD_LOGIC;
12     CLK         : in STD_LOGIC;
13     BTN         : in STD_LOGIC;
14     CLEARED_BTN : out STD_LOGIC
15 );
16 end component;
17 component insieme_cose is port (
18     clock      : in std_logic;
19     read       : in std_logic;
20     reset      : in std_logic;
21     output     : out std_logic_vector(2 downto 0)
22 );
23 end component;
24
25 signal read_clear : std_logic := '0';
26 signal reset_clear : std_logic := '0';

```

```
27
28 begin
29 sistema : insieme_cose port map (
30     clock    => clock,
31     read     => read_clear,
32     reset    => reset_clear,
33     output   => output
34 );
35
36 deb_read : ButtonDebouncer port map (
37     RST      => '0',
38     CLK      => clock,
39     BTN      => read,
40     CLEARED_BTN => read_clear
41 );
42
43 deb_reset : ButtonDebouncer port map (
44     RST      => '0',
45     CLK      => clock,
46     BTN      => reset,
47     CLEARED_BTN => reset_clear
48 );
49 end structural;
```

7 Comunicazione con Handshaking

L'handshake è un protocollo di comunicazione asincrono unidirezionale che gestisce la comunicazione tra 2 entità A (mittente) e B (destinatario). Esistono 2 implementazioni diverse del protocollo:

- Handshaking parziale; (fig: 52)
- Handshaking completo. (fig: 53)

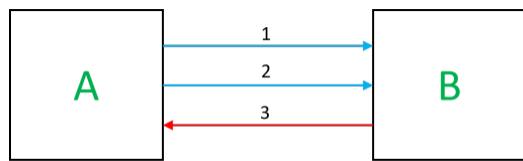


Figura 52: esempio handshake parziale

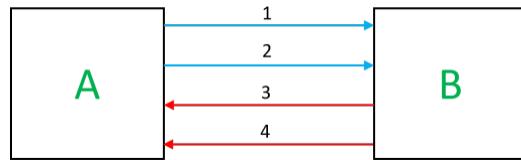


Figura 53: esempio handshake completo

Nel caso di Handshaking parziale si ha un protocollo di comunicazione del tipo:

1. A inserisce il dato da inviare sulla linea d'informazione;
2. A pone alza la linea che avvisa della presenza del dato a B;
3. Se disponibile B alza la linea che avvisa A della ricezione;
4. A abbassa la linea di avviso della comunicazione a B;
5. A toglie il dato sulla linea d'informazione.

Nel caso di handshaking completo si ha un protocollo di comunicazione del tipo:

1. A inserisce il dato da inviare sulla linea d'informazione;

2. A pone alza la linea che avvisa della comunicazione a B;
3. Se disponibile B pone alza la linea che avvisa A della ricezione;
4. B elabora il dato ed alza la linea che avvisa la fine dell'operazione ad A;
5. A abbassa la linea di avviso della comunicazione a B;
6. A toglie il dato sulla linea d'informazione.

La differenza sostanziale tra le 2 tipologie di handshaking consiste nel fatto che nel caso di handshaking parziale il trasmittente (prima di chiudere la comunicazione) non sa se il ricevitore ha finito le operazioni da svolgere col dato, mentre in caso di handshaking completo è stesso il ricevitore ad avvisare al trasmittente di poter rimuovere il dato e di chiudere la comunicazione.

Inoltre il protocollo di handshake può essere migliorato, infatti nel caso in cui il ricevitore non è mai disponibile alla comunicazione, capita che il trasmettitore rimane bloccato in uno stato di attesa di "ack". È possibile ovviare al problema attraverso una piccola modifica al comportamento del mittente, infatti il mittente può essere liberato dalla comunicazione (e definirla fallita) in caso in cui si sia ricevuto dell'"ack" entro un determinato tempo massimo di attesa.

7.1 Progettazione in VHDL

Traccia

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 ($i = 0, \dots, N - 1$). 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.

7.1.1 Schematici

In questo esercizio viene sviluppata la comunicazione tra 2 nodi, A (mittente) e B (ricevitore), che comunicano attraverso un protocollo di handshaking parziale (senza l'aggiunta di un controllo sul fallimento della comunicazione da parte di A). I dati da mandare da parte del mittente sono presenti all'interno di una memoria ROM, mentre il ricevitore ogni volta che riceve un dato, lo addiziona con il corrispettivo valore presente nella ROM e salva il risultato all'interno di una memoria RAM. Il mittente avvia la comunicazione solo dopo aver ricevuto un segnale di start ed arriva in uno stato di idle nel momento in cui tutti i dati sono stati inviati sulla linea di comunicazione con B. Il nodo del ricevitore è pronto alla comunicazione tramite handshake con il nodo trasmittente, una volta ricevuto il dato dal nodo A, si esegue la somma del dato con il valore del corrispettivo presente sulla ROM (tramite un ripple carry adder) e successivamente il risultato viene salvato nella memoria RAM. Alla fine del processo di invio dei dati, si avranno nella ram del nodo B tutte le somme che erano state richieste nella traccia. Di seguito sono rappresentati tutti i componenti dei 2 nodi.

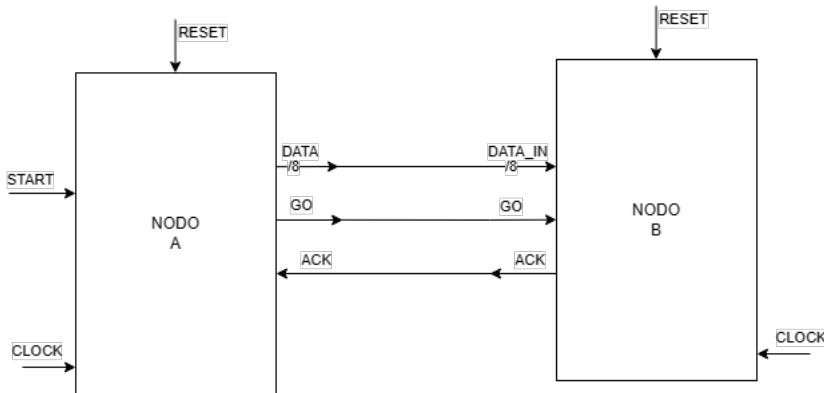


Figura 54: Collegamento tra nodo A e nodo B

Nodo A

Nel nodo A sono presenti:

- una ROM, al cui interno sono presenti tutti i valori da inviare al nodo B;
- un contatore modulo 8, utilizzato per poter scegliere quale valore deve essere preso dalla ROM ed inviato al nodo B;
- una control unit, per gestire l'handshake e l'incremento del contatore.

ROM La ROM utilizzata é la stessa di quella dell'esercizio 6 (peró con 3 bit d'indirizzo e 8 bit d'uscita), é possibile trovarne la descrizione al seguente paragrafo 6.1.1.

counter_8 Il contatore modulo 8 é lo stesso di quello utilizzato nell'esercizio 5, con la differenza che conta su 3 bit. È possibile trovare una descrizione del componente al seguente paragrafo 5.1.1.

uc_A La control unit del nodo A, viene utilizzata per poter gestire l'invio dei dati tramite handshake. Il componente é stato sviluppato tramite una macchina a stati finiti di Moore a 3 stati:

- idle, che é lo stato in cui la macchina attende l'inizio della trasmissione;
- inserisci_dato, che é lo stato in cui la macchina inserisce il dato da inviare sulla linea di trasmissione;
- invio, che é lo stato in cui la macchina avvia la comunicazione tramite handshake;
- ricevuto, che é lo stato in cui la macchina incrementa il valore del contatore e controlla se la comunicazione é finita o meno, nel caso il valore del contatore é pari a "111" si considera finita la comunicazione e si ritorna in idle, in caso contrario si ritorna nello stato di invio.

Di seguito é presente la figura dell'automa della control unit 55

Nodo B

Nel nodo B sono presenti:

- una ROM, al cui interno sono presenti tutti i valori $Y(i)$ da sommare ai messaggi dal nodo A;
- una RAM, al cui interno vengono salvate le somme dei valori inviati dal nodo A;
- un ripple carry adder, per poter eseguire la somma: $S(i) = X(i) + Y(i)$;
- un contatore modulo 8, utilizzato per poter scegliere da quale indirizzo della ROM prendere il dato ed in quale locazione di memoria della Ram una somma deve essere salvata;
- una control unit, per gestire l'handshake, l'incremento del contatore e la memorizzazione delle somme.

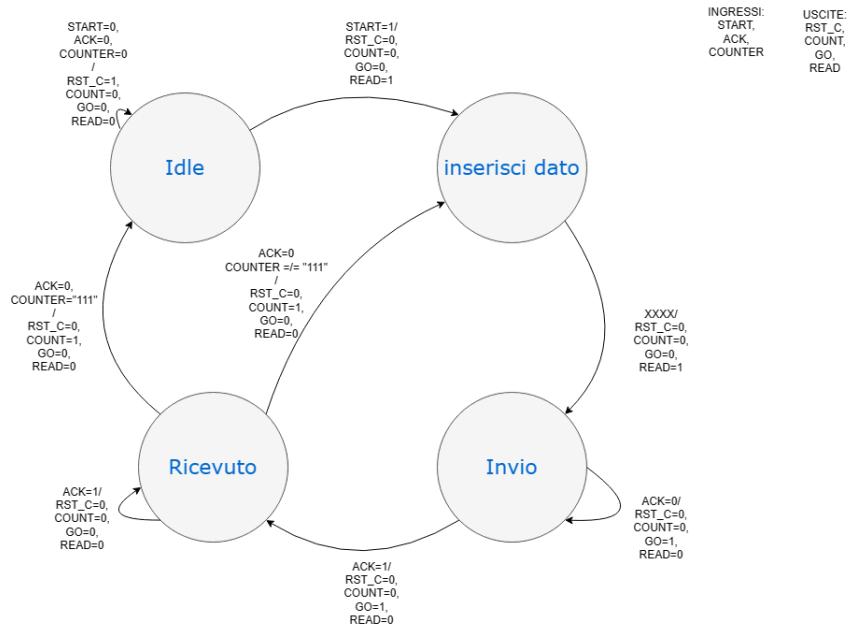


Figura 55: automa nodo A handshake

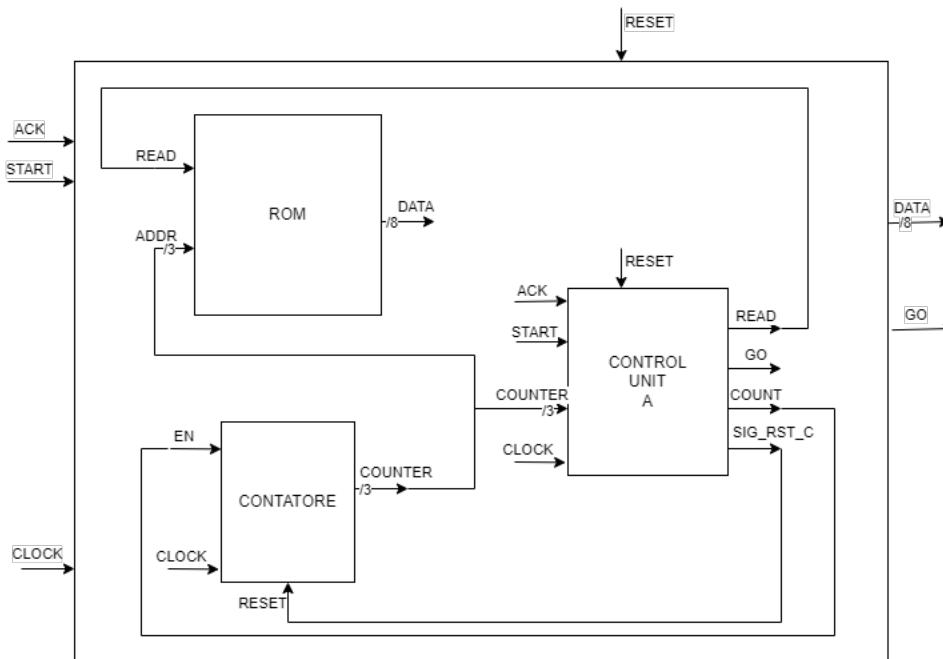


Figura 56: collegamenti nodo A

ROM La ROM utilizzata é la stessa di quella dell'esercizio 6 (però con 3 bit d'indirizzo e 8 bit d'uscita), é possibile trovarne la descrizione al seguente paragrafo: 6.1.1. La Rom del nodo B contiene tutti i valori Y(i) che vanno sommati agli ingressi del nodo A.

memoria La memoria utilizzata é la stessa di quella dell'esercizio 6 (peró con 3 bit d'indirizzo e 9 bit d'uscita, si ha un bit in piú per non perdere il carry della somma), é possibile trovarne la descrizione al seguente paragrafo: 6.1.1. Nella memoria vengono salvati tutti le somme $S(i) = X(i) + Y(i)$ durante la trasmissione dei dati.

counter_8 Il contatore modulo 8 é lo stesso di quello utilizzato nell'esercizio 5, con la differenza che conta su 3 bit. É possibile trovare una descrizione del componente al seguente paragrafo: 5.1.1.

adder_sub Nonostante l'esercizio sia precedente all'esercizio 11, si ha una trattazione piú approfondita del componente al paragrafo: 11.1.2.

full_adder Nonostante l'esercizio sia precedente all'esercizio 11, si ha una trattazione piú approfondita del componente al paragrafo: 11.1.2.

uc_B La control unit del nodo B, viene utilizzata per poter gestire la ricezione dei dati tramite handshake e la memorizzazione delle somme. Il componente é stato sviluppato tramite una macchina a stati finiti di Moore a 3 stati:

- idle, che é lo stato in cui arriva la macchina alla fine della comunicazione;
- waiting, che é lo stato in cui la macchina attende i dati dal nodo A;
- send_ack, che é lo stato in cui la macchina incrementa il valore del contatore, alza il bit di write enable, manda l'ack e controlla se la comunicazione é finita o meno, nel caso il valore del contatore é pari a "111" si considera finita la comunicazione e si va in idle, in caso contrario si ritorna nello stato di invio.

Di seguito é presente la figura dell'automa della control unit: 57

7.1.2 Codice VHDL

uc_A La control_unit del nodo A è stata sviluppata in modo comportamentale attraverso il costrutto process e la descrizione dell'automa rappresentato nel paragrafo degli schematici. Di seguito è presente il codice VHDL del componente.

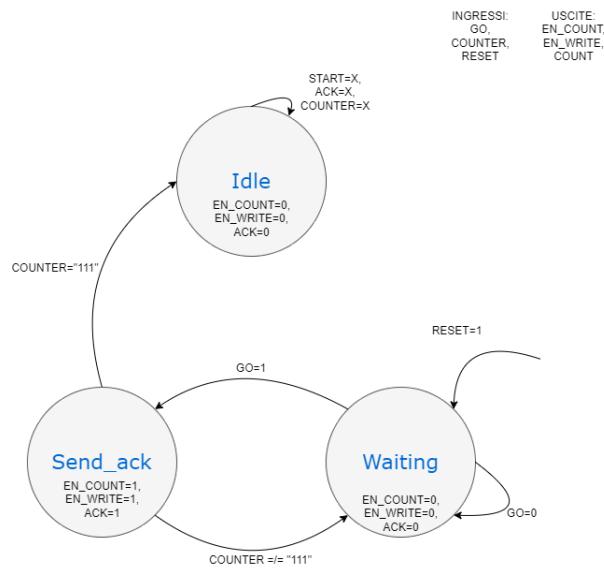


Figura 57: automa nodo B handshake

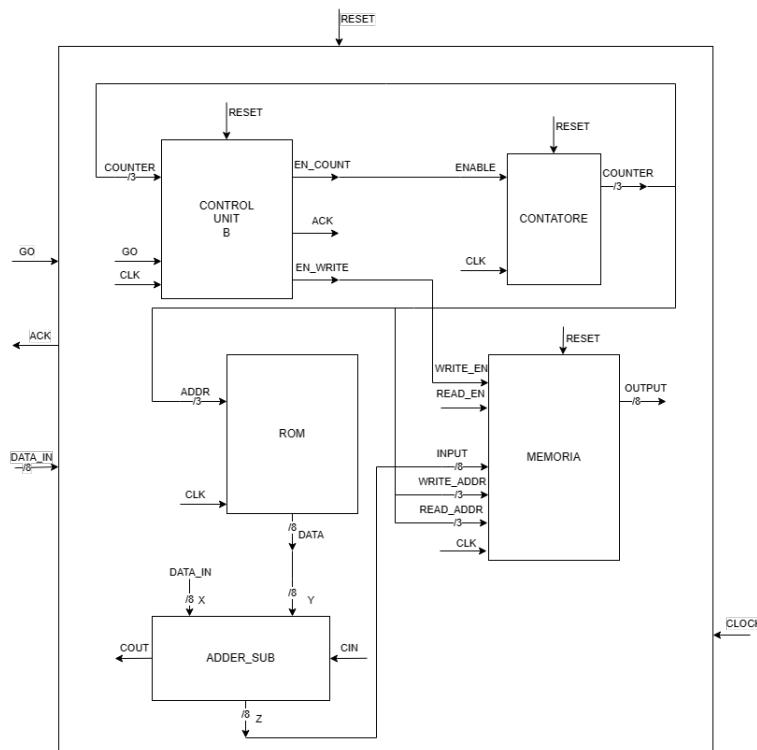


Figura 58: collegamenti nodo B

```

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

```

```

4 entity uc_A is port (
5     clock      : in std_logic;
6     reset      : in std_logic;
7     start      : in std_logic;
8     ack        : in std_logic;
9     counter    : in std_logic_vector(2 downto 0);
10    rst_c      : out std_logic;
11    count      : out std_logic;
12    go         : out std_logic;
13    read       : out std_logic
14 );
15 end uc_A;
16
17 architecture Behavioral of uc_A is
18
19 type stato is (idle, inserisci_dato, invio, ricevuto);
20 signal curr_state : stato := idle;
21 signal next_state : stato := idle;
22
23 begin
24
25 registri : process (clock)
26 begin
27 if rising_edge(clock) then
28
29     if reset='1' then
30         curr_state <= idle;
31     else
32         curr_state  <= next_state;
33     end if;
34 end if;
35 end process;
36
37 calcolo_stato : process(curr_state, start, ack)
38 begin
39
40 rst_c <= '0';
41 read <= '0';
42 go <= '0';
43 count <= '0';
44
45 case curr_state is
46     when idle      =>
47         if (start = '1') then
48             rst_c <= '1';
49             next_state <= inserisci_dato;
50         else
51             next_state <= idle;
52         end if;
53     when inserisci_dato =>
54         read <= '1';
55         next_state<= invio;
56     when invio      =>
57         go <= '1';
58         if (ack = '1') then
59             next_state <= ricevuto;
60         else
61             next_state <= invio;
62         end if;
63     when ricevuto   =>

```

```

64      if (ack = '0') then
65          count <= '1';
66          if (counter = "111") then
67              next_state <= idle;
68          else
69              next_state <= inserisci_dato;
70          end if;
71      else
72          next_state <= ricevuto;
73      end if;
74
75
76 end case;
77 end process;
78
79 end Behavioral;

```

uc_B Come per la control_unit del nodo A, anche la control unit del nodo B è stata sviluppata in modo comportamentale attraverso il costrutto process e la descrizione dell'automa rappresentato nel paragrafo degli schematici. Di seguito è presente il codice VHDL del componente.

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 entity uc_B is port (
5     clk           : in    std_logic;
6     reset         : in    std_logic;
7     go            : in    std_logic;
8     counter       : in    std_logic_vector(2 downto 0);
9     en_count      : out   std_logic;
10    en_write      : out   std_logic;
11    ack           : out   std_logic
12 );
13 end uc_B;
14
15 architecture Behavioral of uc_B is
16
17 type stato is (idle, waiting, send_ack);
18 signal curr_state : stato := waiting;
19 signal next_state : stato := waiting;
20
21 begin
22
23 registri      : process (clk)
24 begin
25 if rising_edge(clk) then
26
27     if reset='1' then
28         curr_state <= waiting;
29     else
30         curr_state  <= next_state;
31     end if;
32 end if;

```

```

33 end process;
34
35
36 calcolo_stato : process(curr_state, go)
37 begin
38
39 en_count    <= '0';
40 en_write    <= '0';
41 ack         <= '0';
42
43 case curr_state is
44
45   when idle           =>
46     next_state <= idle;
47
48   when waiting        =>
49     if go = '1' then
50       next_state <= send_ack;
51     else
52       next_state <= waiting;
53     end if;
54
55   when send_ack       =>
56     en_write    <= '1';
57     en_count    <= '1';
58     ack <= '1';
59     if (counter = "111") then
60       next_state <= idle;
61     else
62       next_state <= waiting;
63     end if;
64
65
66 end case;
67 end process;
68
69 end Behavioral;

```

7.1.3 Simulazione

Essendo un protocollo asincrono abbiamo testato l'invio dei messaggi da un componente all'altro con l'utilizzo di 2 clock diversi, in particolare si ha:

- clock A = 10ns;
- clock B = 40ns.

All'inizio della simulazione si ha un segnale alto di reset per 100ns, per avere la corretta inizializzazione della macchina e subito dopo viene alzato il segnale di start per 10ns. Come è possibile osservare dal testbench in figura (59) il nodo A riesce ad inviare correttamente tutti i dati al nodo B.

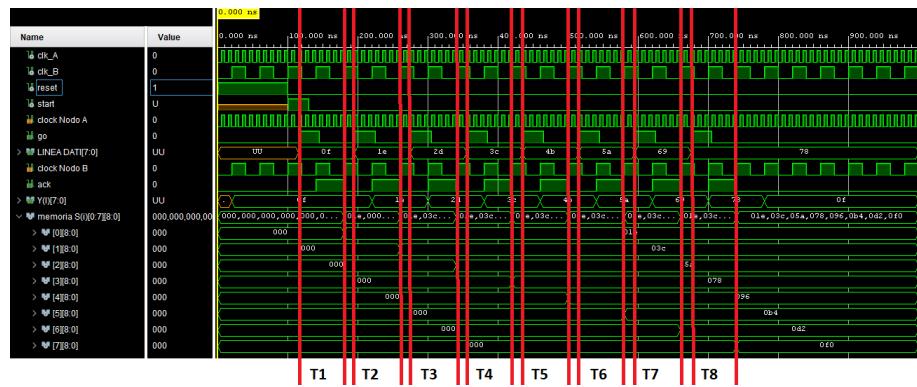


Figura 59: testbench handshake

8 Processore MIC-1

Il MIC-1 è un'architettura di processore inventata da Andrew S. Tanenbaum come esempio all'interno del suo libro Structured Computer Organization [1]. Questa architettura è molto semplice per essere un processore, ma presenta tutte le tematiche e difficoltà che si affrontano nella progettazione di sistemi complessi.

Il processore MIC-1 utilizza un modello di processore a stack, quindi preleva gli operandi, esclusivamente, da uno stack. Non utilizzando il modello a registri generali le istruzioni utilizzate dal processore sono più semplici, perché prevedono solo un tipo di indirizzamento degli operandi (indirizzamento implicito), avendo però come svantaggio il fatto di dover fare continui accessi in memoria per poter recuperare gli operandi.

Questo processore è realizzato tramite decomposizione in parte operativa e parte di controllo, vediamole brevemente prima di affrontare l'esercizio.

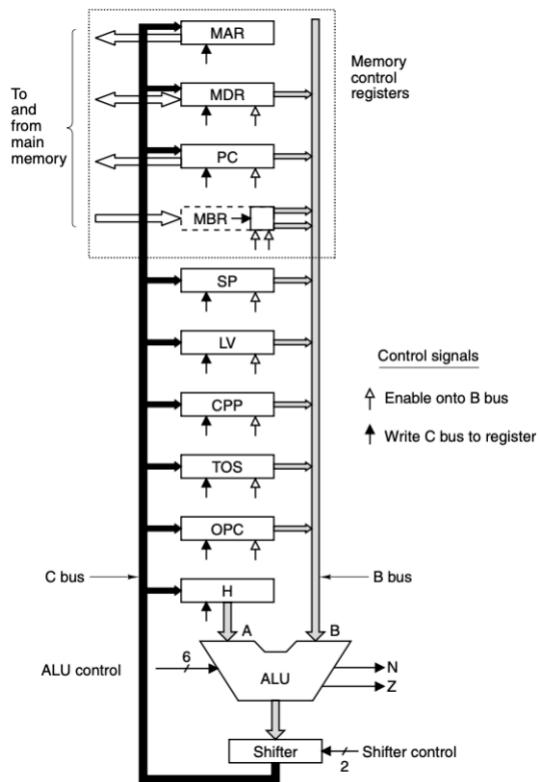


Figura 60: Parte operativa del processore MIC-1

Parte operativa

In figura 60 è presentata la parte operativa del processore MIC-1. I componenti essenziali di questa architettura sono:

- **Registri:** hanno dimensione di 32 bit e assolvono a diversi compiti, alcuni sono di interfacciamento con la memoria (MAR, MDR, PC, MBR), alcuni mantengono gli indirizzi base di alcune zone di memoria (LV, CPP), altri servono a gestire la stack (SP, TOS), un registro tampone (H) e un registro che mantiene valori temporanei (OPC, mantiene di solito il vecchio PC, ma si può usare anche per i calcoli)
 - **ALU:** possiede due ingressi, uno è collegato al registro H e l'altro ad un bus, mentre in output vi è una linea da 32 bit che posiziona il risultato in uno shift register e 2 flag di stato (N, Z). Per controllare l'ALU vi sono 6 bit in ingresso per la selezione delle operazioni
 - **Bus:** vi sono 2 bus (B, C) che collegano i vari registri del processore. Un bus è collegato all'output dei registri e all'input della ALU, mentre l'altro è collegato all'output dello shifter e all'input dei registri

Parte di controllo

Per poter comandare completamente la parte operativa sono necessari 36 bit, che quindi è necessario facciano parte di ogni istruzione, più correttamente micro-istruzioni, del processore. In figura 61 è presentata una microistruzione del processore MIC-1, che presenta i seguenti campi:

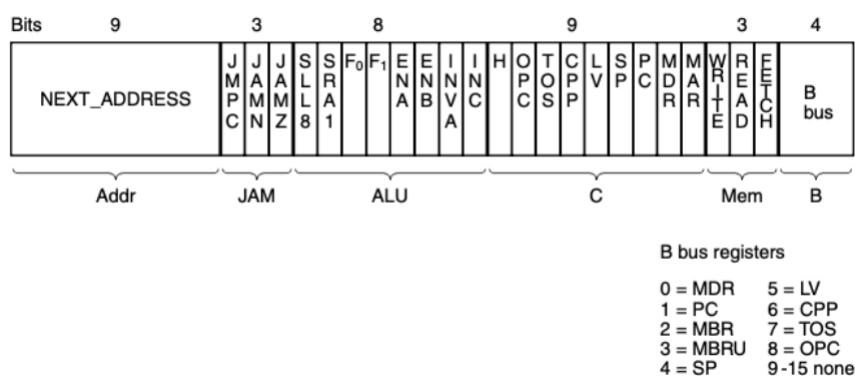


Figura 61: Istruzione del processore MIC-1

- Addr: contiene l'indirizzo della prossima microistruzione
 - JAM: permette di gestire i salti nel ciclo delle istruzioni (i salti possono dipendere anche dai flag N e Z dell'ALU)
 - ALU: permette di controllare l'ALU
 - C: permette di selezionare quali registri vengono scritti dal bus C (tanti bit alti quanti sono i registri su cui si scrive)
 - Mem: permette di controllare le operazioni da/verso la memoria
 - B: permette di selezionare quale registro viene letto dal bus B (dato che un solo registro può essere letto nello stesso momento questo valore viene codificato)

Il set di microistruzioni implementato dal MIC-1 è un sottoinsieme di quello della Java Virtual Machine, denominato IJVM in quanto opera unicamente sugli interi. Più microistruzioni formano una sequenza di controllo.

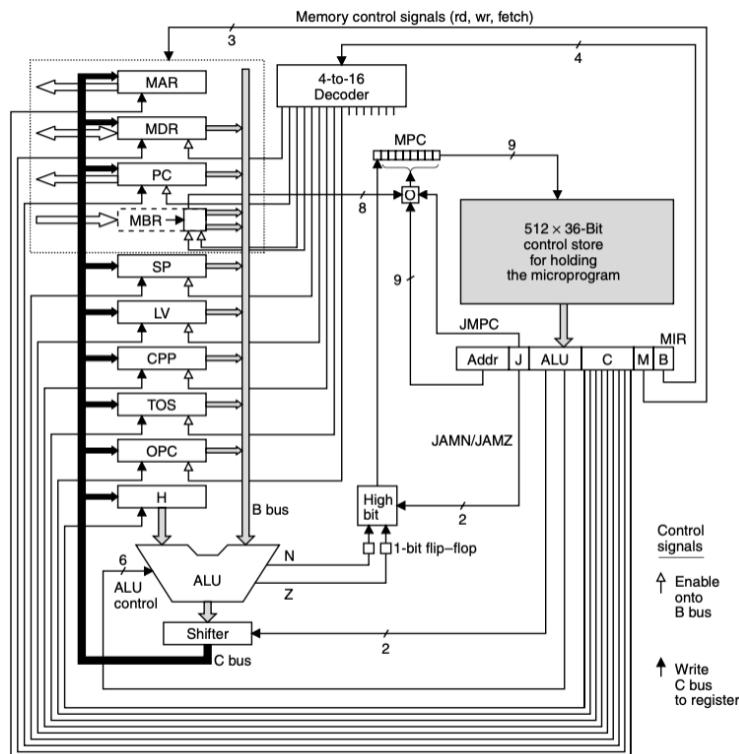


Figura 62: Parte di controllo del processore MIC-1

La parte di controllo è realizzata seguendo il modello micropogrammato: le sequenze di controllo relative ai codici operativi del processore sono memorizzate all'interno del control store, che è una ROM 512x36 bit. In figura 62 vi è l'intera architettura del processore MIC-1. Oltre i componenti già illustrati vi sono degli altri componenti appartenenti alla parte di controllo, un decoder che decodifica i bit B della microistruzione e delle componenti necessarie per gestire i salti nelle sequenze di controllo.

Per programmare più facilmente il processore viene utilizzato un linguaggio denominato MAL (MicroAssembly Language) e un microassemblatore si occupa poi di eseguire la traduzione da linguaggio al formato della microistruzione presentata poco sopra.

8.1 Studio di due istruzioni del processore

Traccia

A partire dall'implementazione fornita di un 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.

8.1.1 Codici operativi analizzati

Abbiamo deciso di analizzare due codici operativi: BIPUSH e IOR.

BIPUSH

Il codice operativo BIPUSH permette di caricare sullo stack un byte specificato alla chiamata del codice. E' un'istruzione molto importante in quanto, dato che il processore sfrutta il modello a stack, è essenziale poter caricare dei byte sullo stack per l'esecuzione di molte istruzioni. Un esempio di istruzione chiamata in un microprogramma è presente di seguito:

BIPUSH 0xA

L'istruzione ha una lunghezza di due byte: un byte è dedicato a mantenere il valore da caricare in stack, mentre l'altro byte è quello utilizzato per specificare l'indirizzo, riferito al control store, del codice operativo BIPUSH. Quest'ultimo è detto anche entry point, perchè è il punto in cui il microprogramma, che utilizza il codice operativo ,entra nel control store per eseguire le microistruzioni della BIPUSH.

Vediamo ora le microistruzioni, in linguaggio MAL, che descrivono il funzionamento del BIPUSH:

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

Vediamo prima di tutto che l'indirizzo di questa istruzione nel contro store è 0x10, quindi ci aspettiamo di ritrovarlo anche nel microprogramma come entry point della BIPUSH quando controlleremo la RAM.

Ricordiamo che prima che venga eseguita la BIPUSH si è già eseguito il microprogramma "main" che si occupa di fare il fetch della prossima istruzione, quindi, al momento dell'esecuzione della prima microistruzione della BIPUSH il valore da caricare sullo stack sarà già memorizzato all'interno di Memory Byte Register (MBR). La prima istruzione del codice operativo è l'incremento del registro Stack Pointer (SP) (perchè la stack si espande verso il basso in MIC-1) e l'assegnazione di questo valore sia al Memory Address Register che allo Stack Pointer stesso, in questo modo ci predisponiamo alla scrittura del byte in memoria e manteniamo coerente con la realtà il valore di SP nel processore. Viene poi incrementato il Program Counter (PC) e fatto il fetch dell'istruzione in modo che quando ritorneremo a "main" sarà già caricata nel Memory Data Register (MDR) la prossima istruzione. Quindi in questo momento in MBR è salvato il byte da caricare in memoria e in MAR vi è l'indirizzo in cui caricare il byte (ottenuto incrementando SP). Quindi viene spostato il byte nel registro Top of Stack (TOS) per mantenerlo coerente con lo stato attuale dello stack e viene anche inserito anche nel registo MDR. Quindi viene richiamata l'operazione di write (wr) per scrivere il contenuto di MDR all'indirizzo specificato in MAR, in modo da completare l'esecuzione dell'istruzione. Infine vi è un salto al main per consentire il fetch della prossima istruzione e quindi proseguire con l'esecuzione del microprogramma caricato nella RAM.

```
-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000001000000000000000100000000",
1 => "0000111000010000000101000010000000",
2 => "1010011100000010011011001100101",
3 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
);
```

Figura 63: Ram di un microprogramma che utilizza la bipush

Analizziamo ora la RAM che contiene un microprogramma che richiama una BIPUSH per il valore "0xA". In figura 63 sono evidenziati due byte del microprogramma, il byte

evidenziato in rosso è il valore "0xA" in binario che dovrà essere caricato in memoria, mentre nel byte evidenziato in azzurro c'è, in binario, l'entry point della BIPUSH, cioè "0x10". Notiamo inoltre che la RAM viene riempita da sinistra verso destra.

Ora analizziamo la simulazione, visualizzata utilizzando il tool GTKwave, del processore che esegue il microprogramma che contiene la microistruzione in esame. Come vediamo nella figura 64 la simulazione segue perfettamente quanto illustrato poco sopra in questo testo quando è stato descritto il funzionamento della BIPUSH usando il codice MAL. Notiamo inoltre che l'intero codice operativo è eseguito da 245 ns a 265 ns, cioè vengono utilizzati 3 colpi di clock e ad ogni colpo di clock viene eseguita una microistruzione delle tre viste sopra.

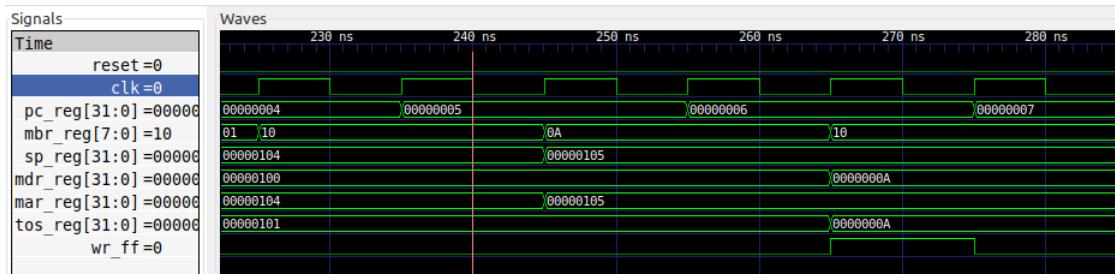


Figura 64: Simulazione di un microprogramma che utilizza la bipush

IOR

Il codice operativo IOR permette di svolgere l'operazione logica di OR tra i primi due valori che si trovano in cima allo stack, eliminandoli dallo stack e posizionando il risultato in cima allo stack. Un esempio di istruzione chiamata in un microprogramma è presente di seguito:

IOR

L'istruzione assume senso nel momento in cui nella stack sono stati caricati almeno due valori.

L'istruzione è lunga solo un byte che è quello utilizzato per specificare l'entry point nel control store per il codice operativo IOR.

Vediamo ora le microistruzioni, in linguaggio MAL, che descrivono il funzionamento del IOR:

```
ior = 0xB6:
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR OR H; wr; goto main
```

Vediamo prima di tutto che l'indirizzo di questa istruzione nel control store è 0xB6, quindi ci aspettiamo di ritrovarlo anche nel microprogramma come entry point del IOR quando controlleremo la RAM. Inizialmente lo Stack Pointer (SP) punta al primo elemento dello stack, il cui valore sarà anche posizionato nel registro Top of Stack (TOS), quindi per recuperare gli operandi dell'operazione logica è necessario fare un solo accesso in memoria per recuperare il secondo elemento nello stack. Proprio per questo la prima microistruzione decremente lo Stack Pointer (SP) (la stack si espande verso il basso in MIC-1) in modo che SP punti al secondo operando della OR, questo valore viene anche assegnato al Memory Address Register (MAR) e viene richiamata l'operazione di read (rd) per leggere il secondo operando e averlo disponibile sul Memory Data Register (MDR). Successivamente si posiziona il primo operando, il cui valore è memorizzato in TOS, nel registro tampone dell'ALU (H). Nell'ultima microistruzione viene svolta effettivamente l'operazione di OR dall'ALU, tra il registro H e il registro MDR. Il risultato è salvato in TOS, in quanto sarà il prossimo valore in cima allo stack (si mantiene la semantica di TOS), ed anche in MDR, in quanto con l'operazione di write (wr) posizionerà il risultato dell'operazione (che è in MDR) nello stack all'indirizzo specificato in MAR, che è lo stesso che è in SP. In questo modo l'operazione di scrittura del risultato avviene in cima allo stack, mantenendo quindi la semantica del registro SP, che punterà sempre al primo elemento nella stack. I due operandi vengono entrambi rimossi dallo stack, come anticipato poco prima. Alla fine vi è un salto al main per consentire il fetch della prossima istruzione e quindi proseguire con l'esecuzione del microprogramma caricato nella RAM.

```
-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "0000000000000000000000000000000000000000000000000000000000000000",
0 => "00000001000000000000000010000000",
1 => "00001110001000000010100010000",
2 => "00000000000000001010011110110110",
others => (others => '0')
--END_WORDS_ENTRY
);
```

Figura 65: Ram di un microprogramma che utilizza la ior

Analizziamo ora la RAM che contiene un microprogramma che richiama due BIPUSH e successivamente la IOR. In figura 65 c'è la RAM dopo aver scritto il microprogramma. La riga 1 contiene due istruzioni di BIPUSH dei valori "0xA" e "0xE" e sulla riga 3, evidenziato dal box verde è presente, in binario, il richiamo al codice operativo della IOR ("0xB6"), cioè il suo indirizzo nel control store. IL byte prima della IOR è l'indirizzo nel control store del codice operativo ijvm_goto in binario (cioè "0xA7") che serve a salvare il valore precedente

del PC nel registro OPC (Old Program Counter) che può essere utile per alcune elaborazione. Notiamo inoltre che la RAM viene riempita da destra verso sinistra.

Ora analizziamo la simulazione, visualizzata utilizzando il tool GTKwave, del processore che esegue il microprogramma che contiene la microistruzione in esame. Come vediamo nella figura 66 la simulazione segue perfettamente quanto illustrato poco sopra in questo testo quando è stato descritto il funzionamento della IOR usando il codice MAL. Il risultato dell'operazione che vediamo nel registro MDR e TOS dopo l'operazione è effettivamente la OR logica tra i due operandi, cioè "0xE". Notiamo inoltre che l'intero codice operativo è eseguito da 325 ns a 345 ns, cioè vengono utilizzati 3 colpi di clock e ad ogni colpo di clock viene eseguita una microistruzione delle tre viste sopra.

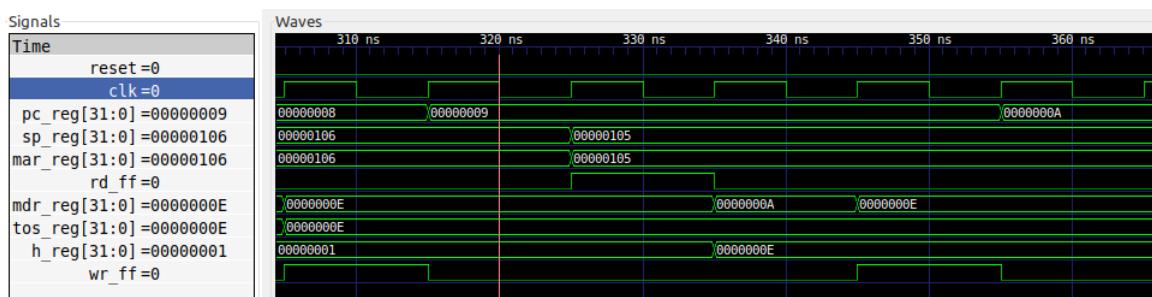


Figura 66: Simulazione di un microprogramma che utilizza la ior

8.2 Modifica di un codice operativo

Traccia

A partire dall'implementazione fornita di un processore operante secondo il modello IJVM, si modifichi un codice operativo a scelta, documentando tutte le modifiche effettuate.

8.2.1 Creazione di un codice operativo: IXOR

Invece di effettuare una modifica ad un codice operativo abbiamo deciso di creare uno completamente nuovo. In particolare abbiamo deciso di implementare una nuova operazione logica che non è possibile eseguire con l'ALU fornita dall'implementazione, cioè la XOR logica. Come per la IOR esaminata nella sezione scorsa abbiamo deciso di implementare la XOR in modo che rimuova gli operandi dallo stack e vi inserisca la XOR tra i due. I due operandi sono sempre i primi due elementi nello stack.

Per fare ciò abbiamo dovuto affrontare diverse step:

- Scrivere il codice MAL per il codice operativo della XOR
- Trovare un "buco" nel control store in cui inserire le microistruzioni relative alla XOR
- Scrivere nella RAM un microprogramma che includa la XOR e simulare la sua esecuzione

Vediamo i vari step nel dettaglio:

Scrittura della XOR

Prima di tutto ci occupiamo di scrivere il codice MAL del codice operativo che vogliamo creare. Nativamente l'ALU non prevede l'operazione di XOR, quindi è stato necessario trovare un modo per calcolare la XOR utilizzando le operazioni previste dall'ALU. In particolare la XOR può essere calcolata come:

$$A \text{ XOR } B = (\text{!}A \text{ AND } B) \text{ OR } (A \text{ AND } \text{!}B)$$

Quindi dovrebbe essere necessario utilizzare AND, OR e NOT. Vediamo però come possiamo modificare la relazione per poter successivamente implementarla più facilmente. Infatti, utilizzando la Legge di De Morgan possiamo sostituire il secondo operando della OR con la sua forma negata ed opportunamente modificata. Quindi otteniamo la seguente:

$$A \text{ XOR } B = (\text{!}A \text{ AND } B) \text{ OR } (\text{!}(\text{!}A \text{ OR } B)) \quad (1)$$

Utilizzando la formula 1 dovremmo svolgere il calcolo in quattro fasi:

1. Calcolare $(\text{!}A \text{ AND } B)$, recuperando gli operandi dalla memoria
2. Salvare momentaneamente (da qualche parte) il risultato
3. Calcolare $\text{!}(\text{!}A \text{ OR } B)$
4. Calcolare la OR tra i due risultato delle due precedenti operazioni e salvarlo in stack

Quindi bisogna trovare qualche registro in cui salvare i risultati intermedi. Di seguito il codice che abbiamo scritto per implementare la XOR:

```
ixor = 0x8D:  
    MAR = SP = SP - 1; rd
```

```
H = NOT TOS
OPC = MDR AND H
H = MDR OR H
H = NOT H
MDR = TOS = OPC OR H; wr; goto main
```

La prima istruzione, come per la OR, predisponde la lettura di un operando (B nel nostro caso) decrementando SP, mettendo il valore nel MAR e chiamando l'operazione di read. L'altro operando (A nel nostro caso), essendo il primo valore della stack, è memorizzato nel registro TOS, la seconda istruzione utilizza l'ALU per effettuare la NOT dell'operando e memorizza il risultato nel registro H. Con la terza istruzione viene svolta effettivamente la prima AND e viene quindi calcolato ($A \text{ AND } B$) che viene memorizzato, temporaneamente, nel registro OPC. A questo punto abbiamo in H il valore negato dell'operando A e in MDR il valore dell'operando B, in questo momento diventa utile la trasformazione all'equazione fatta per ottenere la relazione 1, infatti non sarà necessario fare ulteriore operazioni per negare gli operandi, ma li potremo usare direttamente. Quindi con la quarta istruzione possiamo direttamente calcolare la quantità vista al passo 3 e viene memorizzata nel registro H. Con la quinta istruzione neghiamo il valore appena calcolato e lo posizioniamo nuovamente in H. Infine, svolgendo l'ultima istruzione, calcoliamo la OR tra i due valori calcolati in precedenza che sono posizionati in OPC e in H. Il risultato lo posizioniamo in TOS in quanto sarà il nuovo valore in cima alla stack e lo posizioniamo anche in MDR per predisporre la scrittura in stack. Infatti viene eseguita l'operazione di write per scrivere il risultato della XOR (presente in MDR) in memoria all'indirizzo riportato in MAR che è lo stesso che è in SP, quindi manteniamo anche la semantica corretta di SP che punta al primo elemento dello stack. Alla fine del codice vi è un salto al main per consentire il fetch della prossima istruzione e quindi proseguire con l'esecuzione del microprogramma caricato nella RAM. Abbiamo utilizzato il registro OPC per salvare il risultato intermedio in quanto è un registro che non è strettamente necessario in questo codice operativo e la sua semantica viene automaticamente rigenerata quando viene chiamato il goto che si trova alla fine del codice della XOR, in quanto, con il goto, viene assegnato ad OPC il valore precedente del Program Counter (PC).

Inserimento del codice nel control store

Scritto il codice MAL per la XOR è stato necessario trovare una zona del control store non occupata dalle microistruzioni di altri codici operativi. Il control store è composto da 512

righe di 36 bit, la prima metà è composta dalle microistruzioni relative ai codici operativi implementati, mentre la secondo metà, quindi dalla riga 256 a 511 è una zona riservata alla microistruzioni per gestire i salti legati ai codici operativi presenti nella prima metà. Quindi possiamo inserire il nostro codice operativo solo nella prima metà del control store. Si possono notare diverse parti vuote, in particolare abbiamo deciso di inserire le microistruzioni della XOR a partire dalla riga 141, quindi "0x8D" sarà l'entry point della XOR nel control store. In figura 67 vediamo come l'inserimento della XOR modifica il control store proprio dalla riga 141 in poi.

(a) Prima

(b) Dopo

Figura 67: Control store prima e dopo l'inserimento della XOR

L'inserimento in control store è stato effettuato sfruttando un comando di build fornito per rigenerare il control store una volta scritto il codice MAL del codice operativo.

Scrittura del microprogramma ed inserimento in RAM

Abbiamo quindi scritto un semplice microprogramma per testare il nuovo codice operativo creato. Di seguito il codice del microprogramma:

```
BIPUSH 0xA  
BIPUSH 0xE  
IXOR
```

Vengono inseriti due valori nella stack, il valore "0xA" e "0xE", e successivamente viene effettuata la XOR tra i due valori inseriti. Quindi il risultato della operazione, che ritro-

veremo in MDR, sarà "0x4".

Scritto il programma bisogna inserirlo nella RAM, per fare questo è possibile utilizzare un altro comando di build forntino per generare automaticamente la RAM partendo dal codice del microprogramma. Nel nostro caso il comando di build riscontrava diversi errori sulla sintassi, quindi abbiamo inserito manualmente in RAM le stringhe binarie derivanti dal microprogramma sopra riportato. Di seguito la traduzione in binario delle microistruzioni del microprogramma:

```
BIPUSH 0xA => 0000101000010000
BIPUSH 0xE => 0000111000010000
IXOR      => 10001101
```

Tradotte le microistruzioni in binario, abbiamo inserito le stringhe binarie nel file VHD della RAM. In figura 68 la Ram dopo la modifica, in rosso le stringhe che codificano l'entry point della BIPUSH (0x10), in azzurro la stringa che codifica l'entry point della IXOR (0x8D), in verde i valori che vengono memorizzati nello stack dalla BIPUSH (0xA e 0xE) e in nero la stringa che codifica l'entry point per il goto che fa tornare il processore al main.

```
-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000001000000000000000010000000",
1 => "00001110000100000001010000100000",
2 => "00000000000000001010011110001101",
others => (others => '0')
--END_WORDS_ENTRY
);
```

Figura 68: RAM dopo l'inserimento del microprogramma che usa la XOR

Simulazione del microprogramma che usa la XOR

A questo punto è stato simulato il microprogramma e la simulazione è stata visualizzata utilizzando il tool GTKwave. Come vediamo nella figura 69 la simulazione segue perfettamente quanto illustrato nel paragrafo 8.2.1. Il risultato dell'operazione che vediamo nel registro MDR e TOS dopo l'operazione è effettivamente la XOR logica tra i due operandi (0xA e 0xE), cioè "0x4". Notiamo inoltre che l'intero codice operativo è eseguito da 325 ns a 375 ns, cioè vengono utilizzati 6 colpi di clock e ad ogni colpo di clock viene eseguita una microistruzione delle tre viste sopra. E' da notare come il registro OPC, nonostante venga

usato per memorizzare risultato temporanei durante la XOR, alla fine del codice operativo riacquista la sua semantica contenendo l'indirizzo che precede il PC.

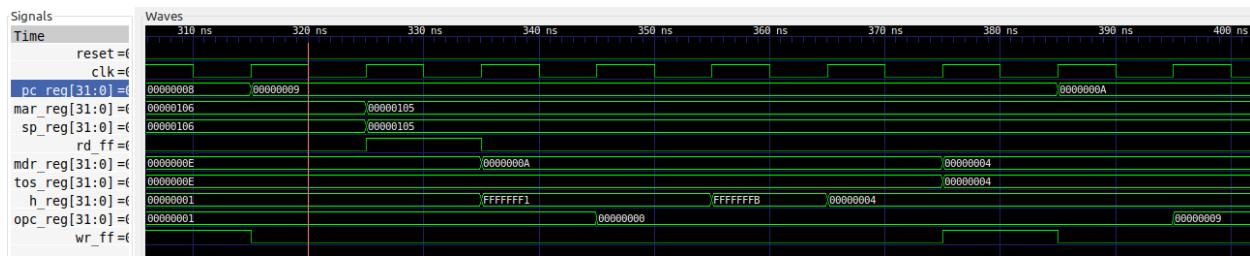


Figura 69: Simulazione del microprogramma che utilizza la ixor

9 Interfaccia seriale

In questo esercizio viene trattato la comunicazione tra 2 nodi attraverso l'uso del componente UART (Universal Asynchronous Receiver Transmitter), l'implementazione è data dalla digilent ed è il componente "RS232". Il protocollo permette di inviare un totale di 8 bit attraverso una singola linea dati, attraverso una conversione del tipo parallelo-seriale, infatti il componente acquisisce i dati in parallelo e li invia su una linea di comunicazione in modo seriale. Come indicato dal nome la trasmissione è asincrona, questo tipo di trasmissione può essere implementata in 3 modi diversi:

- simplex, dove si ha la comunicazione unidirezionale da un nodo all'altro;
- half-duplex, dove i 2 dispositivi a turno ricevono ed inviano segnali;
- full-duplex, dove i 2 dispositivi possono sia ricevere che inviare dati in contemporanea.

Nel caso del componente utilizzato è possibile utilizzare tutte e 3 le implementazioni del protocollo di comunicazione, infatti è possibile utilizzarlo in modo:

- simplex, se si collegano i 2 nodi in modo che solo uno possa trasmettere e solo uno possa ricevere;
- half-duplex, se si gestisce il componente e permettere l'invio dei dati solo se l'altra linea non è in utilizzo;
- full-duplex, se si collegano ai 2 nodi entrambe le linee di comunicazione.

Il protocollo di comunicazione prevede che la frequenza di campionamento del ricevitore sia più alta della velocità di trasmissione dei dati (solitamente di 16 volte). Per poter comprendere a pieno come funziona il protocollo è necessario sapere che di base la linea di comunicazione tra trasmettitore e ricevitore è posta ad 1 (così è possibile anche controllare se c'è un guasto al trasmettitore). Il funzionamento preciso del trasmettitore e del ricevitore viene spiegato più approfonditamente all'interno degli schematici. Infine c'è da specificare che l'UART è un componente hardware che permette a 2 dispositivi di comunicare, ma nel caso dell'esercizio svolto i 2 nodi sono interni alla scheda e non viene utilizzata la porta seriale fisica.

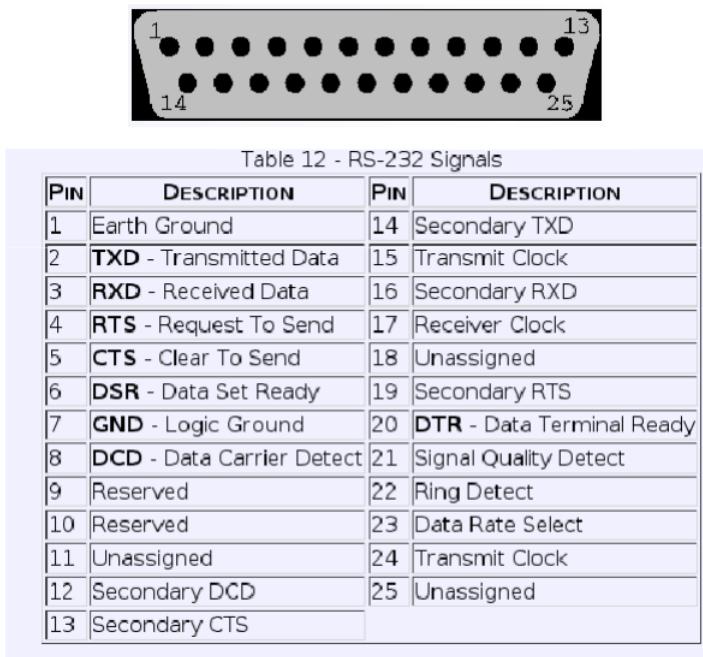


Table 12 - RS-232 Signals

PIN	DESCRIPTION	PIN	DESCRIPTION
1	Earth Ground	14	Secondary TXD
2	TXD - Transmitted Data	15	Transmit Clock
3	RXD - Received Data	16	Secondary RXD
4	RTS - Request To Send	17	Receiver Clock
5	CTS - Clear To Send	18	Unassigned
6	DSR - Data Set Ready	19	Secondary RTS
7	GND - Logic Ground	20	DTR - Data Terminal Ready
8	DCD - Data Carrier Detect	21	Signal Quality Detect
9	Reserved	22	Ring Detect
10	Reserved	23	Data Rate Select
11	Unassigned	24	Transmit Clock
12	Secondary DCD	25	Unassigned
13	Secondary CTS		

Figura 70: porta hardware trasmissione seriale rs232

9.1 Progettazione in VHDL e implementazione su board

Traccia

Sfruttando l'implementazione fornita dalla Digilent di un dispositivo UART (componente RS232RefComp.vhd), progettare e implementare in VHDL un sistema costituito da 2 nodi A e B collegati tra loro mediante una interfaccia seriale. Il sistema A acquisisce una stringa di 8 bit dall'utente (mediante gli switch della board di sviluppo) e la invia mediante la seriale al sistema B, che la manda in output sui led della board di sviluppo.

9.1.1 Schematici

In questo esercizio è stato utilizzato il componente "rs232" (UARTcomponent) con un funzionamento del protocollo di trasmissione di tipo "simplex", dove il Nodo A svolge la funzione di trasmettitore ed il nodo B svolge la funzione di ricevitore. Il nodo A può inviare al nodo B un'informazione alla volta di 8 bit (presi dagli switch) e come richiesto dalla traccia l'informazione (arrivata al nodo B) viene mostrata sui led.

UARTcomponent

L'UARTcomponent, gestisce la comunicazione del componente "rs232" implementando un funzionamento full-duplex, infatti il componente può gestire sia trasmissioni che ricezioni in contemporanea. Per poter permettere la comunicazione tramite UART è necessario impostare il componente sui due nodi, in modo che funzionino su uno stesso baud rate (tasso di trasmissione), la velocità di clock minima per permettere il funzionamento del componente con baud rate pari a 9600 è di 50MHZ (nel caso della scheda avendo un clock di 100MHZ il componente funziona correttamente anche senza l'uso di un divisore di frequenza). L'invio di dati è effettuato attraverso l'invio di una sequenza particolare lungo la linea di trasmissione, infatti il ricevitore osserva una sequenza solo nel momento in cui registra una variazione da '1' a '0' sulla linea dati in ingresso, successivamente registra gli 8 bit in ingresso, registra il parity bit e la linea torna alta in uno stato di idle (un esempio della comunicazione è presente nella figura 71).



Figura 71: esempio comunicazione seriale

Visto che nello stesso componente è sviluppata sia la parte del trasmettitore che la parte del ricevitore per una migliore chiarezza del funzionamento del componente verranno distinte le 2 implementazioni nei successivi paragrafi.

UARTcomponent - trasmettitore Per utilizzare un UARTcomponent come trasmettitore è necessario definire:

- baud rate, cioè la frequenza di trasmissione (che deve essere uguale a quello dell'altro componente);
- DBIN, cioè la linea di dati paralleli in ingresso (che dovranno essere inviati all'altro nodo);
- TXD, cioè la linea di trasmissione con l'altro nodo;
- WR, cioè il segnale che avvia il trasferimento dei dati in ingresso;
- TBE, il segnale che avvisa che il buffer di dati in invio è vuoto e quindi è possibile inviare nuovi dati.

Dopo aver definito correttamente tutti i segnali necessari per il normale funzionamento del componente è possibile definire la sua parte operativa e la sua parte di controllo. La parte operativa è formata da:

- 2 contatori, un contatore per indicare quanti valori sono stati inviati (infatti arriverà a 9 finisce la fase di trasmissione) e l'altro per creare il delay tra l'invio di un bit e l'altro;
- una control unit per gestire il trasferimento dei valori;
- uno shift register di 11 bit con i valori del segnale che va inviato sulla linea di comunicazione (il valore in ingresso allo shift register è: '1' & parity-bit & dati & '0').

I collegamenti tra i vari elementi del componente sono presenti in figura 72.

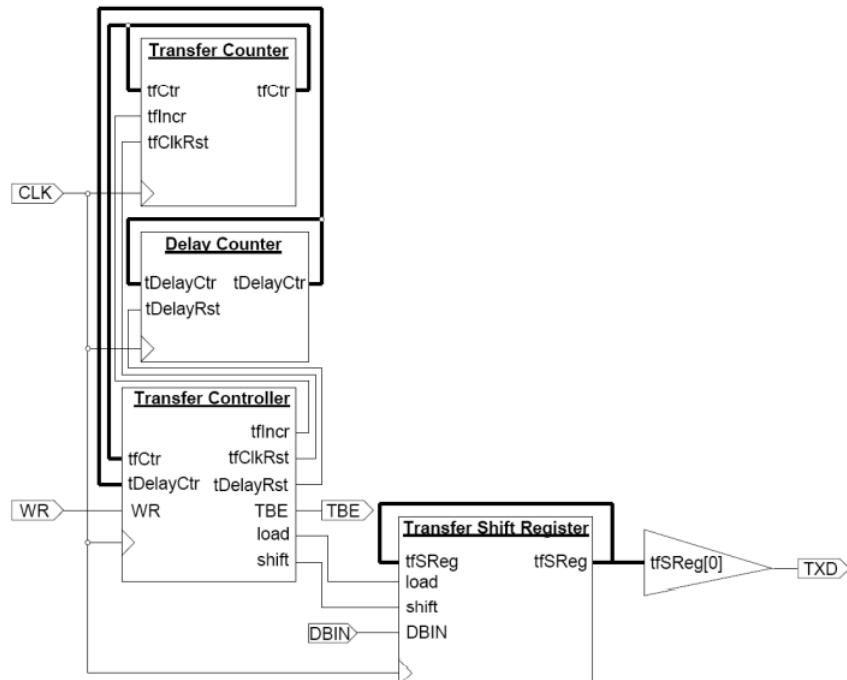


Figura 72: collegamenti trasmittitore rs232

La control unit del componente svolge la funzione della parte operativa e definisce il comportamento di una FSM con i seguenti stati:

- IDLE, che è lo stato in cui il trasmittitore attende il segnale per avviare la trasmissione;

- TRANSFER, che è lo stato in cui lo shift register viene caricato e vengono resettati i 2 contatori;
- SHIFT, che è lo stato in cui si ha lo shift allo shift register e viene incrementato il contatore dei valori inviati;
- DELAY, che è lo stato in cui si attende il delay tra un invio e l'altro;
- CHECKSTOP, che è lo stato in cui si attende che il segnale wr si abbassa.

L'automa è rappresentato in figura 73.

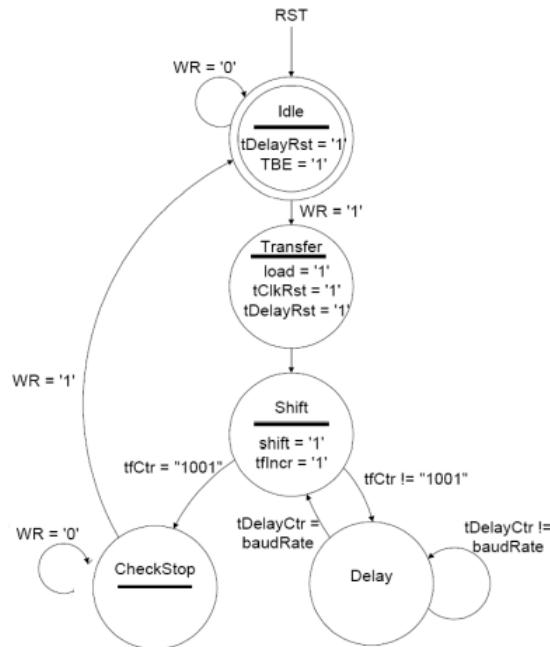


Figura 73: automa trasmettitore rs232

UARTcomponent - ricevitore Per utilizzare un uart come ricevitore è necessario definire:

- baud rate, cioè la frequenza di trasmissione (che deve essere uguale a quello dell'altro componente);
- DBOUT, cioè la linea di dati paralleli in USCITA (che sono stati ricevuto dall'altro nodo);

- RXD, cioè la linea di ricezione con l'altro nodo;
- RDA, cioè il segnale che avvisa della presenza di dati ricevuti (READ DATA AVAILABLE)
- RD, cioè il segnale che avvisa la lettura effettuata su DBOUT da parte del ricevitore;
- OE, cioè la flag di overwrite error (che si alza in caso c'è stata una sovrascrittura dei dati);
- PE, cioè la flag di parity error (che si alza se c'è un errore sulla parità dei bit);
- FE, cioè la flag di frame error (che si alza nel caso c'è un forte sfasamento tra i 2 nodi ed il nodo ricevitore non riesce a leggere correttamente i dati in ingresso).

I collegamenti tra i vari elementi del componente sono presenti in figura 74.

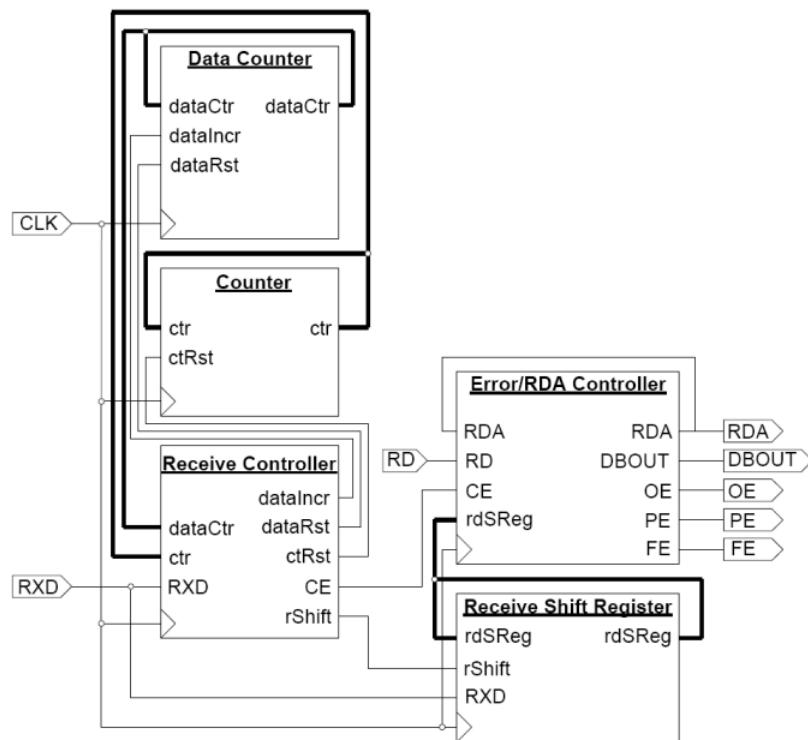


Figura 74: collegamenti ricevitore rs232

Dopo aver definito correttamente tutti i segnali necessari per il normale funzionamento del componente è possibile definire la sua parte operativa e la sua parte di controllo. La parte operativa è formata da:

- 2 contatori, un contatore per indicare quanti valori sono stati inviati (infatti arrivato a 9 finisce la fase di trasmissione) e l'altro per calcolare il delay tra l'invio di un bit e l'altro;
- una control unit per gestire la ricezione dei valori;
- uno shift register di 10 bit con i bit ricevuti dal trasmettitore (lo shift register affinché la comunicazione sia andata a buon fine deve avere: '1' & parity-bit & dati).
- una control unit per comunicare la presenza di dati disponibili da leggere, riconoscere la lettura dei dati, riconoscere e avvertire i casi di errore.

La parte di controllo é formata da 2 control unit differenti, una per gestire la ricezione dei dati da un altro nodo e l'altra per avvertire il nodo della ricezione dei dati e di eventuali errori. la control unit che gestisce la ricezione dei dati da un altro nodo é strutturata sulla base di una FSM con i seguenti stati:

- IDLE, che é lo stato in cui si attende l'inizio della ricezione dei dati da parte del nodo trasmettitore (durante questo stato vengono resettati i 2 contatori);
- EIGHTDELAY, che é lo stato in cui si attende di arrivare alla metá del primo bit ricevuto (durante questo stato viene resettato il contatore con il valore di elementi ricevuti);
- WaitFor0, che é lo stato in cui si attende la fine del tempo di attesa di un segnale e si controlla se sono stati letti 10 valori (la fine del tempo di attesa di un segnale é indicata dall'azzerarsi del contatore di delay);
- WaitFor1, che é lo stato in cui si aspetta metá del tempo di delay (8 colpi di clock in questo caso) per poter poi passare all'acquisizione del dato;
- GetData, che é lo stato in cui si esegue lo shift dello shift register;
- CHECKSTOP, che é lo stato in cui si considera finita la comunicazione e si invia il segnale CE all'altra control unit per poter avviare il controllo degli errori ed avvisare all'utente della disponibilitá dei dati (dopo questo stato si ritorna in IDLE).

L'automa é rappresentato in figura 75.

Infine l'altra control unit é sviluppata in modo comportamentale ed attende un segnale di CE in ingresso pari ad 1 per poter avviare tutti i controlli sui dati in ingresso, dopo aver

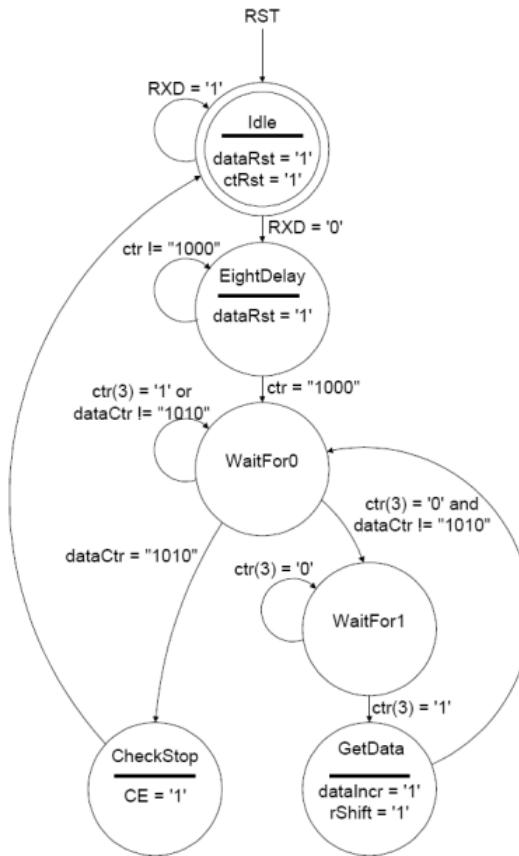


Figura 75: automa ricevitore rs232

effettuato i vari controlli la control unit manda in uscita RDA='1' per avvisare all'utente che sono stati ricevuti dei dati ed alza OE, FE o PE in caso della presenza di errori.

Nodo Trasmettitore

control_unit La control unit del trasmettitore é utilizzata per poter gestire la comunicazione tramite con componente rs232. La control unit é stata sviluppata a partire da un automa che ha i seguenti stati:

- IDLE, che é lo stato in cui si attende l'arrivo del segnale di inizio trasmissione;
- INIZIO_COMUNICAZIONE, che é la stato in cui si avverte il componente rs232 di voler trasmettere i dati in ingresso (ponendo WR=1), l'automa rimane in questo stato finché il componente non avvia la trasmissione (leggendo TBE = '0');
- FINE_COMUNICAZIONE, che é lo stato in cui si attende che finisca la trasmissione dei dati (TBE='1'), durante questo stato viene posto l'ingresso WR = '0'.

L'automa è in figura 76.

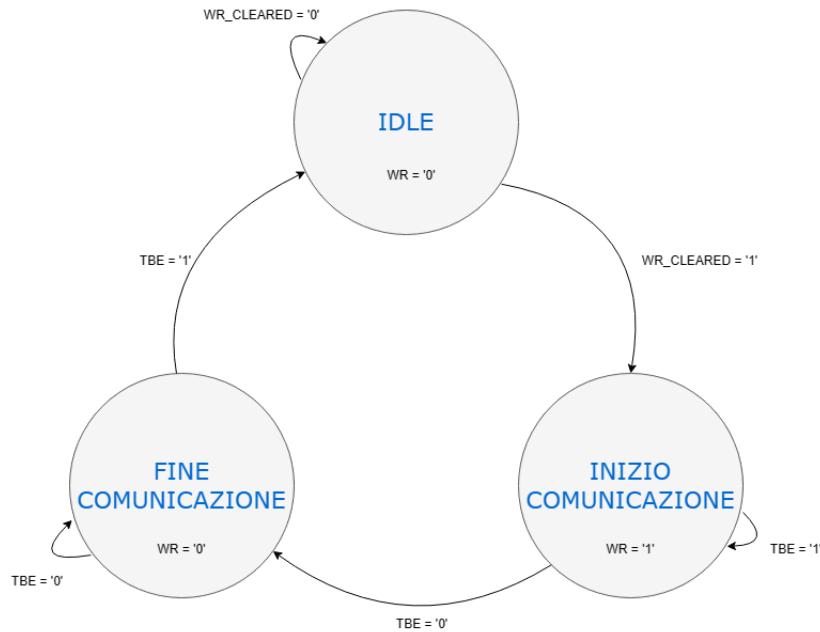


Figura 76: Control unit nodo trasmettitore esercizio 9

Collegamenti Il nodo trasmettitore è formato solo dalla control unit e dal componente rs232, il componente è stato inserito in modo da avere solo la funzione di trasmettitore e quindi non è abilitato a ricevere dati dall'esterno. I collegamenti tra i 2 componenti sono in figura 77.

Nodo ricevitore

uc_receiver il componente gestisce il controllo della ricezione di dati dal componente rs232, della lettura e della conferma della lettura. Il componente è stato sviluppato come un automa a stati finiti con i seguenti stati:

- attesa_dati, dove si attende che il componente rs232 metta l'uscita RDA = '1' (avviando della presenza di dati da leggere);
- lettura_dati, dove vengono salvati i valori interni al componente rs232 e vengono messi sui led in uscita dal componente;
- conferma_lettura, dove si resetta il componente rs232 e si ritorna allo stato attesa_dati.

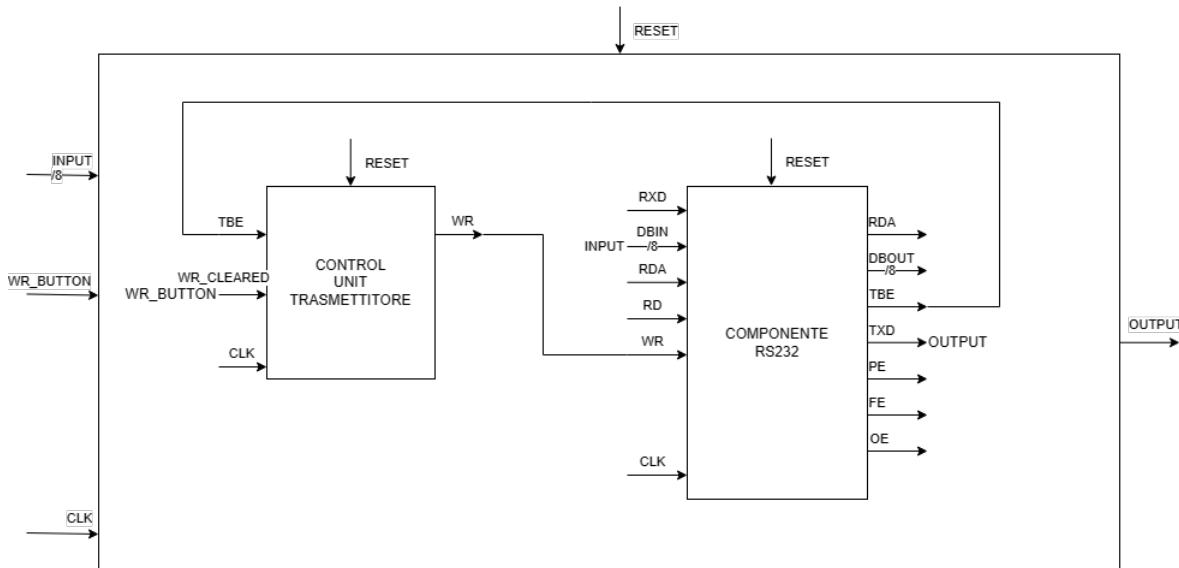


Figura 77: collegamenti nodo trasmettitore esercizio 9

- reset_state, che è uno stato in cui si resetta il nodo del ricevitore.

L'automa è rappresentato nella figura: 78.

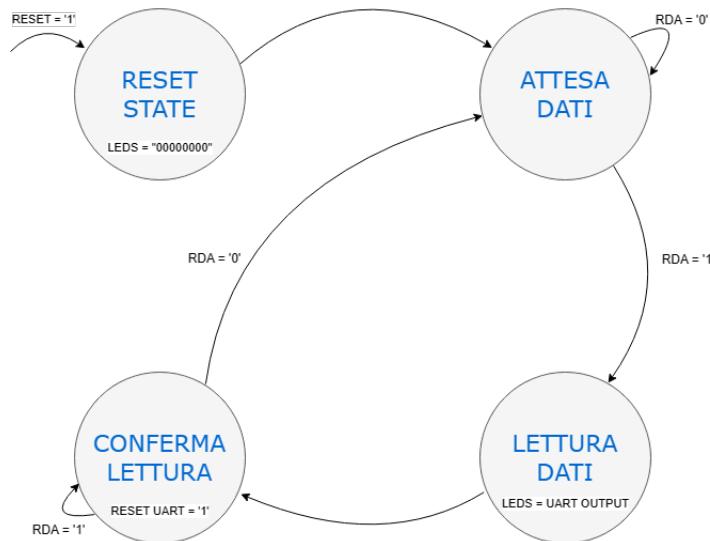


Figura 78: automa uc_receiver

Collegamenti Il nodo ricevitore è formato solo dalla control unit e dal componente rs232, il componente è stato inserito in modo da avere solo la funzione di ricevitore e quindi

non é abilitato ad inviare dati all'esterno. I collegamenti tra i 2 componenti sono in figura 79.

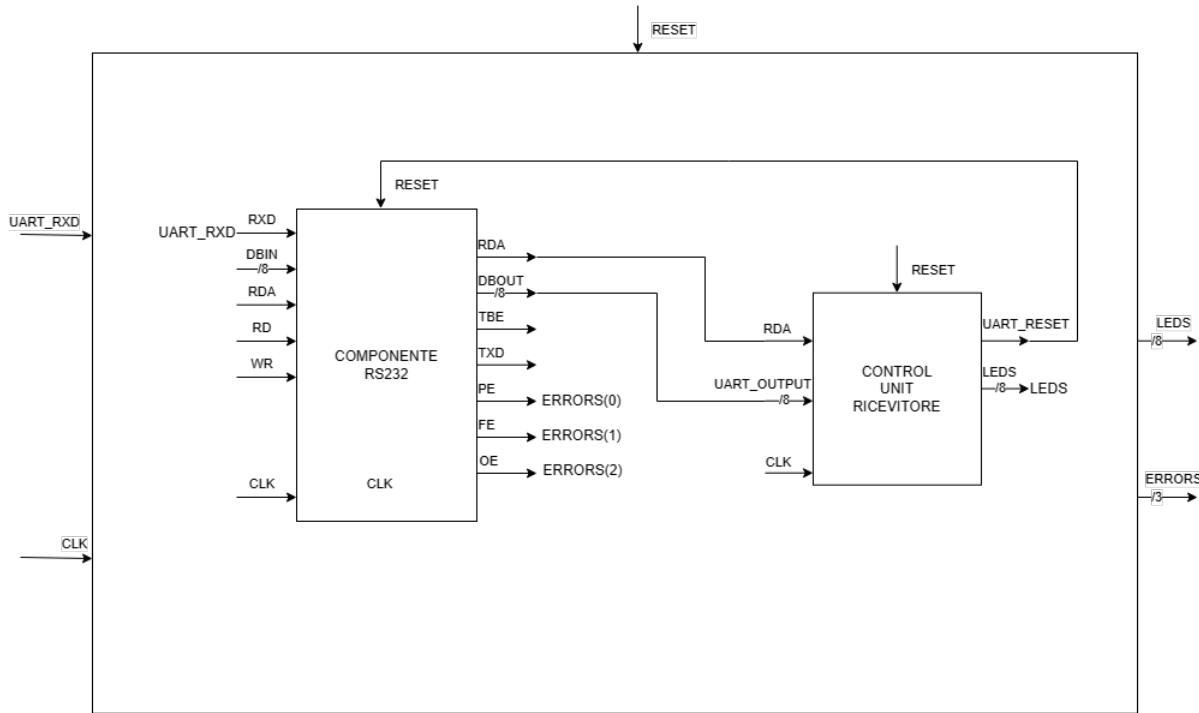


Figura 79: collegamenti nodo ricevitore esercizio 9

ButtonDebouncer

il componente buttondebouncer é lo stesso di quello sviluppato nell'esercizio 3, la sua descrizione é sviluppata nel paragrafo 3.2.1

comunicazione_seriale

Il componente é stato sviluppato per collegare il buttondebouncer del segnale di avvio trasmissione ed i dati in ingresso con il trasmettitore e quest'ultimo con il ricevitore. La connessione tra gli elementi é presente in figura 80

9.1.2 Codice VHDL

UARTcomponent

Il componente utilizzato é stato sviluppato da parte della Digilent, dopo un piccolo studio é possibile notare che é stato sviluppato completamente seguendo una descrizione comporta-

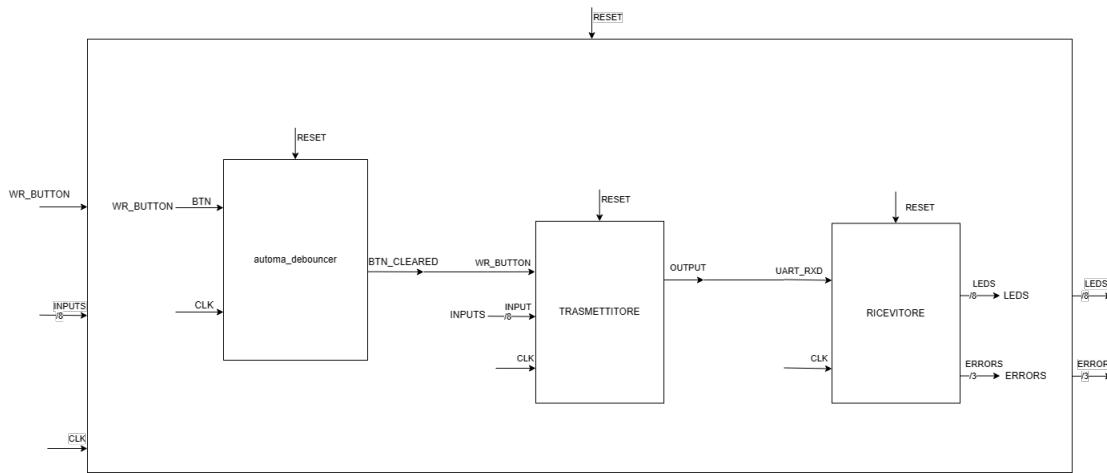


Figura 80: collegamenti componente comunicazione seriale

mentale ed il file é stato commentato correttamente per la distinzione di tutti i componenti. Di seguito é presente il codice VHDL del componente.

```

1  -----
2  -- uartcomponent.vhd
3  -----
4  -- Author: Dan Pederson
5  -- Copyright 2004 Digilent, Inc.
6  -----
7  -- Description: This file defines a UART which transfers data to and
8  -- from serial and parallel information. It requires two
9  -- major processes: receiving and transferring. The
10 -- receiving portion reads serially transmitted data, and
11 -- converts it into parallel data, while the transferring
12 -- portion reads parallel data, and transmits it as serial
13 -- data. There are three error signals provided with this
14 -- UART. They are frame error, parity error, and overwrite
15 -- error signals. This UART is configured to use an ODD
16 -- parity bit at a baud rate of 9600.
17 --
18  -----
19  -- Revision History:
20  -- 07/15/04 (DanP) Created
21  -- 05/24/05 (DanP) Updated commenting style
22  -- 06/06/05 (DanP) Synchronized state machines to fix timing bug
23  -----
24
25 library IEEE;
26 use IEEE.STD_LOGIC_1164.ALL;
27 use IEEE.STD_LOGIC_ARITH.ALL;
28 use IEEE.STD_LOGIC_UNSIGNED.ALL;
29
30  -----
31  --
32 -- Title: UARTcomponent entity
33  --
34 -- Inputs: 7 : RXD

```

```

35      -- CLK
36      -- DBIN
37      -- RDA
38      -- RD
39      -- WR
40      -- RST
41
42      -- Outputs: 7 : TXD
43      --          DBOUT
44      --          RDA
45      --          TBE
46      --          PE
47      --          FE
48      --          OE
49
50      -- Description: This describes the UART component entity. The inputs are
51      --          the Pegasus 50 MHz clock, a reset button, The RXD from
52      --          the serial cable, an 8-bit data bus from the parallel
53      --          port, and Read Data Available (RDA)and Transfer Buffer
54      --          Empty(TBE) handshaking signals. The outputs are the TXD
55      --          signal for the serial port, an 8-bit data bus for the
56      --          parallel port, RDA and TBE handshaking signals, and three
57      --          error signals for parity, frame, and overwrite errors.
58
59 -----
60 entity UARTcomponent is
61   Generic (
62     --@48MHz
63     BAUD_DIVIDE_G : integer := 26;    --115200 baud
64     BAUD_RATE_G   : integer := 417
65
66     --@26.6MHz
67     BAUD_DIVIDE_G : integer := 14;    --115200 baud
68     BAUD_RATE_G   : integer := 231
69   );
70   Port (
71     TXD  : out  std_logic  := '1';      -- Transmitted serial data output
72     RXD  : in   std_logic;           -- Received serial data input
73     CLK  : in   std_logic;           -- Clock signal
74     DBIN : in   std_logic_vector(7 downto 0); -- Input parallel data to be transmitted
75     DBOUT: out  std_logic_vector(7 downto 0); -- Recevived parallel data output
76     RDA  : inout std_logic;           -- Read Data Available
77     TBE  : out  std_logic  := '1';      -- Transfer Buffer Emty
78     RD   : in   std_logic;           -- Read Strobe
79     WR   : in   std_logic;           -- Write Strobe
80     PE   : out  std_logic;           -- Parity error
81     FE   : out  std_logic;           -- Frame error
82     OE   : out  std_logic;           -- Overwrite error
83     RST  : in   std_logic := '0');    -- Reset signal
84
85 end UARTcomponent;
86
87 architecture Behavioral of UARTcomponent is
88
89 -----
90   -- Local Type and Signal Declarations
91 -----
92
93 -----
94   -- Title: Local Type Declarations

```

```

95 --
96 --Description: There are two state machines used in this entity. The
97 --      rstate is used to synchronize the receiving portion of
98 --      the UART, and the tstate is used to synchronize the
99 --      sending portion of the UART.
100 --
101 -----
102 type rstate is (
103     strIdle,
104     strEightDelay,
105     strGetData,
106     strWaitFor0,
107     strWaitFor1,
108     strCheckStop
109 );
110
111 type tstate is (
112     sttIdle,
113     sttTransfer,
114     sttShift,
115     sttDelay,
116     sttWaitWrite
117 );
118
119 -----
120 --
121 --Title: Local Signal Declarations
122 --
123 --Description: The constants and signals used by this entity are
124 --      described below:
125 --
126 --      -baudRate : This is the Baud Rate constant used to
127 --                  synchronize the Pegasus 50 MHz clock with a
128 --                  baud rate of 9600. To get this number, divide
129 --                  50MHz by 9600.
130 --      -baudDivide : This is the Baud Rate divider used to safely
131 --                  read data transmitted at a baud rate of 9600.
132 --                  It is simply the above described baudRate
133 --                  constant divided by 16.
134 --
135 --      -rdReg      : this is the receive holding register
136 --      -rdSReg    : this is the receive shift register
137 --      -tfReg      : this is the transfer holding register
138 --      -tfSReg    : this is the transfer shifting register
139 --      -clkDiv    : counter used to get rClk
140 --      -ctr       : used for delay times
141 --      -tfCtr     : used to delay in the transfer process
142 --      -dataCtr   : counts the number of read data bits
143 --      -parError  : parity error bit
144 --      -frameError : frame error bit
145 --      -CE        : clock enable bit for the writing latch
146 --      -ctRst     : reset for the ctr
147 --      -load      : load signal used to load the transfer shift
148 --                  register
149 --      -shift     : shift signal used to unload the transfer
150 --                  shift register
151 --      -par       : represents the parity in the transfer
152 --                  holding register
153 --      -tClkRST   : reset for the tfCtr
154 --      -rShift    : shift signal used to load the receive shift

```

```

155      register
156      -dataRST : reset for the dataCtr
157      -dataIncr : signal to increment the dataCtr
158      -tfIncr : signal to increment the tfCtr
159      -tDelayCtr : counter used to delay the transfer state
160      machine.
161      -tDelayRst : reset signal for the tDelayCtr counter.
162
163      The following signals are used by the two state machines
164      for state control:
165      -Receive State Machine : strCur, strNext
166      -Transfer State Machine : sttCur, sttNext
167
168 -----
169
170      -- @26.7MHz
171      constant baudRate : std_logic_vector(12 downto 0) := "1 0100 0101 1000";
172      constant baudRate : std_logic_vector(12 downto 0) := conv_std_logic_vector(1406,13); --
19200
173      constant baudRate : std_logic_vector(12 downto 0) := conv_std_logic_vector(703,13); --
38400
174      constant baudRate : std_logic_vector(12 downto 0) := conv_std_logic_vector(469,13); --
57600
175      constant baudRate : std_logic_vector(12 downto 0) := conv_std_logic_vector(417,13);
--115200
176
177      -- @26.7MHz
178      constant baudDivide : std_logic_vector(8 downto 0) := conv_std_logic_vector(1,9); --
Used for simulation
179      constant baudDivide : std_logic_vector(8 downto 0) := conv_std_logic_vector(88,9); --
Used for 19 200 baud
180      constant baudDivide : std_logic_vector(8 downto 0) := conv_std_logic_vector(44,9); --
Used for 38 400 baud
181      constant baudDivide : std_logic_vector(8 downto 0) := conv_std_logic_vector(29,9); --
Used for 57 600 baud
182      constant baudDivide : std_logic_vector(8 downto 0) := conv_std_logic_vector(26,9); --
Used for 115 200 baud
183
184      constant baudRate : std_logic_vector(12 downto 0) := conv_std_logic_vector(BAUD_RATE_G
,13); --115200
185      constant baudDivide : std_logic_vector(8 downto 0) := conv_std_logic_vector(
BAUD_DIVIDE_G-1,9); -- Used for 115 200 baud
186
187      signal rdReg : std_logic_vector(7 downto 0) := "00000000";
188      signal rdsReg : std_logic_vector(9 downto 0) := "1111111111";
189      signal tfReg : std_logic_vector(7 downto 0);
190      signal tfsReg : std_logic_vector(10 downto 0) := "1111111111";
191      signal clkDiv : std_logic_vector(9 downto 0) := "0000000000";
192      signal ctr : std_logic_vector(3 downto 0) := "0000";
193      signal tfCtr : std_logic_vector(3 downto 0) := "0000";
194
195      signal dataCtr : std_logic_vector(3 downto 0) := "0000";
196      signal parError : std_logic;
197      signal frameError : std_logic;
198      signal CE : std_logic;
199      signal ctrRst : std_logic := '0';
200      signal load : std_logic := '0';
201      signal shift : std_logic := '0';
202      signal par : std_logic;
203      signal tClkRST : std_logic := '0';

```

```

203 signal rShift      : std_logic      := '0';
204 signal dataRST     : std_logic      := '0';
205 signal dataIncr    : std_logic      := '0';
206 signal tfIncr     : std_logic      := '0';
207 signal tDelayCtr   : std_logic_vector(12 downto 0);
208 signal tDelayRst   : std_logic      := '0';
209
210 signal strCur      : rstate       := strIdle;
211 signal strNext     : rstate;
212 signal sttCur      : tstate        := sttIdle;
213 signal sttNext     : tstate;
214
215 -----
216 -- Module Implementation
217 -----
218 begin
219 -----
220 --
221 --Title: Initial signal definitions
222 --
223 --Description: The following lines of code define 4 internal and 1
224 --              external signal. The most significant bit of the rdSReg
225 --              signifies the frame error bit, so frameError is tied to
226 --              that signal. The parError is high if there is a parity
227 --              error, so it is set equal to the inverse of rdSReg(8)
228 --              XOR-ed with the data bits. In this manner, it can
229 --              determine if the parity bit found in rdSReg(8) matches
230 --              the data bits. The parallel information output is equal
231 --              to rdReg, so DBOUT is set equal to rdReg. Likewise, the
232 --              input parallel information is equal to DBIN, so tfReg is
233 --              set equal to DBIN. Because the tfSReg is used to shift
234 --              out transmitted data, the TXD port is set equal to the
235 --              first bit of tfsReg. Finally, the par signal represents
236 --              the parity of the data, so par is set to the inverse of
237 --              the data bits XOR-ed together. This UART can be changed
238 --              to use EVEN parity if the "not" is omitted from the par
239 --              definition.
240 --
241 -----
242 frameError <= not rdSReg(9);
243 parError <= not ( rdSReg(8) xor (((rdSReg(0) xor rdSReg(1)) xor
244 (rdSReg(2) xor rdSReg(3))) xor ((rdSReg(4) xor rdSReg(5)) xor
245 (rdSReg(6) xor rdSReg(7)))) );
246 DBOUT <= rdReg;
247 tfReg <= DBIN;
248 TXD <= tfsReg(0);
249 par <= not ( ((tfReg(0) xor tfReg(1)) xor (tfReg(2) xor tfReg(3))) xor
250 ((tfReg(4) xor tfReg(5)) xor (tfReg(6) xor tfReg(7))) );
251 -----
252 --
253 --Title: Clock Divide counter
254 --
255 --Description: This process defines clkDiv as a signal that increments
256 --              with the clock up until it is either reset by ctRst, or
257 --              equals baudDivide. This signal is used to define a
258 --              counter called ctr that increments at the rate of the
259 --              divided baud rate.
260 --
261 -----
262 process (CLK, clkDiv)

```

```

263 begin
264   if (CLK = '1' and CLK'event) then
265     if (clkDiv = baudDivide or ctRst = '1') then
266       clkDiv <= "0000000000";
267     else
268       clkDiv <= clkDiv +1;
269     end if;
270   end if;
271 end process;
272 -----
273 --
274 --Title: Transfer delay counter
275 --
276 --Description: This process defines tDelayCtr as a counter that runs
277 --              until it equals baudRate, or until it is reset by
278 --              tDelayRst. This counter is used to measure delay times
279 --              when sending data out on the TXD signal. When the
280 --              counter is equal to baudRate, or is reset, it is set
281 --              equal to 0.
282 --
283 -----
284 process (CLK, tDelayCtr)
285 begin
286   if (CLK = '1' and CLK'event) then
287     if (tDelayCtr = baudRate or tDelayRst = '1') then
288       tDelayCtr <= "00000000000000";
289     else
290       tDelayCtr <= tDelayCtr+1;
291     end if;
292   end if;
293 end process;
294 -----
295 --
296 --Title: ctr set up
297 --
298 --Description: This process sets up ctr, which uses clkDiv to count
299 --              increase at a rate needed to properly receive data in
300 --              from RXD. If ctRst is strobed, the counter is reset. If
301 --              clkDiv is equal to baudDivide, then ctr is incremented
302 --              once. This signal is used by the receiving state machine
303 --              to measure delay times between RXD reads.
304 --
305 -----
306 process (CLK)
307 begin
308   if CLK = '1' and CLK'Event then
309     if ctRst = '1' then
310       ctr <= "0000";
311     elsif clkDiv = baudDivide then
312       ctr <= ctr + 1;
313     else
314       ctr <= ctr;
315     end if;
316   end if;
317 end process;
318 -----
319 --
320 --Title: transfer counter
321 --
322 --Description: This process makes tfCtr increment whenever the tfIncr

```

```

323 --      signal is strobed high. If the tClkRst signal is strobed
324 --      high, the tfCtr is reset to "0000." This counter is used
325 --      to keep track of how many data bits have been
326 --      transmitted.
327 --
328 -----
329 process (CLK, tClkRST)
330 begin
331   if (CLK = '1' and CLK'event) then
332     if tClkRST = '1' then
333       tfCtr <= "0000";
334     elsif tfIncr = '1' then
335       tfCtr <= tfCtr +1;
336     end if;
337   end if;
338 end process;
339 -----
340 --
341 --Title: Error and RDA flag controller
342 --
343 --Description: This process controls the error flags FE, OE, and PE, as
344 --      well as the Read Data Available (RDA) flag. When CE goes
345 --      high, it means that data has been read into the rdSReg.
346 --      This process then analyzes the read data for errors, sets
347 --      rdReg equal to the eight data bits in rdSReg, and flags
348 --      RDA to indicate that new data is present in rdReg. FE
349 --      and PE are simply equal to the frameError and parError
350 --      signals. OE is flagged high if RDA is already high when
351 --      CE is strobed. This means that unread data was still in
352 --      the rdReg when it was written over with the new data.
353 --
354 -----
355 process (CLK, RST, RD, CE)
356 begin
357   if RD = '1' or RST = '1' then
358     FE <= '0';
359     OE <= '0';
360     RDA <= '0';
361     PE <= '0';
362   elsif CLK = '1' and CLK'event then
363     if CE = '1' then
364       FE <= frameError;
365       PE <= parError;
366       rdReg(7 downto 0) <= rdSReg (7 downto 0);
367     if RDA = '1' then
368       OE <= '1';
369     else
370       OE <= '0';
371       RDA <= '1';
372     end if;
373   end if;
374 end if;
375 end process;
376 -----
377 --
378 --Title: Receiving shift register
379 --
380 --Description: This process controls the receiving shift register
381 --      (rdSReg). Whenever rShift is high, implying that data
382 --      needs to be shifted in, rdSReg is shifted in RXD to the

```

```

383 --      most significant bit, while shifting its existing data
384 --      right.
385 --
386 -----
387 process (CLK, rShift)
388 begin
389   if CLK = '1' and CLK'Event then
390     if rShift = '1' then
391       rdSReg <= (RxD & rdSReg(9 downto 1));
392     end if;
393   end if;
394 end process;
395 -----
396 --
397 --Title: Incoming Data counter
398 --
399 --Description: This process controls the dataCtr to keep track of
400 --      shifted values into the rdSReg. The dataCtr signal is
401 --      incremented once every time dataIncr is strobed high.
402 --
403 -----
404
405 process (CLK, dataRST)
406 begin
407   if (CLK = '1' and CLK'event) then
408     if dataRST = '1' then
409       dataCtr <= "0000";
410     elsif dataIncr = '1' then
411       dataCtr <= dataCtr +1;
412     end if;
413   end if;
414 end process;
415 -----
416 --
417 --Title: Receiving State Machine controller
418 --
419 --Description: This process takes care of the Receiving state machine
420 --      movement. It causes the next state to be evaluated on
421 --      each rising edge of CLK. If the RST signal is strobed,
422 --      the state is changed to the default starting state,
423 --      which is strIdle
424 --
425 -----
426 process (CLK, RST)
427 begin
428   if CLK = '1' and CLK'Event then
429     if RST = '1' then -- nadj
430       strCur <= strIdle;
431     else
432       strCur <= strNext;
433     end if;
434   end if;
435 end process;
436 -----
437 --
438 --Title: Receiving State Machine
439 --
440 --Description: This process contains all of the next state logic for the
441 --      Receiving state machine.
442 --

```

```

443 -----
444 process (strCur, ctr, RXD, dataCtr)
445 begin
446 case strCur is
447 -----
448 --
449 --Title: strIdle state
450 --
451 --Description: This state is the idle and startup default stage for the
452 -- Receiving state machine. The machine stays in this state
453 -- until the RXD signal goes low. When this occurs, the
454 -- ctrRst signal is strobed to reset ctr for the next state,
455 -- which is strEightDelay.
456 --
457 -----
458 when strIdle =>
459   dataIncr <= '0';
460   rShift <= '0';
461   dataRst <= '1';
462   CE <= '0';
463   ctrRst <= '1';
464
465   if RXD = '0' then
466     strNext <= strEightDelay;
467   else
468     strNext <= strIdle;
469   end if;
470 -----
471 --
472 --Title: strEightDelay state
473 --
474 --Description: This state simply delays the state machine for eight clock
475 -- cycles. This is needed so that the incoming RXD data
476 -- signal is read in the middle of each data emission. This
477 -- ensures an accurate RXD signal reading. ctr counts from
478 -- 0 to 8 to keep track of rClk cycles. When it equals 8
479 -- (1000) the next state, strWaitFor0, is loaded. During
480 -- this state, the dataRst signal is strobed high to reset
481 -- the shift-in data counter (dataCtr).
482 --
483 -----
484 when strEightDelay =>
485   dataIncr <= '0';
486   rShift <= '0';
487   dataRst <= '1';
488   CE <= '0';
489   ctrRst <= '0';
490
491   if ctr(3 downto 0) = "1000" then
492     strNext <= strWaitFor0;
493   else
494     strNext <= strEightDelay;
495   end if;
496 -----
497 --
498 --Title: strGetData state
499 --
500 --Description: In this state, the dataIncr and rShift signals are
501 -- strobbed high for one clock cycle. By doing this, the
502 -- rdSReg shift register shifts in RXD once, while the

```

```

503 --      dataCtr is incremented by one. This state simply
504 --      captures the incoming data on RXD into the rdSReg shift
505 --      register. The next state loaded is strWaitFor0, which
506 --      starts the two delay states needed between data shifts.
507 --
508 -----
509     when strGetData =>
510         CE <= '0';
511         dataRst <= '0';
512         ctrRst <= '0';
513         dataIncr <= '1';
514         rShift <= '1';
515
516         strNext <= strWaitFor0;
517 -----
518 --
519 -- Title: strWaitFor0 state
520 --
521 -- Description: This state is a delay state, which delays the receive
522 --      state machine if not all of the incoming serial data has
523 --      not been shifted in yet. If dataCtr does not equal 10
524 --      (1010), the state is stayed in until the fourth bit of
525 --      ctr is equal to 1. When this happens, half of the delay
526 --      has been achieved, and the second delay state is loaded,
527 --      which is strWaitFor1. If dataCtr does equal 10 (1010),
528 --      all of the needed data has been acquired, so the
529 --      strCheckStop state is loaded to check for errors and
530 --      reset the receive state machine.
531 --
532 -----
533     when strWaitFor0 =>
534         CE <= '0';
535         dataRst <= '0';
536         ctrRst <= '0';
537         dataIncr <= '0';
538         rShift <= '0';
539
540         if dataCtr = "1010" then
541             strNext <= strCheckStop;
542         elsif ctr(3) = '0' then
543             strNext <= strWaitFor1;
544         else
545             strNext <= strWaitFor0;
546         end if;
547 -----
548 --
549 -- Title: strEightDelay state
550 --
551 -- Description: This state is much like strWaitFor0, except it waits for
552 --      the fourth bit of ctr to equal 1. Once this occurs, the
553 --      strGetData state is loaded in order to shift in the next
554 --      data bit from RXD. Because strWaitFor0 is the only state
555 --      that calls this state, no other signals need to be
556 --      checked.
557 --
558 -----
559     when strWaitFor1 =>
560         CE <= '0';
561         dataRst <= '0';
562         ctrRst <= '0';

```

```

563      dataIncr <= '0';
564      rShift <= '0';
565
566      if ctr(3) = '0' then
567          strNext <= strWaitFor1;
568      else
569          strNext <= strGetData;
570      end if;
571
572  --
573 --Title: strCheckStop state
574 --
575 --Description: This state allows the newly acquired data to be checked
576 --              for errors. The CE flag is strobed to start the
577 --              previously defined error checking process. This state is
578 --              passed straight through to the strIdle state.
579 --
580
581      when strCheckStop =>
582          dataIncr <= '0';
583          rShift <= '0';
584          dataRst <= '0';
585          ctrRst <= '0';
586          CE <= '1';
587          strNext <= strIdle;
588      end case;
589  end process;
590
591  --
592 --Title: Transfer shift register controller
593 --
594 --Description: This process uses the load, shift, and clk signals to
595 --              control the transfer shift register (tfSReg). Once load
596 --              is equal to '1', the tfSReg gets a '1', the parity bit,
597 --              the data bits found in tfReg, and a '0'. Under this
598 --              format, the shift register can be used to shift out the
599 --              appropriate signal to serially transfer the data. The
600 --              data is shifted out of the tfSReg whenever shift = '1'.
601 --
602
603  process (load, shift, CLK, tfSReg)
604  begin
605      if CLK = '1' and CLK'Event then
606          if load = '1' then
607              tfSReg (10 downto 0) <= ('1' & par & tfReg(7 downto 0) &'0');
608          elsif shift = '1' then
609              tfSReg (10 downto 0) <= ('1' & tfSReg(10 downto 1));
610          end if;
611      end if;
612  end process;
613
614  --
615 --Title: Transfer State Machine controller
616 --
617 --Description: This process takes care of the Transfer state machine
618 --              movement. It causes the next state to be evaluated on
619 --              each rising edge of CLK. If the RST signal is strobed,
620 --              the state is changed to the default starting state, which
621 --              is sttIdle.
622

```

```

623 -----
624 process (CLK, RST)
625 begin
626   if (CLK = '1' and CLK'Event) then
627     if RST = '1' then
628       sttCur <= sttIdle;
629     else
630       sttCur <= sttNext;
631     end if;
632   end if;
633 end process;
634 -----
635 --
636 --Title: Transfer State Machine
637 --
638 --Description: This process controls the next state logic in the
639 --      transfer state machine. The transfer state machine
640 --      controls the shift and load signals that are used to load
641 --      and transmit the parallel data in a serial form. It also
642 --      controls the Transmit Buffer Empty (TBE) signal that
643 --      indicates if the transmit buffer (tfSReg) is in use or
644 --      not.
645 --
646 -----
647 process (sttCur, tfCtr, WR, tDelayCtr)
648 begin
649   case sttCur is
650 -----
651 --
652 --Title: sttIdle state
653 --
654 --Description: This state is the idle and startup default stage for the
655 --      transfer state machine. The state is stayed in until
656 --      the WR signal goes high. Once it goes high, the
657 --      sttTransfer state is loaded. The load and shift signals
658 --      are held low in the sttIdle state, while the TBE signal
659 --      is held high to indicate that the transmit buffer is not
660 --      currently in use. Once the idle state is left, the TBE
661 --      signal is held low to indicate that the transfer state
662 --      machine is using the transmit buffer.
663 --
664 -----
665   when sttIdle =>
666     TBE <= '1';
667     tClkRST <= '0';
668     tfIncr <= '0';
669     shift <= '0';
670     load <= '0';
671     tDelayRst <= '1';
672 
673     if WR = '0' then
674       sttNext <= sttIdle;
675     else
676       sttNext <= sttTransfer;
677     end if;
678 -----
679 --
680 --Title: sttTransfer state
681 --
682 --Description: This state sets the load, tClkRST, and tDelayRst signals

```

```

683 --      high, while setting the TBE signal low. The load signal
684 --      is set high to load the transfer shift register with the
685 --      appropriate data, while the tClkRST and tDelayRst signals
686 --      are strobed to reset the tfCtr and tDelayCtr. The next
687 --      state loaded is the sttDelay state.
688 --
689 -----
690     when sttTransfer =>
691         TBE <= '0';
692         shift <= '0';
693         load <= '1';
694         tClkRST <= '1';
695         tfIncr <= '0';
696         tDelayRst <= '1';
697
698         sttNext <= sttDelay;
699 -----
700 --
701 --Title: sttShift state
702 --
703 --Description: This state strobes the shift and tfIncr signals high, and
704 --      checks the tfCtr to see if enough data has been
705 --      transmitted. By strobing the shift and tfIncr signals
706 --      high, the tfSReg is shifted, and the tfCtr is incremented
707 --      once. If tfCtr does not equal 9 (1001), then not all of
708 --      the bits have been transmitted, so the next state loaded
709 --      is the sttDelay state. If tfCtr does equal 9, the final
710 --      state, sttWaitWrite, is loaded.
711 --
712 -----
713     when sttShift =>
714         TBE <= '0';
715         shift <= '1';
716         load <= '0';
717         tfIncr <= '1';
718         tClkRST <= '0';
719         tDelayRst <= '0';
720
721         if tfCtr = "1010" then
722             sttNext <= sttWaitWrite;
723         else
724             sttNext <= sttDelay;
725         end if;
726 -----
727 --
728 --Title: sttDelay state
729 --
730 --Description: This state is responsible for delaying the transfer state
731 --      machine between transmissions. All signals are held low
732 --      while the tDelayCtr is tested. Once tDelayCtr is equal
733 --      to baudRate, the sttShift state is loaded.
734 --
735 -----
736     when sttDelay =>
737         TBE <= '0';
738         shift <= '0';
739         load <= '0';
740         tClkRst <= '0';
741         tfIncr <= '0';
742         tDelayRst <= '0';

```

```

743      if tDelayCtr = baudRate then
744          sttNext <= sttShift;
745      else
746          sttNext <= sttDelay;
747      end if;
748
749 -----
750 --
751 -- Title: sttWaitWrite state
752 --
753 -- Description: This state checks to make sure that the initial WR signal
754 --                 that triggered the transfer state machine has been
755 --                 brought back low. Without this state, a write signal
756 --                 that is held high for a long time will result in multiple
757 --                 transmissions. Once the WR signal is low, the sttIdle
758 --                 state is loaded to reset the transfer state machine.
759 --
760 -----
761 when sttWaitWrite =>
762     TBE <= '0';
763     shift <= '0';
764     load <= '0';
765     tClkRst <= '0';
766     tfIncr <= '0';
767     tDelayRst <= '0';
768
769     if WR = '1' then
770         sttNext <= sttWaitWrite;
771     else
772         sttNext <= sttIdle;
773     end if;
774 end case;
775 end process;
776 end Behavioral;

```

Nodo trasmettitore

control_unit

Il componente è stato sviluppato in modo comportamentale ed implementa l'automa descritto negli schematici attraverso il costrutto process ed il costrutto "CASE...WHEN". Di seguito è presente il codice VHDL del componente.

```

1
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4
5 entity control_unit is port (
6     reset        : in  std_logic;
7     wr_cleared   : in  std_logic;
8     clk          : in  std_logic;
9     TBE          : in  std_logic;
10    WR           : out std_logic
11 );

```

```

12 end control_unit;
13
14 architecture Behavioral of control_unit is
15
16 type stato is (idle, inizio_comunicazione, fine_comunicazione);
17 signal curr_state : stato := idle;
18 signal next_state : stato := idle;
19
20 begin
21
22 registri      : process (clk)
23 begin
24 if rising_edge(clk) then
25
26   if reset='1' then
27     curr_state <= idle;
28   else
29     curr_state  <= next_state;
30   end if;
31 end if;
32 end process;
33
34
35 calcolo_stato : process(curr_state, next_state, wr_cleared, TBE)
36 begin
37 case curr_state is
38   when idle           =>
39             WR <= '0';
40             if wr_cleared = '1',      THEN
41               next_state <= inizio_comunicazione;
42             end if;
43   when inizio_comunicazione =>
44             WR <= '1';
45             if TBE = '0',      THEN
46               next_state <= fine_comunicazione;
47             end if;
48   when fine_comunicazione =>
49             WR <= '0';
50             if TBE = '1',      THEN
51               next_state <= idle;
52             end if;
53 end case;
54 end process;
55
56 end Behavioral;

```

trasmettitore

Il componente é stato sviluppato in modo strutturale ed é utilizzato per formare il nodo del trasmettitore, infatti al suo interno viene collegata la control unit con il componente "rs232". Di seguito é presente il codice del trasmettitore.

```

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

```

```

4 entity trasmittitore is port (
5     reset          : in  std_logic;                      -- reset
6     clk            : in  std_logic;                      -- clock
7     wr_button      : in  std_logic;                      -- bottone di avvio
8     transmissione : in  std_logic_vector (7 downto 0);   -- input dall'esterno che si
9     vuole comunicare
10    output         : out std_logic := '1';              -- segnale di uscita dell'
11    uart
12 );
13 end trasmittitore;
14
15 architecture structural of trasmittitore is
16
17 component UARTcomponent is
18     Generic (
19         --@48MHz
20         BAUD_DIVIDE_G : integer := 26;  --115200 baud
21         BAUD_RATE_G   : integer := 417
22
23         --@26.6MHz
24         BAUD_DIVIDE_G : integer := 14;  --115200 baud
25         BAUD_RATE_G   : integer := 231
26     );
27     Port (
28         TXD    : out  std_logic := '1';           -- Transmitted serial data output
29         RXD    : in   std_logic;                  -- Received serial data input
30         CLK    : in   std_logic;                  -- Clock signal
31         DBIN   : in  std_logic_vector (7 downto 0);  -- Input parallel data to be transmitted
32         DBOUT  : out  std_logic_vector (7 downto 0);  -- Recevived parallel data output
33         RDA    : inout std_logic;                 -- Read Data Available
34         TBE    : out  std_logic := '1';           -- Transfer Buffer Emty
35         RD     : in   std_logic;                  -- Read Strobe
36         WR     : in   std_logic;                  -- Write Strobe
37         PE     : out  std_logic;                  -- Parity error
38         FE     : out  std_logic;                  -- Frame error
39         OE     : out  std_logic;                  -- Overwrite error
40         RST    : in   std_logic := '0');          -- Reset signal
41
42
43 component control_unit is port (
44     reset          : in  std_logic;
45     wr_cleared    : in  std_logic;
46     clk            : in  std_logic;
47     TBE            : in  std_logic;
48     WR             : out std_logic
49 );
50 end component;
51
52
53
54 signal sig_dbin   : std_logic_vector(7 downto 0) := (others => '1');
55 signal sig_tbe    : std_logic;
56 signal sig_rd     : std_logic;
57 signal sig_wr     : std_logic;
58
59 begin
60

```

```

61 | UART      : UARTcomponent  port map(
62 |   CLK       => clk,
63 |   RST       => reset,
64 |   TXD       => output,
65 |   RXD       => '1',
66 |   DBIN      => input,
67 |   TBE       => sig_tbe,
68 |   RD        => '1',
69 |   WR        => sig_wr
70 );
71 |
72 | controllo : control_unit port map(
73 |   reset     => reset,
74 |   wr_cleared => wr_button,
75 |   clk       => clk,
76 |   TBE       => sig_tbe,
77 |   WR        => sig_wr,
78 );

```

Nodo Ricevitore

uc_receiver

Il componente uc_receiver é stato sviluppato in modo comportamentale ed implementa in VHDL l'automa specificato negli schematici. I costrutti utilizzati sono: il costrutto process ed il costrutto "CASE...WHEN". Di seguito é presente il codice del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity cu_ricevitore is port(
5   CLK      : in std_logic;
6   RESET    : in std_logic;
7   RDA      : in std_logic;
8   UART_OUTPUT : in std_logic_vector (7 downto 0);
9   uart_reset : out std_logic;
10  leds      : out std_logic_vector (7 downto 0)
11 );
12 end cu_ricevitore;
13
14 architecture Behavioral of cu_ricevitore is
15
16 type stato is (reset_state, attesa_dati, lettura_dati, conferma_lettura);
17 signal leds_temp : std_logic_vector (7 downto 0) := (OTHERS=>'0');
18 signal reset_temp : std_logic := '0';
19 signal curr_state : stato := attesa_dati;
20 signal next_state : stato := attesa_dati;
21
22 begin
23
24 registri : process (clk)
25 begin
26 if rising_edge(clk) then
27

```

```

28     if reset='1' then
29         curr_state <= reset_state;
30     else
31         curr_state  <= next_state;
32         leds        <= leds_temp;
33         uart_reset <= reset_temp;
34     end if;
35
36 end if;
37 end process;
38
39
40 calcolo_stato : process(curr_state, next_state, RDA)
41 begin
42 case curr_state is
43     when reset_state =>
44         leds_temp <= (others=>'0');
45         reset_temp    <= '0';
46         next_state <= attesa_dati;
47     when attesa_dati =>
48         reset_temp <= '0';
49         if RDA = '1' then
50             next_state<=lettura_dati;
51         elsif RDA = '0' then
52             next_state<= attesa_dati;
53         end if;
54
55     when lettura_dati =>
56         reset_temp <= '0';
57         leds_temp <=UART_OUTPUT;
58         next_state<=conferma_lettura;
59
60     when conferma_lettura =>
61
62         reset_temp <= '1';
63         if RDA = '1' then
64             next_state<=conferma_lettura;
65         elsif RDA = '0' then
66             next_state<= attesa_dati;
67         end if;
68 end case;
69 end process;
70
71
72
73 end Behavioral;

```

Receiver

Il componente é stato sviluppato in modo strutturale ed é utilizzato per formare il nodo del ricevitore, infatti al suo interno viene collegata la control unit con il componente "rs232". Di seguito é presente il codice del Receiver.

```

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

```

```

3
4 entity ricevitore is port (
5   CLK      : in std_logic;
6   RST      : in std_logic;
7   UART_RXD : in std_logic;
8   LEDS     : out std_logic_vector (7 downto 0);
9   errors   : out std_logic_vector (2 downto 0)
10 );
11 end ricevitore;
12
13 architecture Behavioral of ricevitore is
14
15 component cu_ricevitore is port(
16   CLK      : in std_logic;
17   RESET    : in std_logic;
18   RDA      : in std_logic;
19   UART_OUTPUT : in std_logic_vector (7 downto 0);
20   uart_reset : out std_logic;
21   leds     : out std_logic_vector (7 downto 0)
22 );
23 end component;
24
25
26 component UARTcomponent is
27   Generic (
28     --@48MHz
29     BAUD_DIVIDE_G : integer := 26;  --115200 baud
30     BAUD_RATE_G   : integer := 417
31
32     --@26.6MHz
33     BAUD_DIVIDE_G : integer := 14;  --115200 baud
34     BAUD_RATE_G   : integer := 231
35   );
36   Port (
37     TXD      : out std_logic  := '1';      -- Transmitted serial data output
38     RXD      : in  std_logic;                -- Received serial data input
39     CLK      : in  std_logic;                -- Clock signal
40     DBIN     : in  std_logic_vector (7 downto 0);  -- Input parallel data to be transmitted
41     DBOUT    : out std_logic_vector (7 downto 0);  -- Recevived parallel data output
42     RDA      : inout std_logic;              -- Read Data Available
43     TBE      : out std_logic  := '1';      -- Transfer Buffer Emty
44     RD       : in  std_logic;                -- Read Strobe
45     WR       : in  std_logic;                -- Write Strobe
46     PE       : out std_logic;                -- Parity error
47     FE       : out std_logic;                -- Frame error
48     OE       : out std_logic;                -- Overwrite error
49     RST      : in  std_logic  := '0');     -- Reset signal
50
51 end component;
52
53
54 signal uart_reset      : std_logic;
55 signal uart_rda        : std_logic;
56 signal leds_conn       : std_logic_vector (7 downto 0);
57
58 begin
59
60   UART_R  : UARTcomponent port map(
61     RDA      => uart_rda,
62     RXD      => UART_RXD,

```

```

63      RST      => uart_reset,
64      CLK      => CLK,
65      RD       => '0',
66      DBOUT    => leds_conn,
67      DBIN     => (others =>'0'),
68      WR       => '0',
69      PE       => errors(0),
70      FE       => errors(1),
71      OE       => errors(2)
72 );
73
74
75 cu      : cu_ricevitore port map(
76   CLK          => CLK,
77   RESET        => RST,
78   RDA          => uart_rda,
79   UART_OUTPUT  => leds_conn,
80   uart_reset   => uart_reset,
81   leds         => LEDS
82 );

```

Top_entity

buttondebouncer

Il componente é utilizzato per pulire l'input del bottone start, il codice é lo stesso del button debouncer sviluppato nel capitolo 3 nel paragrafo: 3.2.1.

comunicazione_seriale

Il componente é stato sviluppato in modo strutturale ed é utilizzato per gestire i collegamenti tra il debouncer, il trasmettitore ed il ricevitore. Di seguito é presente il codice del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity comunicazione_seriale is port(
5   CLK          : in std_logic;
6   RST          : in std_logic;
7   wr_button    : in std_logic;
8   input         : in std_logic_vector (7 downto 0);
9   LEDS         : out std_logic_vector (7 downto 0);
10  errors       : out std_logic_vector (2 downto 0)
11 );
12 end comunicazione_seriale;
13
14 architecture Behavioral of comunicazione_seriale is
15
16 component trasmettitore is port (
17   reset        : in std_logic;                      -- reset
18   clk          : in std_logic;                      -- clock

```

```

19      wr_button      :  in  std_logic;                      -- bottone di avvio
20      trasmissione   :  in  std_logic_vector (7 downto 0);    -- input dall'esterno che si
21      vuole comunicare
22      output         :  out std_logic      := '1'          -- segnale di uscita
23      dell'uart
24 );
25
26 component ricevitore is port (
27     CLK      :  in  std_logic;
28     RST      :  in  std_logic;
29     UART_RXD :  in  std_logic;
30     LEDS     :  out std_logic_vector (7 downto 0);
31     errors   :  out std_logic_vector (2 downto 0)
32 );
33 end component;
34
35
36 component ButtonDebouncer is
37     generic (
38         CLK_period: integer := 10;  -- periodo del clock in nanosec
39         btn_noise_time: integer := 10000000 --durata dell'oscillazione in nanosec
40     );
41     Port ( RST : in STD_LOGIC;
42             CLK : in STD_LOGIC;
43             BTN : in STD_LOGIC;
44             CLEARED_BTN : out STD_LOGIC);
45 end component;
46
47
48 signal connessione_seriale :  std_logic := '1';
49 signal wr_cleared        :  std_logic := '0';
50
51 begin
52
53 trasm  :  trasmittitore port map(
54     reset      => RST,
55     clk        => CLK,
56     wr_button  => wr_cleared,
57     input       => input,
58     output      => connessione_seriale
59 );
60
61 ricev  :  ricevitore port map(
62     CLK      => CLK,
63     RST      => RST,
64     UART_RXD => connessione_seriale,
65     LEDS     => LEDS,
66     errors   => errors
67 );
68
69
70 wr_clearer :  ButtonDebouncer port map(
71     RST      => RST,
72     CLK      => CLK,
73     BTN      => wr_button,
74     CLEARED_BTN => wr_cleared
75 );

```

76 | end Behavioral;

9.1.3 Implementazione su scheda

Per far funzionare il progetto sulla scheda è necessario settare gli switch sulla scheda, premere il bottone centrale ed il risultato della comunicazione è osservabile sui led superiori (infatti è possibile vedere i da una parte i leds illuminati con i dati ricevuti e dall'altra i led con gli stati di errore). Di seguito è presente il codice di constraint utilizzato.

```

1 ## This file is a general .xdc for the Nexys A7-50T
2 ## To use it in a project:
3 ## - uncomment the lines corresponding to used pins
4 ## - rename the used ports (in each line, after get_ports) according to the top level signal
5 ## names in the project
6
7 ## Clock signal
8 set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { CLK }];
9 IO_L12P_T1_MRCC_35 Sch=clk100mhz
10
11 ##Switches
12 set_property -dict { PACKAGE_PIN J15     IOSTANDARD LVCMOS33 } [get_ports { input[0] }];
13 IO_L24N_T3_RS0_15 Sch=sw[0]
14 set_property -dict { PACKAGE_PIN L16     IOSTANDARD LVCMOS33 } [get_ports { input[1] }];
15 IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
16 set_property -dict { PACKAGE_PIN M13     IOSTANDARD LVCMOS33 } [get_ports { input[2] }];
17 IO_L6N_T0_D08_VREF_14 Sch=sw[2]
18 set_property -dict { PACKAGE_PIN R15     IOSTANDARD LVCMOS33 } [get_ports { input[3] }];
19 IO_L13N_T2_MRCC_14 Sch=sw[3]
20 set_property -dict { PACKAGE_PIN R17     IOSTANDARD LVCMOS33 } [get_ports { input[4] }];
21 IO_L12N_T1_MRCC_14 Sch=sw[4]
22 set_property -dict { PACKAGE_PIN T18     IOSTANDARD LVCMOS33 } [get_ports { input[5] }];
23 IO_L7N_T1_D10_14 Sch=sw[5]
24 set_property -dict { PACKAGE_PIN U18     IOSTANDARD LVCMOS33 } [get_ports { input[6] }];
25 IO_L17N_T2_A13_D29_14 Sch=sw[6]
26 set_property -dict { PACKAGE_PIN R13     IOSTANDARD LVCMOS33 } [get_ports { input[7] }];
27 IO_L5N_T0_D07_14 Sch=sw[7]
28
29 ## LEDs
30 set_property -dict { PACKAGE_PIN H17     IOSTANDARD LVCMOS33 } [get_ports { LEDS[0] }];
31 IO_L18P_T2_A24_15 Sch=led[0]
32 set_property -dict { PACKAGE_PIN K15     IOSTANDARD LVCMOS33 } [get_ports { LEDS[1] }];
33 IO_L24P_T3_RS1_15 Sch=led[1]
34 set_property -dict { PACKAGE_PIN J13     IOSTANDARD LVCMOS33 } [get_ports { LEDS[2] }];
35 IO_L17N_T2_A25_15 Sch=led[2]
36 set_property -dict { PACKAGE_PIN N14     IOSTANDARD LVCMOS33 } [get_ports { LEDS[3] }];
37 IO_L8P_T1_D11_14 Sch=led[3]
38 set_property -dict { PACKAGE_PIN R18     IOSTANDARD LVCMOS33 } [get_ports { LEDS[4] }];
39 IO_L7P_T1_D09_14 Sch=led[4]
40 set_property -dict { PACKAGE_PIN V17     IOSTANDARD LVCMOS33 } [get_ports { LEDS[5] }];
41 IO_L18N_T2_A11_D27_14 Sch=led[5]
42 set_property -dict { PACKAGE_PIN U17     IOSTANDARD LVCMOS33 } [get_ports { LEDS[6] }];
43 IO_L17P_T2_A14_D30_14 Sch=led[6]
```

```

28 set_property -dict { PACKAGE_PIN U16      IOSTANDARD LVCMOS33 } [get_ports { LEDs[7] }]; #
29     IO_L18P_T2_A12_D28_14 Sch=led[7]
30
31 ##Buttons
31 set_property -dict { PACKAGE_PIN N17      IOSTANDARD LVCMOS33 } [get_ports { wr_button }]; #
32     IO_L9P_T1_DQS_14 Sch=btnc
32 set_property -dict { PACKAGE_PIN M18      IOSTANDARD LVCMOS33 } [get_ports { RST }]; #
32     IO_L4N_T0_D05_14 Sch=btnu

```

9.2 Modifica del progetto utilizzando una ROM

Traccia

come variante dell'esercizio 8.1, il sistema A invia al sistema B tramite l'interfaccia seriale N stringhe di 8 bit contenute all'interno di una memoria ROM. Le stringhe ricevute vengono memorizzate in una memoria locale a B. Il progetto deve prevedere che A utilizzi un componente contatore per scandire le N stringhe da inviare.

9.2.1 Schematici

In questa parte del progetto viene modificato il funzionamento della macchina, infatti ora non c'è più bisogno di prendere i dati dall'esterno (tramite switch), ma sono già presenti all'interno di una ROM, vengono inviati uno alla volta (ad ogni pressione del bottone di start) ed infine vengono salvati in una memoria e mostrati sui led della scheda a partire dall'uscita della memoria. Per lo svolgimento di questo esercizio viene mantenuta la stessa struttura precedente (con la sola aggiunta delle memorie e dei contatori per l'indirizzo), anche se vengono aggiunti dei nuovi stati agli automi per gestire le 2 memorie e gli indirizzi.

Nodo Trasmettitore

Counter_mod16

Il contatore modulo 16 è utilizzato come ingresso di indirizzo per la rom, il componente è stato sviluppato come nel paragrafo:5.1.1.

ROM

La rom contiene tutti i dati che vanno inviati attraverso il componente "rs232" (di conseguenza contiene informazioni su 8 bit) ed è formata da 16 locazioni di memoria. Il componente è stato sviluppato come nel paragrafo: 6.1.1.

control_unit

La control unit del trasmettitore é stata leggermente modificata rispetto a quella nel punto precedente, infatti ora é necessario tener conto di quale dato inviare. Di conseguenza é stato opportuno cambiare l'automa implementato, il nuovo automa ha i seguenti stati:

- IDLE, che é lo stato in cui si attende l'arrivo del segnale di inizio trasmissione;
- INIZIO_COMUNICAZIONE, che é la stato in cui si avverte il componente rs232 di voler trasmettere i dati in ingresso (ponendo WR=1), l'automa rimane in questo stato finché il componente non avvia la trasmissione (leggendo TBE = '0');
- FINE_COMUNICAZIONE, che é lo stato in cui si attende che finisca la trasmissione dei dati (TBE='1'), durante questo stato viene posto l'ingresso WR = '0';
- INCREMENTO, che é lo stato in cui si incrementa il contatore (in modo che si possa prelevare il nuovo dato dalla ROM).

L'automa é rappresentato nella figura:81,

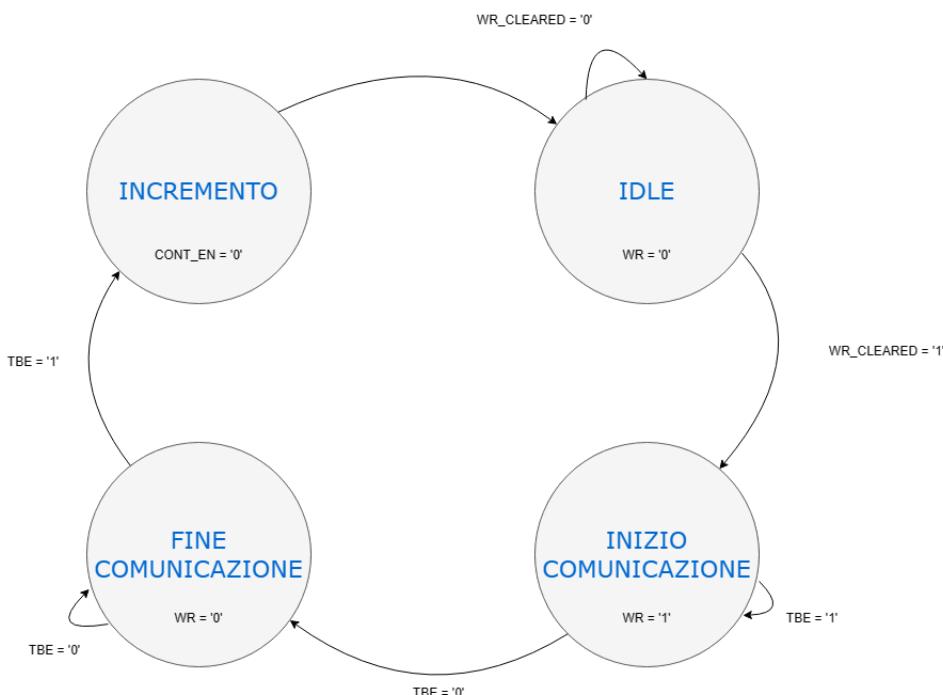


Figura 81: control unit trasmettitore

trasmettitore

Avendo aggiunto la ROM ed il contatore c'è stato un cambiamento del nodo trasmettitore, il componente con i nuovi collegamenti è presente nella figura: 82.

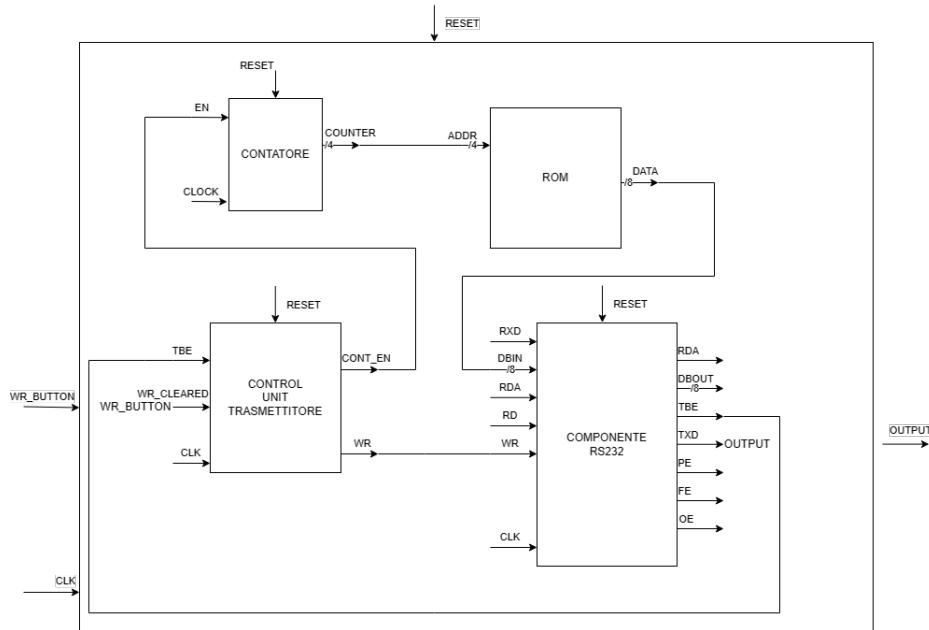


Figura 82: collegamenti nodo trasmettitore

Nodo Ricevitore

Memoria

Il componente è utilizzato per salvare il valore che è stato registrato dalla control unit e viene mostrato sulla scheda tramite i leds. La memoria ha 16 indirizzi disponibili e in ogni indirizzo di memoria è possibile salvare 8 bits. Il componente utilizzato ha lo stesso funzionamento del componente nel paragrafo: 6.1.1.

counter _16

Il contatore (come per il trasmettitore) viene utilizzato per dare l'indirizzo sia in scrittura che in lettura alla memoria (di conseguenza conta da 0 a 15 su 4 bit), il funzionamento del componente è presente al paragrafo: 5.1.1.

uc_receiver

La control unit del ricevitore é stata leggermente modificata rispetto a quella nel punto precedente, infatti ora é necessario tener conto della locazione di memoria nel quale salvare il dato e dell'incremento del contatore. Di conseguenza é stato opportuno cambiare l'automa implementato, il nuovo automa ha i seguenti stati:

- idle, dove si attende che il componente rs232 metta l'uscita RDA = '1' (avvisando della presenza di dati da leggere);
- lettura_dati, dove vengono salvati i valori interni al componente rs232 e vengono messi sui led in uscita dal componente;
- conferma_lettura, dove viene memorizzato il valore nella memoria, viene mostrato sui led (tramite il segnale read) e viene resettato il componente rs232.
- incremento, dove viene incrementato il contatore e si ritorna allo stato idle.
- reset_state, che é uno stato in cui si resetta il nodo del ricevitore.

L'automa é rappresentato nella figura: 83.

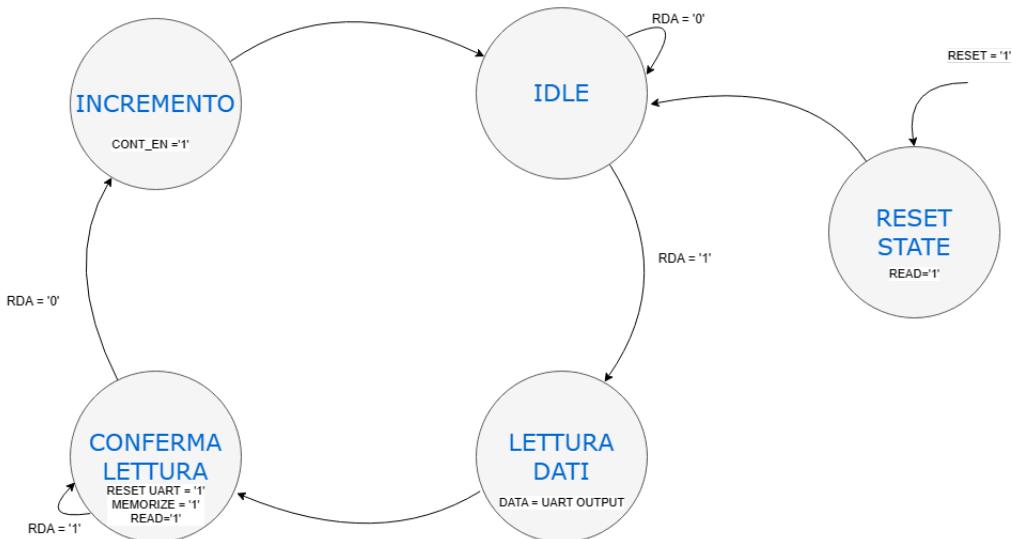


Figura 83: automa uc_receiver

Ricevitore

Avendo aggiunto la memoria ed il contatore c'é stato un cambiamento del nodo ricevitore, il componente con i nuovi collegamenti é presente nella figura: 84.

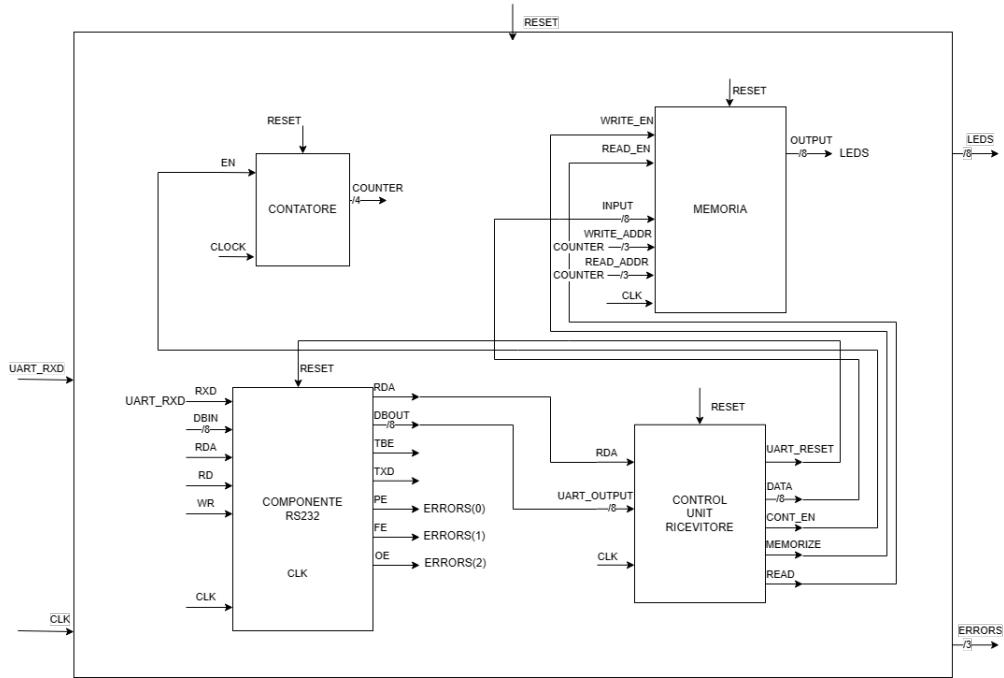


Figura 84: collegamenti nodo ricevitore

comunicazione_seriale

La struttura della macchina è cambiata solo sugli ingressi, infatti ora non è più necessario avere degli input dall'esterno per il valore da inviare da un nodo all'altro, ma solo il segnale di invio. I collegamenti tra le varie entità sono presenti in figura: 85

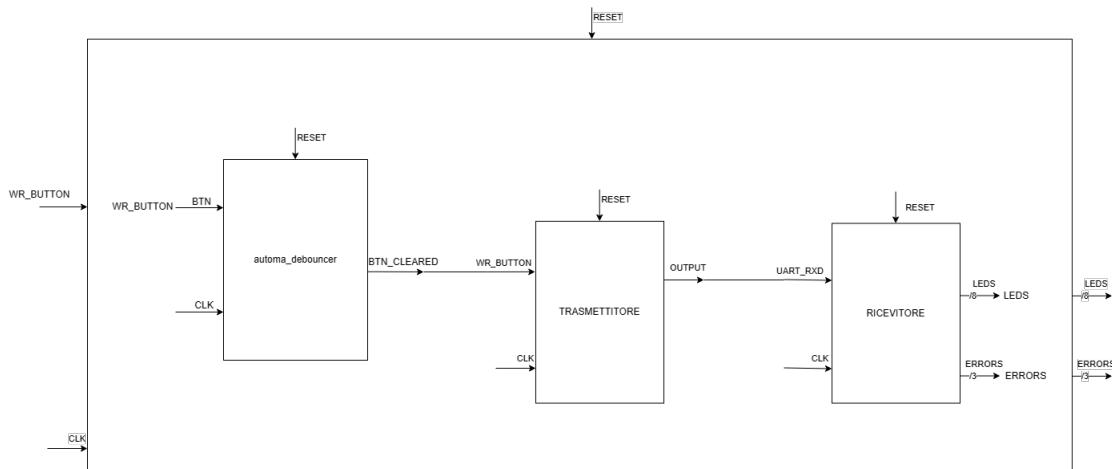


Figura 85: collegamenti comunicazione seriale

9.2.2 Codice VHDL

Nodo Trasmettitore

Counter_mod16

Il contatore modulo 16 é stato sviluppato come il contatore al paragrafo: 5.1.2. A differenza del contatore modulo 64, si hanno solo 4 bits in uscita e non é presente la funzione di preset (in quanto non era richiesto dalla traccia).

ROM

La ROM é stata sviluppata come il componente al paragrafo: 6.1.2. A differenza di quella ROM si hanno 8 bit di dati (che sono i bit di dati inviati dal trasmettitore)

control_unit

Come nel punto precedente la control unit del trasmettitore é stata sviluppata in modo comportamentale attraverso il costrutto process ed il costrutto "CASE...WHEN". Avendo gi specicato il funzionamento nel paragrafo precedente viene riportato solo il codice vhdl.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity uc_A is port (
5     reset          : in  std_logic;
6     wr_cleared    : in  std_logic;
7     clk            : in  std_logic;
8     TBE           : in  std_logic;
9     WR            : out std_logic;
10    uart_reset    : out std_logic;
11    cont_en       : out std_logic
12 );
13 end uc_A;
14
15 architecture Behavioral of uc_A is
16
17 type stato is (idle, Inizio_comunicazione, Fine_comunicazione, incremento);
18 signal curr_state : stato := idle;
19 signal next_state : stato := idle;
20
21 begin
22
23 registri      : process (clk)
24 begin
25 if rising_edge(clk) then
26
27     if reset='1' then
28         curr_state <= idle;
29     else
30         curr_state  <= next_state;

```

```

31      end if;
32  end if;
33 end process;
34
35
36 calcolo_stato : process(curr_state, next_state, wr_cleared, TBE)
37 begin
38 case curr_state is
39   when idle          =>
40     uart_reset <= '0';
41     WR <= '0';
42     cont_en <= '0';
43     if wr_cleared = '1'    THEN
44       next_state <= Inizio_comunicazione;
45     end if;
46   when Inizio_comunicazione  =>
47     WR <= '1';
48     uart_reset <= '0';
49     cont_en <= '0';
50     if TBE = '0'      THEN
51       next_state <= Fine_comunicazione;
52     end if;
53   when Fine_comunicazione  =>
54     uart_reset <= '0';
55     WR <= '0';
56     cont_en <= '0';
57     if TBE = '1'      THEN
58       next_state <= incremento;
59     end if;
60   when incremento        =>
61     uart_reset <= '0';
62     WR <= '0';
63     cont_en <= '1';
64     next_state <= idle;
65 end case;
66 end process;
67
68 end Behavioral;

```

trasmettitore

Il componente è stato descritto in modo strutturale e forma il nodo trasmettitore. Di seguito è presente il codice vhdl.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity trasmettitore is port (
5   reset           : in  std_logic;                      -- reset
6   clk             : in  std_logic;                      -- clock
7   wr_button       : in  std_logic;                      -- bottone di avvio
8   trasmmissione  : output std_logic := '1';           -- segnale di uscita dell'
9   uart
);
```

```

10 end trasmittitore;
11
12 architecture structural of trasmittitore is
13
14 component UARTcomponent is
15   Generic (
16     --@48MHz
17     BAUD_DIVIDE_G : integer := 26; --115200 baud
18     BAUD_RATE_G   : integer := 417
19
20     --@26.6MHz
21     BAUD_DIVIDE_G : integer := 14; --115200 baud
22     BAUD_RATE_G   : integer := 231
23 );
24 Port (
25   TXD    : out  std_logic  := '1';          -- Transmitted serial data output
26   RXD    : in   std_logic;                  -- Received serial data input
27   CLK    : in   std_logic;                  -- Clock signal
28   DBIN   : in   std_logic_vector(7 downto 0); -- Input parallel data to be transmitted
29   DBOUT  : out  std_logic_vector(7 downto 0); -- Received parallel data output
30   RDA    : inout std_logic;                -- Read Data Available
31   TBE    : out  std_logic  := '1';          -- Transfer Buffer Emty
32   RD     : in   std_logic;                  -- Read Strobe
33   WR     : in   std_logic;                  -- Write Strobe
34   PE     : out  std_logic;                  -- Parity error
35   FE     : out  std_logic;                  -- Frame error
36   OE     : out  std_logic;                  -- Overwrite error
37   RST    : in   std_logic  := '0');        -- Reset signal
38
39 end component;
40
41
42 component uc_A is port (
43   reset      : in  std_logic;
44   wr_cleared : in  std_logic;
45   clk        : in  std_logic;
46   TBE        : in  std_logic;
47   WR         : out std_logic;
48   uart_reset : out std_logic;
49   cont_en    : out std_logic
50 );
51 end component;
52
53 component counter_mod16 is port (
54   clock      : in  STD_LOGIC;
55   reset      : in  STD_LOGIC;
56   enable     : in  STD_LOGIC;
57   counter   : out STD_LOGIC_VECTOR(3 downto 0)
58 );
59 end component;
60
61 component ROM is port(
62   clk : in std_logic;
63   addr : in std_logic_vector(3 downto 0);
64   data : out std_logic_vector(7 downto 0)
65 );
66 end component;
67
68 --signal sig_dbin    : std_logic_vector(7 downto 0) := (others => '1');
69 signal sig_tbe     : std_logic;

```

```

70 signal sig_rd      : std_logic;
71 signal sig_wr      : std_logic;
72 signal sig_reset   : std_logic;
73 signal cont_en     : std_logic := '0';
74 signal addr        : std_logic_vector(3 downto 0) := (others => '0');
75 signal input        : std_logic_vector(7 downto 0);

76
77 begin
78
79 UART    : UARTcomponent port map(
80     CLK      => clk,
81     RST      => reset,
82     TXD      => output,
83     RXD      => '1',
84     DBIN     => input,
85     TBE      => sig_tbe,
86     RD       => '1',
87     WR       => sig_wr
88 );
89
90 controllo : uc_A port map(
91     reset      => reset,
92     wr_cleared => wr_button,
93     clk        => clk,
94     TBE        => sig_tbe,
95     WR         => sig_wr,
96     uart_reset => sig_reset,
97     cont_en    => cont_en
98 );
99
100 contatore : counter_mod16 port map (
101    clock      => clk,
102    reset      => reset,
103    enable     => cont_en,
104    counter   => addr
105 );
106
107 memoria_rom : ROM port map (
108    clk        => clk,
109    addr       => addr,
110    data       => input
111 );
112
113 end structural;

```

Nodo Ricevitore

Memoria

Il componente della memoria é stato sviluppato come nel paragrafo: 6.1.2. L'unica differenza consiste nel fatto che le locazioni sono di 8 bit, di conseguenza anche l'ingresso e l'uscita é su 8 bit.

uc_receiver

Come nel punto precedente la control unit del ricevitore è stata sviluppata in modo comportamentale attraverso il costrutto process ed il costrutto "CASE...WHEN". Avendo già specificato il funzionamento nel paragrafo precedente viene riportato solo il codice vhdl.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity cu_ricevitore is port(
5     CLK          : in  std_logic;
6     RESET        : in  std_logic;
7     RDA          : in  std_logic;
8     UART_OUTPUT : in  std_logic_vector (7 downto 0);
9     uart_reset   : out std_logic;
10    data         : out std_logic_vector (7 downto 0);
11    cont_en      : out std_logic;
12    memorize    : out std_logic;
13    read         : out std_logic
14 );
15 end cu_ricevitore;
16
17 architecture Behavioral of cu_ricevitore is
18
19 type stato is (reset_state, idle, lettura_dati, conferma_lettura, incremento);
20 signal data_temp    : std_logic_vector (7 downto 0) := (OTHERS=>'0');
21 signal reset_temp   : std_logic      := '0';
22 signal curr_state   : stato         := idle;
23 signal next_state   : stato         := idle;
24
25 begin
26
27 registri      : process (clk)
28 begin
29 if rising_edge(clk) then
30
31     if reset='1' then
32         curr_state <= reset_state;
33     else
34         curr_state <= next_state;
35         data       <= data_temp;
36         uart_reset <= reset_temp;
37     end if;
38
39 end if;
40 end process;
41
42
43 calcolo_stato : process(curr_state, next_state, RDA)
44 begin
45 case curr_state is
46     when reset_state =>
47             memorize <= '0';
48             read    <= '1';
49             cont_en <='0';
50             data_temp <= (others=>'0');
51             reset_temp <= '0';

```

```

52      next_state <= idle;
53      when idle =>
54
55          memorize <= '0';
56          read <= '0';
57          cont_en <='0';
58          reset_temp <= '0';
59          if RDA = '1' then
60              next_state<=lettura_dati;
61          elsif RDA = '0' then
62              next_state<= idle;
63          end if;
64
65      when lettura_dati =>
66
67          read <= '0';
68          cont_en <='0';
69          reset_temp <= '0';
70          data_temp <= UART_OUTPUT;
71          memorize <= '0';
72          next_state<=conferma_lettura;
73
74      when conferma_lettura =>
75
76          memorize <= '1';
77          cont_en <='0';
78          reset_temp <= '1';
79          read <= '1';
80          if RDA = '1' then
81              next_state<=conferma_lettura;
82          elsif RDA = '0' then
83              next_state<= incremento;
84          end if;
85      when incremento =>
86
87          read <= '0';
88          memorize <= '0';
89          cont_en <='1';
90          reset_temp <= '0';
91          next_state <= idle;
92
93 end case;
94 end process;
95
96
97
98 end Behavioral;

```

Ricevitore

Il componente é stato descritto in modo strutturale e forma il nodo ricevitore. Di seguito é presente il codice vhdl.

```

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

```

```

3
4 entity ricevitore is port (
5     CLK         : in std_logic;
6     RST         : in std_logic;
7     UART_RXD   : in std_logic;
8     LEDS        : out std_logic_vector (7 downto 0);
9     errors      : out std_logic_vector (2 downto 0)
10);
11 end ricevitore;
12
13 architecture Behavioral of ricevitore is
14
15 component cu_ricevitore is port(
16     CLK         : in std_logic;
17     RESET       : in std_logic;
18     RDA         : in std_logic;
19     UART_OUTPUT : in std_logic_vector (7 downto 0);
20     uart_reset  : out std_logic;
21     data        : out std_logic_vector (7 downto 0);
22     cont_en     : out std_logic;
23     memorize   : out std_logic;
24     read        : out std_logic
25 );
26 end component;
27
28
29 component UARTcomponent is
30     Generic (
31         --@48MHz
32         BAUD_DIVIDE_G : integer := 26;    --115200 baud
33         BAUD_RATE_G   : integer := 417
34
35         --@26.6MHz
36         BAUD_DIVIDE_G : integer := 14;    --115200 baud
37         BAUD_RATE_G   : integer := 231
38     );
39     Port (
40         TXD        : out std_logic := '1';          -- Transmitted serial data output
41         RXD        : in  std_logic;                  -- Received serial data input
42         CLK        : in  std_logic;                  -- Clock signal
43         DBIN       : in  std_logic_vector (7 downto 0); -- Input parallel data to be transmitted
44         DBOUT      : out std_logic_vector (7 downto 0); -- Received parallel data output
45         RDA        : inout std_logic;                -- Read Data Available
46         TBE        : out  std_logic := '1';          -- Transfer Buffer Emty
47         RD         : in  std_logic;                  -- Read Strobe
48         WR         : in  std_logic;                  -- Write Strobe
49         PE         : out  std_logic;                -- Parity error
50         FE         : out  std_logic;                -- Frame error
51         OE         : out  std_logic;                -- Overwrite error
52         RST        : in  std_logic := '0');          -- Reset signal
53
54 end component;
55
56 component memoria is port (
57     clock      : in std_logic;
58     reset      : in std_logic;
59     input       : in std_logic_vector(7 downto 0);
60     write_addr : in std_logic_vector(3 downto 0);
61     read_addr  : in std_logic_vector(3 downto 0);
62     write_en   : in std_logic;

```

```

63      read_en      : in std_logic;
64      output       : out std_logic_vector(7 downto 0)
65 );
66 end component;
67
68 component counter_mod16 is port (
69   clock      : in STD_LOGIC;
70   reset      : in STD_LOGIC;
71   enable     : in STD_LOGIC;
72   counter    : out STD_LOGIC_VECTOR (3 downto 0)
73 );
74 end component;
75
76 signal uart_reset  : std_logic;
77 signal uart_rda   : std_logic;
78 signal leds_conn   : std_logic_vector(7 downto 0);
79 signal data_conn   : std_logic_vector(7 downto 0);
80 signal cont_en    : std_logic;
81 signal write      : std_logic;
82 signal read       : std_logic;
83 signal addr       : std_logic_vector(3 downto 0);
84
85 begin
86
87 UART_R : UARTcomponent port map(
88   RDA      => uart_rda,
89   RXD      => UART_RXD,
90   RST      => uart_reset,
91   CLK      => CLK,
92   RD       => '0',
93   DBOUT   => leds_conn,
94   DBIN    => (others =>'0'),
95   WR       => '0',
96   PE       => errors(0),
97   FE       => errors(1),
98   OE       => errors(2)
99 );
100
101
102 cu      : cu_ricevitore port map(
103   CLK      => CLK,
104   RESET   => RST,
105   RDA      => uart_rda,
106   UART_OUTPUT => leds_conn,
107   uart_reset  => uart_reset,
108   data      => data_conn,
109   cont_en   => cont_en,
110   memorize  => write,
111   read      => read
112 );
113
114 memoria_dati : memoria port map (
115   clock      => CLK,
116   reset      => RST,
117   input       => data_conn,
118   write_addr  => addr,
119   read_addr   => addr,
120   write_en    => write,
121   read_en     => read,
122   output      => leds

```

```

123 );
124
125
126 contatore : counter_mod16 port map (
127     clock      => CLK,
128     reset      => RST,
129     enable     => cont_en,
130     counter   => addr
131 );
132
133 end Behavioral;

```

comunicazione_seriale

Il componente come nel paragrafo precedente è scritto in maniera strutturale e svolge la funzione di top entity del progetto, collegando sia inputs che outputs, ma anche il button debouncer, il trasmettitore ed il ricevitore. Di seguito è presente il codice del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity macchina is port(
5     CLK          : in  std_logic;
6     RST          : in  std_logic;
7     wr_button    : in  std_logic;
8     LEDS         : out std_logic_vector (7 downto 0);
9     errors       : out std_logic_vector (2 downto 0)
10 );
11 end macchina;
12
13 architecture Behavioral of macchina is
14
15 component trasmettitore is port (
16     reset        : in  std_logic;                                -- reset
17     clk          : in  std_logic;                                -- clock
18     wr_button    : in  std_logic;                                -- bottone di avvio
19     trasmissione : output std_logic := '1'                      -- segnale di uscita
20     dell'uart
21 );
22 end component;
23
24 component ricevitore is port (
25     CLK          : in  std_logic;
26     RST          : in  std_logic;
27     UART_RXD    : in  std_logic;
28     LEDS         : out std_logic_vector (7 downto 0);
29     errors       : out std_logic_vector (2 downto 0)
30 );
31 end component;
32
33
34 component ButtonDebouncer is
35     generic (

```

```

36      CLK_period: integer := 10;    -- periodo del clock in nanosec
37      btn_noise_time: integer := 10000000 --durata dell'oscillazione in nanosec
38  );
39  Port ( RST : in STD_LOGIC;
40          CLK : in STD_LOGIC;
41          BTN : in STD_LOGIC;
42          CLEARED_BTN : out STD_LOGIC);
43 end component;
44
45
46 signal connessione_seriale : std_logic := '1';
47 signal wr_cleared : std_logic := '0';
48
49 begin
50
51 trasm : trasmittitore port map(
52     reset      => RST,
53     clk        => CLK,
54     wr_button  => wr_cleared,
55     output      => connessione_seriale
56 );
57
58 ricev : ricevitore port map(
59     CLK         => CLK,
60     RST         => RST,
61     UART_RXD   => connessione_seriale,
62     LEDS        => LEDS,
63     errors      => errors
64 );
65
66
67 wr_clearer : ButtonDebouncer port map(
68     RST      => RST,
69     CLK      => CLK,
70     BTN      => wr_button,
71     CLEARED_BTN => wr_cleared
72 );
73 end Behavioral;

```

9.2.3 Implementazione su scheda

Per far funzionare l'esercizio sulla scheda è necessario solo premere il bottone centrale, infatti ad ogni pressione avverrà una trasmissione ed il risultato della trasmissione sarà presente sui led della scheda. Di seguito è presente il codice di constraints utilizzato.

```

1 ## Clock signal
2 set_property -dict { PACKAGE_PIN E3      IO_STANDARD LVCMOS33 } [get_ports { CLK }]; #
3           IO_L12P_T1_MRCC_35 Sch=clk100mhz
4 create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { CLK }];
5
6 ## LEDs
7 set_property -dict { PACKAGE_PIN H17      IO_STANDARD LVCMOS33 } [get_ports { LEDS[0] }]; #
8           IO_L18P_T2_A24_15 Sch=led[0]
9 set_property -dict { PACKAGE_PIN K15      IO_STANDARD LVCMOS33 } [get_ports { LEDS[1] }]; #
10          IO_L24P_T3_RS1_15 Sch=led[1]

```

```

8 | set_property -dict { PACKAGE_PIN J13      IOSTANDARD LVCMOS33 } [get_ports { LEDS[2] }]; #
9 |   IO_L17N_T2_A25_15 Sch=led[2]
10| set_property -dict { PACKAGE_PIN N14      IOSTANDARD LVCMOS33 } [get_ports { LEDS[3] }]; #
11|   IO_L8P_T1_D11_14 Sch=led[3]
12| set_property -dict { PACKAGE_PIN R18      IOSTANDARD LVCMOS33 } [get_ports { LEDS[4] }]; #
13|   IO_L7P_T1_D09_14 Sch=led[4]
14| set_property -dict { PACKAGE_PIN V17      IOSTANDARD LVCMOS33 } [get_ports { LEDS[5] }]; #
15|   IO_L18N_T2_A11_D27_14 Sch=led[5]
16| set_property -dict { PACKAGE_PIN U17      IOSTANDARD LVCMOS33 } [get_ports { LEDS[6] }]; #
17|   IO_L17P_T2_A14_D30_14 Sch=led[6]
18| ##Buttons
19| set_property -dict { PACKAGE_PIN N17      IOSTANDARD LVCMOS33 } [get_ports { wr_button }]; #
20|   IO_L9P_T1_DQS_14 Sch=btnc
21| set_property -dict { PACKAGE_PIN M18      IOSTANDARD LVCMOS33 } [get_ports { RST }]; #
22|   IO_L4N_T0_D05_14 Sch=btnu

```

10 Switch multistadio

Uno switch è un dispositivo in grado di mettere in comunicazione N sorgenti con M destinazioni. E' essenzialmente composto da un multiplexer la cui linea in uscita è connessa ad un demultiplexer, impostando opportunamente le selezioni dei due componenti si è in grado di mettere in comunicazione una sorgente con una destinazione. In figura 86 vi è un esempio con $N=M=4$.

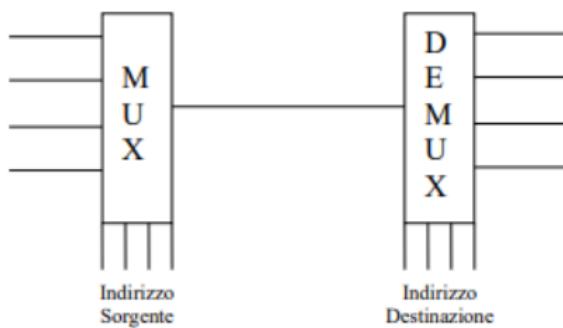


Figura 86: Switch con 4 sorgenti e 4 destinazioni

In questo tipo di comunicazione vi è però un inconveniente: si realizza una mutua esclusione tra i vari nodi in comunicazione, in quanto solo un collegamento alla volta può essere attivo (proprio per come sono realizzati mux e demux). Inoltre è complesso realizzare multiplexer e demultiplexer con molti ingressi/uscite, quindi questo schema di comunicazione non è scalabile su grandi dimensioni.

Per risolvere il problema della scalabilità si possono unire degli switch a singolo stadio in uno schema multistadio, il numero degli stadi varia in base al numero di sorgenti interconnesse, ma il principio alla base di questa struttura è un divide et impera, infatti componendo opportunamente gli switch a singolo stadio (che connettono due sorgenti a due destinazioni) si possono interconnettere grandi quantità di nodi.

Un esempio di schema di rete di connessione multistadio è l'Omega Network, che sfrutta la tecnica del perfect shuffling per connettere gli switch tra gli stadi. Questa tecnica fa riferimento al modo in cui le carte possono essere perfettamente mischiate, si divide il mazzo di due parti uguali e si mischiano le carte in modo che la carta i -esima di una metà sia accoppiata con la i -esima carta della seconda metà. Questo viene fatto per tutti gli stadi dell'Omega Network. Il numero di stadi di questo schema di interconnessione è derivato dal fatto che si utilizzano switch a connessione diretta (vedi figura 87), cioè a 2 ingressi e 2

uscite. Quindi il numero di stadi è $\log_2(N)$, con N il numero di sorgenti e numero di destinazioni, ogni stadio contiene $N/2$ switch. Utilizzando un'architettura di questo tipo si risolve

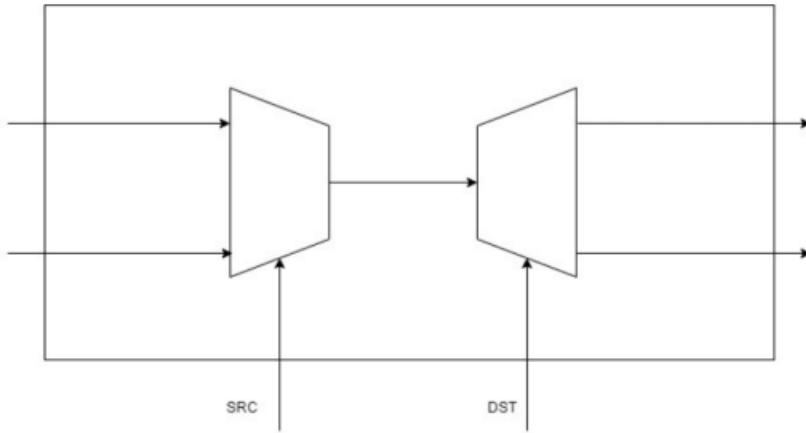


Figura 87: Switch a connessione diretta

il problema della difficile realizzabilità dello switch a singolo mux e demux, rimane però il problema della mutua esclusione tra i vari nodi in comunicazione, infatti, anche utilizzando l'Omega Network, un solo collegamento tra sorgente e destinazione può essere abilitato contemporaneamente. Una possibile soluzione al problema sarebbe permettere a più sorgenti di comunicare contemporaneamente, però bisognerebbe individuare un meccanismo efficace di gestione delle collisioni. Vi sono 4 meccanismi principali di gestione delle collisioni:

- **Tecnica bloccante:** è una tecnica conservativa e non efficiente, in quanto interrompiamo la comunicazione di una sorgente se un'altra comunicazione già è attiva. Questa tecnica può essere applicata anche con meccanismi a priorità per cui, se nello stesso istante più nodi vogliono comunicare, quello più prioritario è il primo ad utilizzare la rete. Questa soluzione non è efficiente perché blocchiamo anche le comunicazioni che non genererebbero collisioni sui nodi intermedi
- **Tecnica con perdite:** si abilitano più comunicazioni contemporanee, ma quando si verifica una collisione, uno dei due messaggi che collide viene eliminato. La rete derivante quindi è affetta da perdita di dati
- **Tecnica con reistradamento:** si re-instrada il messaggio lungo un percorso alternativo a quello bloccato. Questa eventualità di re-instradamento è possibile solo se le tabelle di routing non sono statiche e se la topologia della rete lo permette, nel caso dell'Omega Network questa tecnica non è applicabile

- Tecnica del cut-through: consiste nell'applicare la tecnica del store&forward. Quindi all'atto del conflitto un messaggio viene memorizzato e l'altro prosegue il suo percorso. Questa tecnica naturalmente può gestire un certo numero di conflitti di seguito, in quanto, una volta riempito il buffer in cui si memorizzano i messaggi, anche questa rete è soggetta a perdita di dati

10.1 Progettazione in VHDL con priorità fissa

Traccia

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 un rete con 4 nodi, implementando uno schema a priorità fissa fra i nodi (ed. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).

10.1.1 Schematici

Per la progettazione dello switch multistadio richiesto dalla traccia abbiamo deciso di utilizzare un tipo di messaggio particolare, basandoci sul modello dei pacchetti di rete realmente utilizzati abbiamo deciso di allegare a ciascun messaggio l'indirizzo sorgente e l'indirizzo destinazione del messaggio stesso. In figura 88 vediamo la struttura del pacchetto, dato che abbiamo 4 sorgenti e 4 destinazioni sono necessari due bit per gli indirizzi.

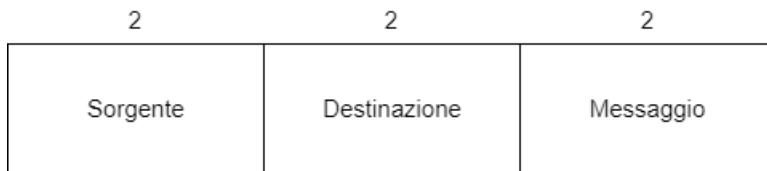


Figura 88: Pacchetto utilizzato

Definito il pacchetto possiamo passare a definire i componenti che saranno necessari. Abbiamo scomposto il problema in parte operativa e parte di controllo. Per la parte operativa dovremo realizzare lo switch a connessione diretta, quindi saranno necessari mux e demux, in particolare, dato che il messaggio sarà di 6 bit, i mux e demux dovranno poter trattare vettori di bit. Per la parte di controllo sarà necessario un arbitro logico per gestire lo schema a priorità fissa tra i nodi, quindi abilita la comunicazione per il nodo più prioritario.

Switch

Il mux utilizzato è un Mux[2:1] e il demux è un Demux[1:2] entrambi visti da nella sezione 1.1.1. Entrambi sono stati utilizzati per creare uno switch[2:2] in grado di lavorare con vettori di 6 bit. Di seguito lo schematico dello switch che mostra come sono collegati mux e demux.

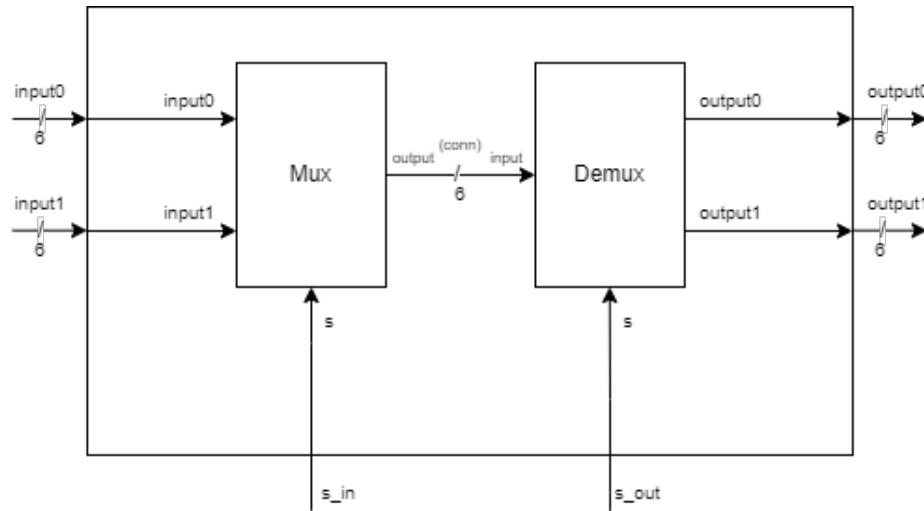


Figura 89: Switch[2:2] con vettori di 6 bit

Parte operativa: Omega Network

La parte operativa è l'effettiva Omega Network con 2 stadi di switch ciascuno contenente 2. I switch sono collegati con la tecnica del perfect shuffling. In figura 90 vi è lo schematico dell'intera Omega Network.

Arbitro logico

Per gestire le priorità abbiamo quindi progettato un arbitro logico che prende in ingresso le richieste di ciascun nodo di aprire una comunicazione e in uscita vi sarà solamente la richiesta più prioritaria tra quelle in ingresso. Abbiamo deciso che la priorità sarà decrescente a partire dalla sorgente 0 verso la sorgente 4. In figura 91 lo schematico del componente.

Parte di controllo

La parte di controllo ha come componente l'arbitro di controllo, ma anche gli ingressi. Infatti, in base alla scelta dell'arbitro logico, estrae indirizzo sorgente e destinazione dal pacchetto

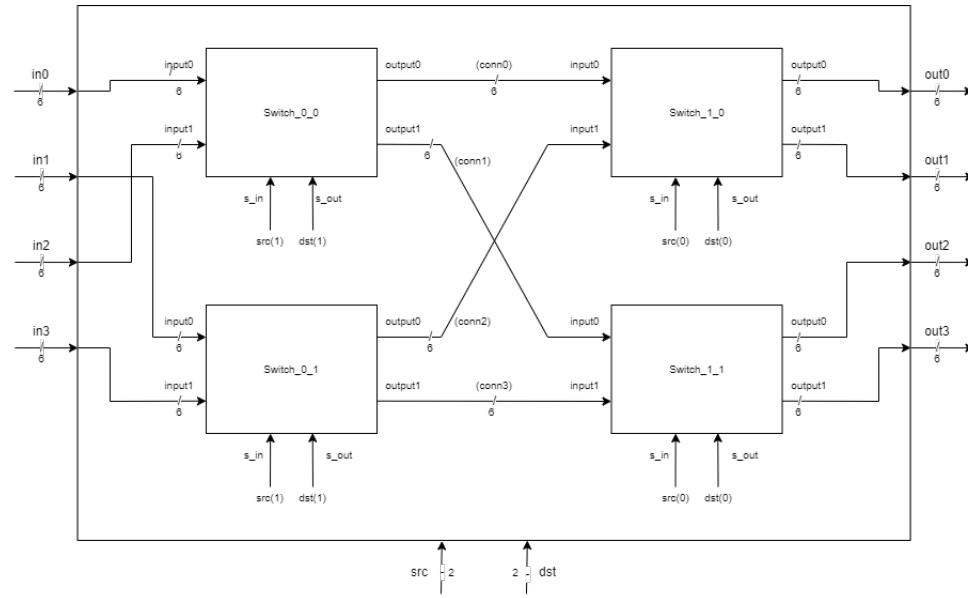


Figura 90: Omega Network con 4 ingressi e 4 uscite

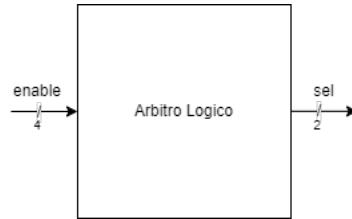


Figura 91: Arbitro logico

e seleziona un solo ingresso per fornirli all'Omega Network. In figura 92 lo schematico del componente.

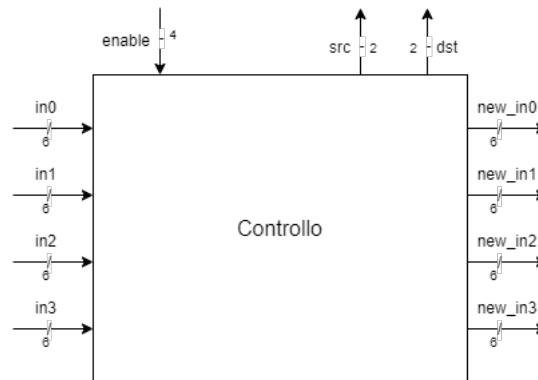


Figura 92: Parte di controllo dell'Omega Network

Architettura completa

Per l'architettura completa non resta che collegare parte operativa e parte di controllo per ottenere l'intera rete di connessione. In figura 93 lo schematico del componente.

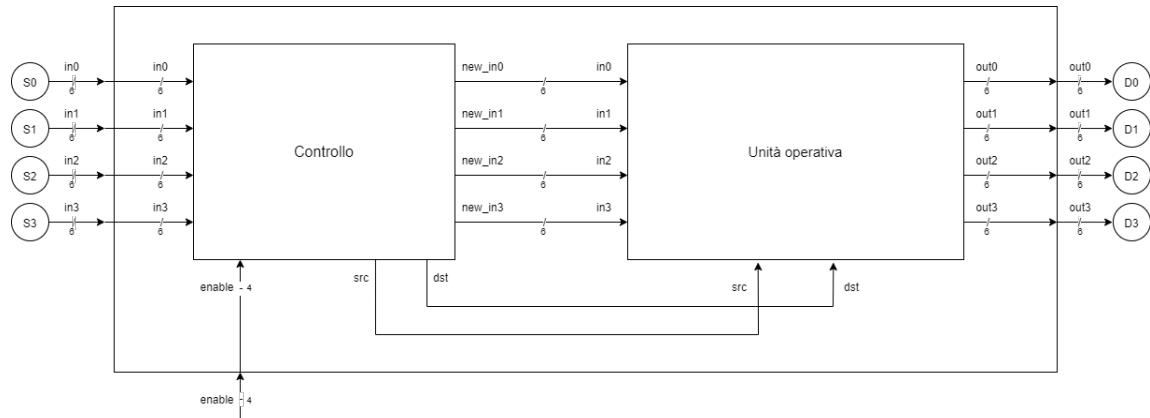


Figura 93: Architettura completa dell'Omega Network con nodi a priorità fissa

10.1.2 Codice VHDL

Passiamo ora a vedere il codice VHDL delle componenti di cui abbiamo visto gli schematici.

Switch

Il codice dello switch sfrutta semplicemente il paradigma di progettazione structural per unire il multiplexer al demultiplexer. Di seguito il codice.

```

1 entity switch_2_2 is port (
2   input0 : in std_logic_vector(5 downto 0);
3   input1 : in std_logic_vector(5 downto 0);
4   s_in   : in std_logic;
5   s_out  : in std_logic;
6   output0 : out std_logic_vector(5 downto 0);
7   output1 : out std_logic_vector(5 downto 0)
8 );
9 end switch_2_2;
10
11 architecture structural of switch_2_2 is
12 signal conn : std_logic_vector(5 downto 0);
13 begin
14 mux : entity work.mux_2_1 port map(
15   input0 => input0,
16   input1 => input1,
17   s      => s_in,
18   output  => conn
19 );
```

```

20
21 demux : entity work.demux_1_2 port map(
22     input    => conn,
23     s        => s_out,
24     output0 => output0,
25     output1 => output1
26 );
27 end structural;

```

Parte operativa

Per la parte operativa, quindi per costruire l'effettiva struttura dell'Omega Network, abbiamo usato la tecnica del perfect shuffling per collegare gli switch dei due stadi. È stato necessario usare il costrutto structural. Oltre gli ingressi e le uscite che si collegano a sorgenti e destinazioni, sono presenti anche gli ingressi per gli indirizzi di sorgente e destinazione. Di seguito il codice.

```

1 entity parte_operativa is port (
2     in0, in1, in2, in3      : in std_logic_vector(5 downto 0);
3     src, dst                : in std_logic_vector(1 downto 0);
4     out0, out1, out2, out3  : out std_logic_vector(5 downto 0)
5 );
6 end parte_operativa;
7
8 architecture structural of parte_operativa is
9 signal conn0 : std_logic_vector(5 downto 0);
10 signal conn1 : std_logic_vector(5 downto 0);
11 signal conn2 : std_logic_vector(5 downto 0);
12 signal conn3 : std_logic_vector(5 downto 0);
13 begin
14     switch_0_0 : entity work.switch_2_2 port map(
15         input0    => in0,
16         input1    => in2,
17         s_in      => src(1),
18         s_out     => dst(1),
19         output0   => conn0,
20         output1   => conn1
21 );
22     switch_0_1 : entity work.switch_2_2 port map(
23         input0    => in1,
24         input1    => in3,
25         s_in      => src(1),
26         s_out     => dst(1),
27         output0   => conn2,
28         output1   => conn3
29 );
30     switch_1_0 : entity work.switch_2_2 port map(
31         input0    => conn0,
32         input1    => conn2,
33         s_in      => src(0),
34         s_out     => dst(0),
35         output0   => out0,
36         output1   => out1

```

```

37 );
38 switch_1_1 : entity work.switch_2_2 port map(
39     input0  => conn1,
40     input1  => conn3,
41     s_in    => src(0),
42     s_out   => dst(0),
43     output0 => out2,
44     output1 => out3
45 );
46 end structural;

```

Arbitro logico

Per implementare l'arbitro logico abbiamo usato il costrutto when-else per fare in modo che l'ingresso più prioritario sia sempre il primo ad essere controllato, nel caso in cui non sia alto si passa al prossimo più prioritario. enable è il vettore delle richieste proveniente dalle varie sorgenti. Di seguito il codice.

```

1 entity arbitro_logico is port (
2     enable  : in std_logic_vector(3 downto 0);
3     sel      : out std_logic_vector(1 downto 0)
4 );
5 end arbitro_logico;
6
7 architecture dataflow of arbitro_logico is
8 begin
9
10 sel <= "00" when enable(0)='1' else
11     "01" when enable(1)='1' else
12     "10" when enable(2)='1' else
13     "11" when enable(3)='1' else
14     "--";
15
16 end dataflow;

```

Parte di controllo

La parte di controllo aggiunge lo slicing del pacchetto necessario per estrarre la sorgente e la destinazione. Naturalmente l'ingresso, la sorgente e la destinazione effettivi in input alla parte operativa sono relativi alla sorgente scelta dall'arbitro logico.

```

1 entity parte_di_controllo is port (
2     in0, in1, in2, in3          : in std_logic_vector(5 downto 0);
3     enable                      : in std_logic_vector(3 downto 0);
4     new_in0, new_in1, new_in2, new_in3 : out std_logic_vector(5 downto 0);
5     src, dst                     : out std_logic_vector(1 downto 0)
6 );
7 end parte_di_controllo;
8

```

```

9 architecture structural of parte_di_controllo is
10 signal sig_sel : std_logic_vector(1 downto 0);
11 begin
12 arbitro : entity work.arbitro_logico port map(
13     enable  => enable,
14     sel      => sig_sel
15 );
16 new_in0 <= in0 when sig_sel = "00" else (others=>'0');
17 new_in1 <= in1 when sig_sel = "01" else (others=>'0');
18 new_in2 <= in2 when sig_sel = "10" else (others=>'0');
19 new_in3 <= in3 when sig_sel = "11" else (others=>'0');
20
21 src <= in0(5 downto 4) when sig_sel="00" else
22     in1(5 downto 4) when sig_sel="01" else
23     in2(5 downto 4) when sig_sel="10" else
24     in3(5 downto 4) when sig_sel="11" else
25     "--";
26
27 dst <= in0(3 downto 2) when sig_sel="00" else
28     in1(3 downto 2) when sig_sel="01" else
29     in2(3 downto 2) when sig_sel="10" else
30     in3(3 downto 2) when sig_sel="11" else
31     "--";
32
33 end structural;

```

Architettura completa

L'Omega Network completa è solo l'unione di parte operativa e parte di controllo precedentemente viste. Di seguito il codice.

```

1 entity omega_network_4_4 is port (
2     in0, in1, in2, in3          : in std_logic_vector(5 downto 0);
3     enable                      : in std_logic_vector(3 downto 0);
4     out0, out1, out2, out3      : out std_logic_vector(5 downto 0)
5 );
6 end omega_network_4_4;
7
8 architecture structural of omega_network_4_4 is
9 signal sig_src                : std_logic_vector(1 downto 0);
10 signal sig_dst                : std_logic_vector(1 downto 0);
11 signal sig_in0, sig_in1, sig_in2, sig_in3  : std_logic_vector(5 downto 0);
12 begin
13
14 controllo : entity work.parte_di_controllo port map(
15     in0 => in0, in1 => in1, in2 => in2, in3 => in3,
16     enable => enable,
17     new_in0 => sig_in0, new_in1 => sig_in1, new_in2 => sig_in2, new_in3 => sig_in3,
18     src => sig_src, dst => sig_dst
19 );
20
21 operativa : entity work.parte_operativa port map(
22     in0 => sig_in0, in1 => sig_in1, in2 => sig_in2, in3 => sig_in3,
23     src => sig_src, dst => sig_dst,
24     out0 => out0, out1 => out1, out2 => out2, out3 => out3

```

```
25 );  
26  
27 end structural;
```

10.1.3 Simulazione

Per la simulazione della rete di interconnessione realizzata abbiamo definito due casi di test:

- Tutti i nodi vogliono parlare contemporaneamente, in questo caso sarà il nodo più prioritario (cioè il nodo 0) ad avere la precedenza
 - Il nodo prioritario interrompe la comunicazione, quindi il secondo più prioritario potrà accedere alla rete

Di seguito il codice che permette di testare questi comportamenti:

```
1  in0 <= "001011";  
2  in1 <= "011000";  
3  in2 <= "101001";  
4  in3 <= "111010";  
5  enable <= "1111";  
6  
7  wait for 20 ns;  
8  
9  in0 <= "001011";  
10 in1 <= "011000";  
11 in2 <= "101001";  
12 in3 <= "111010";  
13 enable <= "1010";
```

Questo codice fornisce effettivamente il risultato atteso, infatti inizialmente arriva il pacchetto associato al nodo 0, dopo 20 ns arriva invece il risultato del secondo nodo più prioritario che vuole comunicare, cioè il nodo 1. In figura 94 c'è la simulazione della rete.

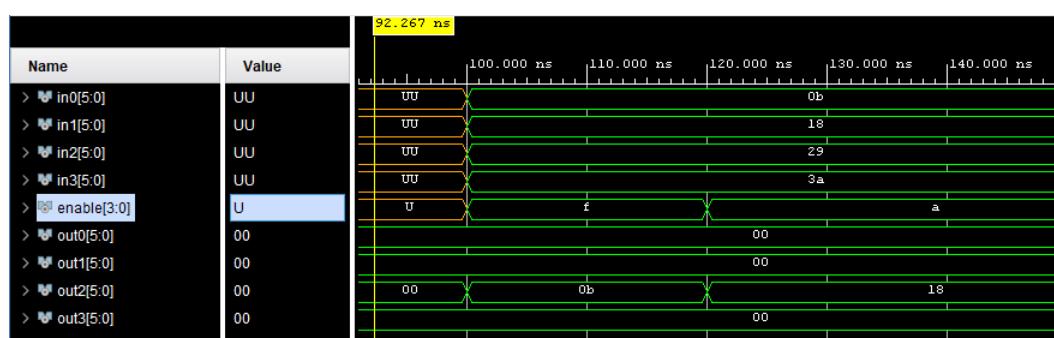


Figura 94: Simulazione dell'Omega Network con nodi a priorità fissa

10.2 Progettazione in VHDL con gestione dei conflitti

Traccia

Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network. Rimuovendo l'ipotesi di lavorare secondo uno schema a priorità fra i nodi e considerando una rete di 8 nodi, lo switch deve gestire eventuali conflitti generati da collisioni secondo un meccanismo a scelta (ad es. perdendo uno dei messaggi in conflitto).

10.2.1 Schematici

L'Omega Network richiesta deve avere 8 sorgenti e 8 destinazioni, quindi dovranno esserci 3 stadi, ciascuno con 4 switch. Inoltre la traccia di questo esercizio prevede di rimuovere lo schema a priorità fissa tra i nodi e di gestire le collisioni sui nodi intermedi con un meccanismo a scelta.

Per gestire le collisioni abbiamo deciso di utilizzare la tecnica dello store & forward, quindi quando uno switch si accorge di una collisione, memorizza un messaggio e inoltra l'altro. Il messaggio memorizzato verrà trasmesso appena non si verificherà più una collisione in ingresso.

Per implementare la semantica dello store& forward nell'Omega Network è necessario aggiungere dell' "intelligenza" all'interno di tutti gli switch della rete. Quindi naturalmente complichiamo la struttura dello switch, avendo però il vantaggio di riuscire ad avere più comunicazioni attive contemporaneamente evitando, in genere, anche perdite di dati. Se si verifica una collisione quando la "memoria" dello switch è già piena uno dei messaggi deve essere scartato, quindi in casi estremi permane la perdita di informazione.

Quindi abbiamo modificato la struttura dello switch, aggiungendo delle componenti in grado di implementare il comportamento desiderato. Oltre il mux e il demux sono stati necessari un registro per memorizzare uno dei due pacchetti che collide, un'unità di controllo che riconosca le collisioni e gestisca la semantica del nodo e un ulteriore mux per gestire l'inserimento dei dati memorizzati in uscita.

Per questo esercizio è stato necessario modificare anche il pacchetto che viene trasmesso. Prima di tutto gli indirizzi di sorgente e destinazione devono essere da 3 bit, perché adesso i nodi sono 8. Inoltre, per permettere al nodo di riconoscere le collisioni, abbiamo inserito un bit a monte del pacchetto che, se alto, identifica un pacchetto effettivamente presente sulla connessione, mentre, se basso, indica che la connessione non è utilizzata e quindi non vi è alcun pacchetto trasmesso. Quindi una nostra ipotesi di progetto è che la sorgente

tenga alto il pacchetto sulla linea di input per solo un periodo di clock dell'omega network, a meno che non vogliano trasmettere più volte lo stesso pacchetto. Infatti la comunicazione non è utilizzata solo se, in ingresso, il bit più significativo è zero, mentre se è alto allora la sorgente sta effettivamente cercando di inviare un pacchetto. In figura 95 la nuova struttura del pacchetto.

1	3	3	2
0/1	Sorgente	Destinazione	Messaggio

Figura 95: Pacchetto

Vediamo ora i vari componenti che introduciamo nel nodo.

Registro

Il registro necessario allo switch deve prevedere un inserimento in parallelo in presenza di un segnale di load. Naturalmente ha 9 bit in ingresso e 9 bit in uscita, oltre ai segnali di clock e reset tipici di una macchina sequenziale. In figura 96 lo schematico del componente.

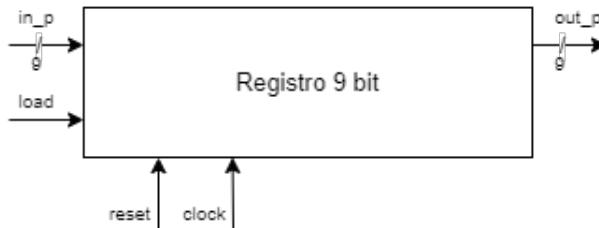


Figura 96: Registro da 9 bit

Unità di controllo

L'unità di controllo deve gestire sia la selezione dei multiplexer che l'eventuale memorizzazione di un messaggio in collisione. Quindi in ingresso ha: i due ingressi dello switch e l'uscita del registro per gestire la selezione del mux e demux facendo uno slicing del pacchetto; le uscite che si collegano alle selezioni dei mux e demux; il segnale di load del registro e un vettore di 9 bit in input al registro. In figura 97 lo schematico del componente e l'automa implementato dall'unità di controllo. In particolare notiamo che riempiamo il registro quando

si verifica una collisione, cioè quando ho due pacchetti validi in ingresso (bit più significativo alto), mentre lo svuotiamo quando quando in ingresso non ho nessun pacchetto.

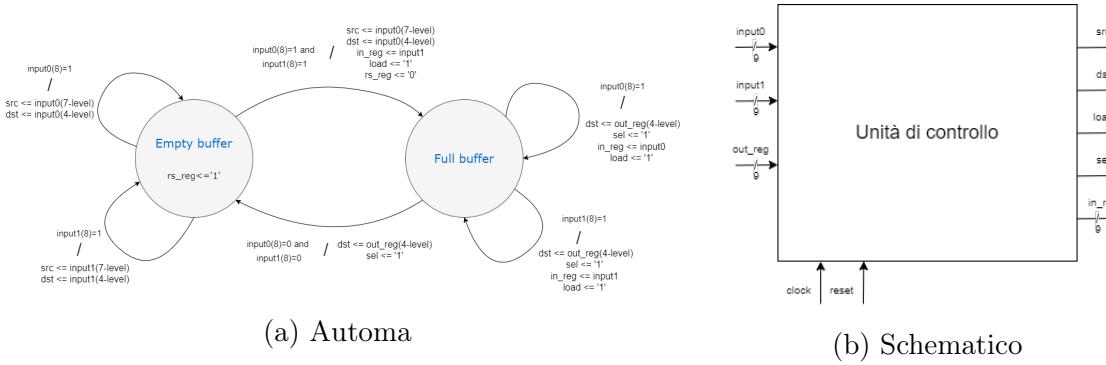


Figura 97: Automa e schematico dell'unità di controllo

Smart Switch

A questo punto non resta che collegare i componenti per creare lo switch. Il comportamento implementato prevede di, in presenza di una collisione, memorizzare il secondo input e inoltrare il secondo. Quando il buffer è pieno ed in ingresso vi è un pacchetto allora memorizza il pacchetto in ingresso e inoltra quello memorizzato, in modo da evitare lo stallo di un pacchetto nello switch. Nel caso in cui il buffer sia pieno e si verifica un'altra collisione in ingresso allora il secondo messaggio in input viene scartato e quindi si ha una perdita di dati. Nel caso di buffer vuoto e solo un messaggio in ingresso lo switch funziona normalmente inoltrandolo. In figura 98 lo schematico del componente. Notiamo come, per inoltrare il pacchetto memorizzato, sia necessario utilizzare un secondo multiplexer per collegare, al demultiplexer, l'uscita del registro invece dell'uscita del primo multiplexer. La logica di questa operazione viene pilotata dall'unità di controllo.

Omega Network

Per terminare la rete di interconnessione basta ora collegare gli smart switch creati nel paragrafo precedente per realizzare l'Omega Network. In particolare è necessario il clock ed il reset per tutti gli switch e i collegamenti interni sono da fare seguendo la regola del perfect shuffling. In figura 99 lo schematico del componente.

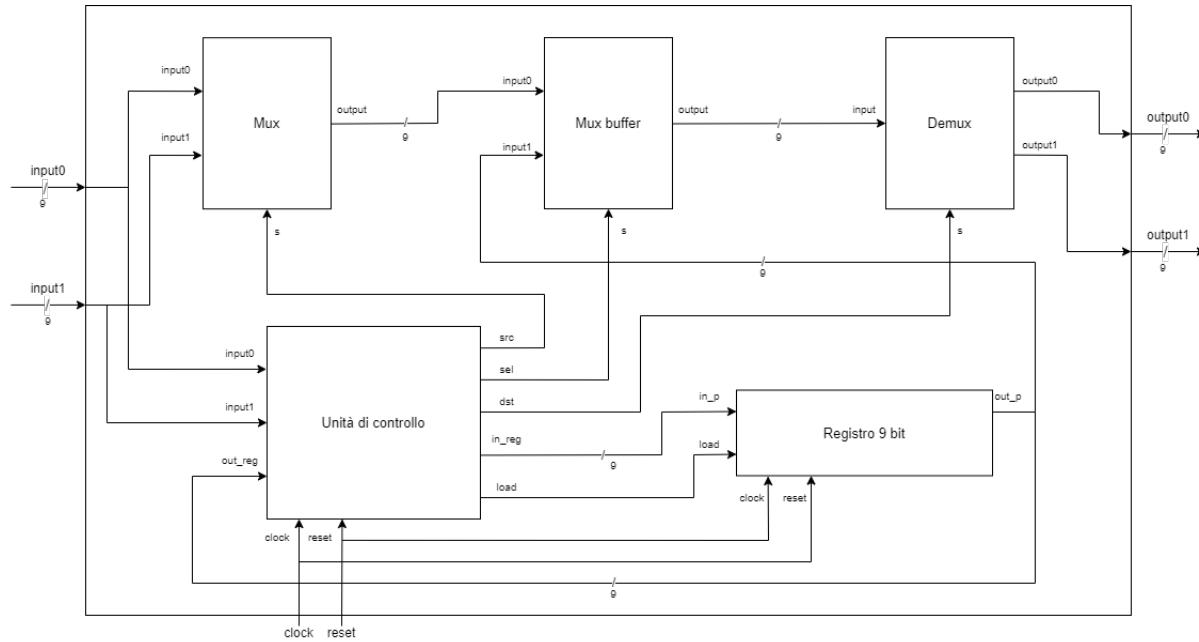


Figura 98: Smart switch

10.2.2 Codice VHDL

Passiamo ora in rassegna il codice VHDL che implementa quanto visto nella sezione precedente.

Registro

L'implementazione del registro è stata fatta usando una progettazione comportamentale molto simile a quella vista nella sezione 4.1, solamente che il caricamento del registro è in parallelo ed è stata rimossa la funzione di shift che non è necessaria in questo caso.

```

1  entity register_9 is port (
2    clock      : in std_logic;
3    reset      : in std_logic;
4    load       : in std_logic;
5    in_p       : in std_logic_vector(8 downto 0);
6    out_p      : out std_logic_vector(8 downto 0)
7  );
8  end register_9;
9
10 architecture behavioural of register_9 is
11 signal tmp: std_logic_vector(8 downto 0) := (others=>'0');
12 begin
13 process (clock)
14 begin
15 if (rising_edge(clock)) then
16   if (reset='1') then

```

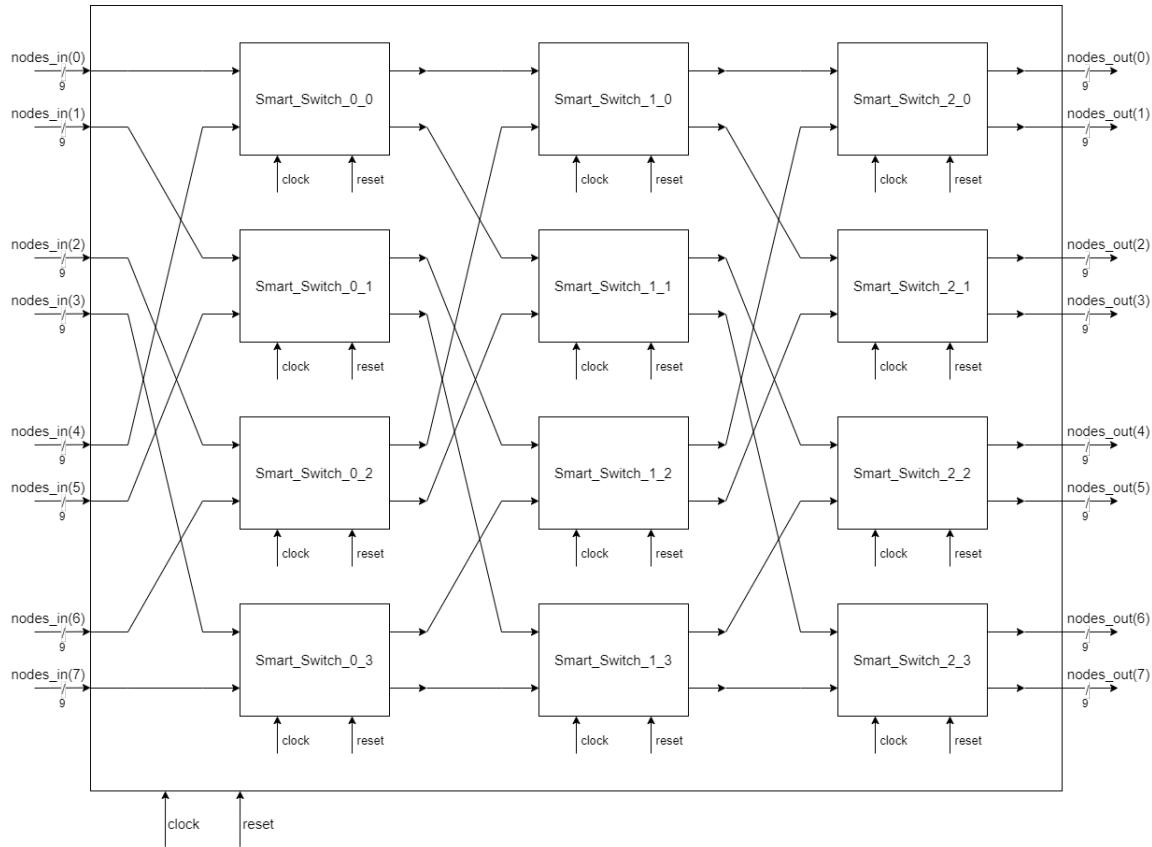


Figura 99: Omega Network "smart"

```

17      tmp <= (others=>'0');
18      elsif (load = '1') then
19          tmp <= in_p;
20      end if;
21  end if;
22  end process;
23  out_p <= tmp;
24  end behavioural;

```

Unità di controllo

Nella precedente sezione abbiamo presentato il funzionamento dell'unità di controllo allegando anche l'automa. L'implementazione è la classica a due process per implementare gli automi: un process combinatorio e un process che implementa il registro per il cambiamento di stato. Inoltre è da notare come ad ogni arco sia associata anche un'operazione su uno tra i pacchetti in ingresso o quello memorizzato. Le operazioni che vengono svolte sono quelle di slicing del pacchetto per impostare le selezioni dei multiplexer, la gestione di memorizzazione nel registro e l'inoltro del pacchetto memorizzato nel registro tramite selezione del secondo

multiplexer. Di seguito il codice. Notiamo come è stato necessario utilizzare il generic per associare all'unità di controllo lo stadio in cui si trova in quanto lo slicing da fare per ricavare le selezioni dei mux e demux dipende dallo stadio in cui si trova lo switch.

```

1 entity uc_switch is
2 generic (
3     level : integer := 0
4 );
5 port (
6     clock    : in std_logic;
7     reset    : in std_logic;
8     input0   : in std_logic_vector(8 downto 0);
9     input1   : in std_logic_vector(8 downto 0);
10    out_reg  : in std_logic_vector(8 downto 0);
11    src      : out std_logic;
12    dst      : out std_logic;
13    load     : out std_logic;
14    in_reg   : out std_logic_vector(8 downto 0);
15    sel      : out std_logic
16 );
17 end uc_switch;
18
19 architecture Behavioral of uc_switch is
20 type stato is (empty_buffer, full_buffer);
21 signal curr_state : stato := empty_buffer;
22 signal next_state : stato := empty_buffer;
23 begin
24
25 registri : process (clock)
26 begin
27 if rising_edge(clock) then
28
29     if reset='1' then
30         curr_state <= empty_buffer;
31     else
32         curr_state <= next_state;
33     end if;
34 end if;
35 end process;
36
37 calcolo_stato : process(curr_state, input0, input1)
38 begin
39 load <= '0';
40 sel <= '0';
41 in_reg <= (others => '0');
42 src <= '0';
43 dst <= '0';
44
45 case curr_state is
46     when empty_buffer      =>
47             rs_reg <= '1';
48             if (input0(8)='1' and input1(8)='1') then
49                 src <= input0(7-level);
50                 dst <= input0(4-level);
51                 in_reg <= input1;
52                 load <= '1';
53                 rs_reg <= '0';

```

```

54         next_state <= full_buffer;
55
56         elsif (input0(8)='1') then
57             src <= input0(7-level);
58             dst <= input0(4-level);
59             next_state <= empty_buffer;
60
61         elsif (input1(8)='1') then
62             src <= input1(7-level);
63             dst <= input1(4-level);
64             next_state <= empty_buffer;
65
66     end if;
67
68     when full_buffer      =>
69         if (input0(8)='1') then
70             dst <= out_reg(4-level);
71             sel <= '1';
72             in_reg <= input0;
73             load <= '1';
74             next_state <= full_buffer;
75
76         elsif (input1(8)='1') then
77             dst <= out_reg(4-level);
78             sel <= '1';
79             in_reg <= input1;
80             load <= '1';
81             next_state <= full_buffer;
82
83     else
84         dst <= out_reg(4-level);
85         sel <= '1';
86         next_state <= empty_buffer;
87
88     end if;
89
90 end case;
91 end process;
92
93 end Behavioral;
```

Smart Switch

Lo smart switch è realizzato facendo una semplice composizione dei componenti visti seguendo lo schematico visto nella sezione precedente. Di seguito il codice del componente. Notiamo come il generic dell'unità di controllo si propaghi anche allo smart switch in quanto lo stadio deve essere assegnato allo switch stesso.

```

1 entity smart_switch_2_2 is
2 generic (
3     level : integer := 0
4 );
5 port (
6     clock    : in std_logic;
```

```

7      reset    : in std_logic;
8      input0   : in std_logic_vector(8 downto 0);
9      input1   : in std_logic_vector(8 downto 0);
10     output0  : out std_logic_vector(8 downto 0);
11     output1  : out std_logic_vector(8 downto 0)
12 );
13 end smart_switch_2_2;
14
15 architecture structural of smart_switch_2_2 is
16 signal sig_mux        : std_logic_vector(8 downto 0);
17 signal sig_conn       : std_logic_vector(8 downto 0);
18 --signal sig_in0       : std_logic_vector(8 downto 0);
19 --signal sig_in1       : std_logic_vector(8 downto 0);
20 signal sig_output     : std_logic_vector(8 downto 0);
21 signal sig_mem_input  : std_logic_vector(8 downto 0);
22 signal sig_s_in       : std_logic;
23 signal sig_s_out      : std_logic;
24 signal sig_load       : std_logic;
25 signal sig_sel        : std_logic;
26 begin
27
28 control0 : entity work.uc_switch
29 generic map (
30 level => level
31 )
32 port map(
33     clock    => clock,
34     reset    => reset,
35     input0   => input0,
36     input1   => input1,
37     out_reg  => sig_output,
38     src      => sig_s_in,
39     dst      => sig_s_out,
40     load     => sig_load,
41     rs_reg   => sig_rs_reg,
42     in_reg   => sig_mem_input,
43     sel      => sig_sel
44 );
45
46 mux_buffer : entity work.mux_2_1 port map(
47     input0  => sig_mux,
48     input1  => sig_output,
49     s       => sig_sel,
50     output  => sig_conn
51 );
52
53 mux : entity work.mux_2_1 port map(
54     input0  => input0,
55     input1  => input1,
56     s       => sig_s_in,
57     output  => sig_mux
58 );
59
60 demux : entity work.demux_1_2 port map(
61     input    => sig_conn,
62     s       => sig_s_out,
63     output0 => output0,
64     output1 => output1
65 );
66

```

```

67 reg : entity work.register_9 port map(
68   clock      => clock,
69   reset      => reset or sig_rs_reg,
70   load       => sig_load,
71   in_p       => sig_mem_input,
72   out_p      => sig_output
73 );
74
75 end structural;

```

Omega Network

A questo punto l'Omega Network viene semplicemente realizzato componendo 4 switch per ogni stadio, quindi in totale 12 switch. Per velocizzare la scrittura abbiamo deciso di definire un nuovo tipo per definire un vettore di ingressi ed un vettore di uscite in modo da poter fare il mapping delle porte utilizzando dei for-generate. Per definire il nuovo tipo abbiamo definito un package che riportiamo di seguito.

```

1 package type_package is
2 type input_array is array (0 to 7) of std_logic_vector(8 downto 0);
3 end package type_package;
4
5 package body type_package is
6 end package body type_package;

```

In questo modo possiamo utilizzare il tipo input_array per definire un vettore di vettori di ingresso e usarlo nei for-generate.

Di seguito il codice dell'Omega Network. Utilizziamo un for-generate per ciascuno stadio in modo da poter importare correttamente il valore nel generic degli smart switch.

```

1 entity smart_omega_network is port (
2   clock      : in std_logic;
3   reset      : in std_logic;
4   nodes_in   : in input_array;
5   nodes_out  : out input_array
6 );
7 end smart_omega_network;
8
9 architecture structural of smart_omega_network is
10 signal first_conn  : input_array;
11 signal second_conn : input_array;
12 begin
13
14 first_lvl : for i in 0 to 3 generate
15   switch_0: entity work.smart_switch_2_2
16     generic map ( level => 0)
17     port map(
18       clock      => clock,
19       reset      => reset,

```

```

20      input0  => nodes_in(i),
21      input1  => nodes_in(4+i),
22      output0 => first_conn(2*i),
23      output1 => first_conn(2*i+1)
24  );
25 end generate;
26
27 second_lvl : for i in 0 to 3 generate
28     switch_0: entity work.smart_switch_2_2
29         generic map ( level => 1)
30         port map(
31             clock  => clock,
32             reset  => reset,
33             input0 => first_conn(i),
34             input1 => first_conn(4+i),
35             output0 => second_conn(2*i),
36             output1 => second_conn(2*i+1)
37         );
38 end generate;
39
40 thrid_lvl : for i in 0 to 3 generate
41     switch_0: entity work.smart_switch_2_2
42         generic map ( level => 2)
43         port map(
44             clock  => clock,
45             reset  => reset,
46             input0 => second_conn(i),
47             input1 => second_conn(4+i),
48             output0 => nodes_out(2*i),
49             output1 => nodes_out(2*i+1)
50         );
51 end generate;
52
53 end structural;
```

10.2.3 Simulazione

In simulazione abbiamo testato la gestione delle collisione degli smart switch cercando di capire se rispettassero le specifiche decise. Di seguito il codice della simulazione.

```

1 nodes_in(0) <= "100000001";
2 nodes_in(1) <= "100100011";
3 wait for clk_period;
4 nodes_in(0) <= "000000000";
5 nodes_in(1) <= "000000000";
6
7 wait for clk_period*2;
8
9 nodes_in(0) <= "100010000";
10 nodes_in(3) <= "101110010";
11 wait for clk_period;
12 nodes_in(0) <= "000000000";
13 nodes_in(3) <= "000000000";
14
15 wait for clk_period*2;
```

```

16
17 nodes_in(0) <= "100000001";
18 wait for 5 ns;
19 nodes_in(1) <= "100100011";
20 wait for 15 ns;
21 nodes_in(0) <= "000000000";
22 wait for 5 ns;
23 nodes_in(1) <= "000000000";

```

Prima di tutto inviamo contemporaneamente due messaggi diversi da due sorgenti diverse, sorgente 0 e sorgente 1 verso la stessa destinazione. Poi rifacciamo la stessa cosa modificando il messaggio, una delle sorgenti e la destinazione. E' da notare come gli ingressi vengano azzerati dopo un periodo di clock dalla sorgente. Infine sfalsiamo gli ingressi di un tempo che è frazione del periodo di clock per vedere se la rete si comporta come dovrebbe. Nel primo e nel terzo caso dovremmo avere una collisione sullo Smart_Switch_2_0, mentre nel secondo caso la collisione dovrebbe avvenire sullo Smart_Switch_2_2.

Vediamo, in figura 100 la simulazione della rete progettata. Notiamo che la rete si comporta come ci si aspetterebbe dal comportamento descritto in questo capitolo. Infatti, date le collisioni che avvengono sugli switch, i registri vengono caricati con il valore del pacchetto che deve essere memorizzato, mentre l'altro viene inoltrato. Il colpo di clock successivo viene inoltrato il pacchetto memorizzato ed il registro viene resettato il colpo di clock successivo. Anche nel terzo caso di test la rete si comporta come ci si aspetterebbe inoltrando prima un pacchetto, memorizzando quello che arriva mentre l'altro è in trasmissione ed infine inoltrando anche qui il pacchetto memorizzato.

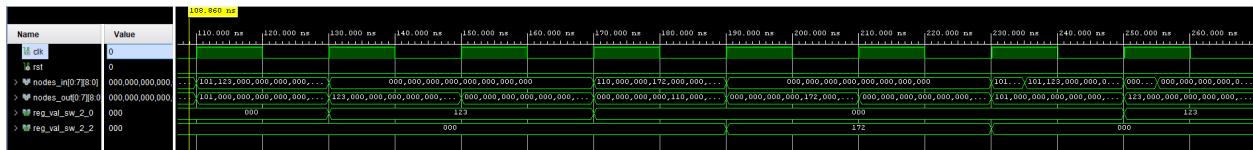


Figura 100: Simulazione dell'Omega Network "smart"

10.3 Progettazione in VHDL con handshaking

Traccia

Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network. Si implementi un protocollo di handshaking semplice regolato da una coppia di segnali (pronto a inviare/pronto a ricevere) per l'invio di ciascun messaggio fra due nodi.

10.3.1 Schematici

L'handshake è stato affrontato nel capitolo 7, per implementarlo nell'Omega Network è necessario che il ricevitore possa mandare un messaggio di ack al nodo trasmettitore. Si pongono davanti due soluzioni:

1. Utilizzare 2 Omega Network, una per mandare messaggi da trasmettitore e ricevitore e un'altra per mandare messaggio nella direzione inversa, in questo modo si possono inviare messaggi in entrambe le direzioni
2. Collegare un nodo sia come sorgente che come destinazione, cioè utilizzare una sola Omega Network con il doppio dei nodi e collegare lo stesso nodo sia tra le sorgenti che tra le destinazioni, in modo che si possa inviare messaggi in entrambi i versi

Abbiamo valutato entrambe le soluzioni, in particolare abbiamo valutato il numero di switch necessari per entrambe, in modo da avere una metrica di paragone oggettiva.

Ponendo di voler collegare N sorgenti con altrettante destinazioni abbiamo bisogno di:

- Per la prima soluzione sono necessari $2 * (N/2\log_2 N = N\log_2 N)$ switch, infatti servono $N/2$ nodi per $(\log_2 N)$ stadi e questo serve per due Omega Network
- Per la seconda soluzione sono necessari $(2 * N/2\log_2 2N) = N\log_2 2N$ switch, infatti serve una Omega Network con il doppio dei nodi, quindi al posto di N ci sarà $2N$

Da questo semplice confronto vediamo che la seconda soluzione richiede sempre un numero maggiore o uguale di switch rispetto la prima. Inoltre abbiamo notato come la seconda soluzione non sia troppo realistica, infatti un'Omega Network, ma in generale una rete di interconnessione, connette nodi che possono essere anche lontani tra loro, in questo caso non sarebbe possibile collegare uno stesso nodo ad entrambe le estremità perché sarebbero lontane tra loro. Fatte queste considerazioni la scelta è diventata semplice, abbiamo infatti optato per la prima soluzione, utilizzando 2 Omega Network per far viaggiare i messaggi in entrambe le direzioni.

Basandoci sull'esercizio precedente abbiamo effettuato alcune modifiche.

E' stato necessario ristrutturare il pacchetto visto nella precedente sezione. Infatti è stato necessario inserire un bit per identificare un pacchetto come parte dell'handshake. Il bit ha significato di bit di "go" se il pacchetto viaggia nella prima Omega Network, cioè quella che collega trasmettitori a riceventi, mentre il nuovo bit ha significato di bit di "ack" se il

pacchetto viaggia nella seconda Omega Network, cioè quella che collega riceventi a trasmettitori. La nuova struttura del pacchetto è visualizzata in figura 101. Questo meccanismo si avvicina al three way handshake che viene utilizzato nelle reti reali dal protocollo HTTP.

1	1	3	3	2
0/1	HS bit	Sorgente	Destinazione	Messaggio

Figura 101: Pacchetto

L'ultima modifica fatta riguarda l'automa dell'unità di controllo, questo perchè il funzionamento dell'unità di controllo vista nella sezione 10.2.1 non era allineato con il funzionamento dei nodi ricevente e trasmettitore tipici dell'handshake. Infatti il nodo trasmettitore continua a mantenere il messaggio sulla linea di uscita finchè non riceve l'ack dal nodo ricevente. Quindi accadeva che, in presenza di una collisione su uno switch in cui il messaggio viene memorizzato, il nodo trasmettitore mantiene il pacchetto in uscita (perchè non riceve l'ack), quindi lo switch su cui è avvenuta la collisione inoltrerà il messaggio vecchio, precedentemente memorizzato, e salverà nel registro nuovamente lo stesso messaggio che è ancora in ingresso allo switch. Quindi il nodo ricevente riceverà 2 volte lo stesso messaggio (solamente due volte perchè alla ricezione della prima copia del messaggio invia l'ack e quindi il trasmettitore imposta 0 sulla linea di uscita) e quindi invierà 2 ack al nodo trasmettitore. Questa cosa è naturalmente non corretta, quindi abbiamo deciso di modificare l'unità di controllo dello switch. Non abbiamo modificato il nodo trasmettitore in quanto l'esercizio riguarda l'Omega Network, quindi abbiamo deciso di prendere un'implementazione dell'handshake fissata, come se fosse una black box da utilizzare per l'esercizio, e modificare la rete di interconnessione in modo da fornire un funzionamento corretto.

La modifica effettuata è molto semplice, nello stato di full buffer, se non c'è un pacchetto sulla prima linea svuotiamo a prescindere il registro e quindi perdiamo l'eventuale pacchetto sulla seconda linea. Questa soluzione funziona perchè il pacchetto memorizzato è sempre quello sulla seconda linea, quindi scartando i successivi pacchetti a quello memorizzato scartiamo tutti i pacchetti ripetuti che vengono inviati perchè c'è un ritardo nella rete. La soluzione potrebbe essere ulteriormente migliorata in quanto, se dopo la prima collisione avviene una seconda collisione con il pacchetto ripetuto ma su uno switch degli stadi precedenti ho comunque lo stesso pacchetto che arriva 2 volte a destinazione. Questo naturalmente può degenerare e succede su tutti gli stadi della rete che sono posizionati prima di quello su cui avviene la prima collisione. Questo accadrebbe in condizioni di grande saturazione

della rete, quindi abbiamo deciso di implementare comunque la soluzione descritta in quanto valida nel caso medio, nonostante permanga il problema degli ack duplicati quando la rete è molto satura (si potrebbero usare ulteriori meccanismi di controllo dei pacchetti, come loro numerazione, per evitare questo problema).

Passiamo quindi a vedere gli schematici dei componenti. Tutto rimane quasi uguale a quello visto nella sezione precedente, tranne il fatto che tutti va ridimensionato per pacchetti da 10 bit. Il componente che cambia è l'unità di controllo dello smart switch, mentre vengono introdotti il nodo ricevente e il nodo trasmettitore, vediamoli.

Unità di controllo dello smart switch

Come accennato l'unità di controllo cambia, ma la modifica viene effettuata solamente all'automa. In figura 102 sia l'automa che lo schematico.

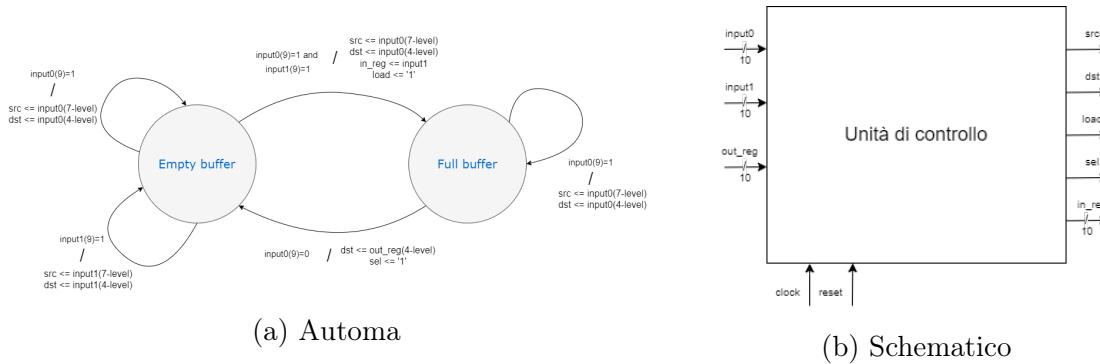


Figura 102: Automa e schematico dell'unità di controllo

Nodo trasmettitore

Per il nodo trasmettitore vale il funzionamento standard di un handshake semplice che abbiamo trattato nel capitolo 7. In figura 103 sia l'automa che lo schematico. Notiamo che il nodo ha un ingresso di start per iniziare la trasmissione, un ingresso relativo all'output della seconda Omega Network (ack), e 2 indirizzi che servono a fornire il messaggio da inviare e l'indirizzo di destinazione del messaggio stesso. In output invece c'è solo il collegamento alla prima Omega Network.

Nodo ricevente

Anche per il nodo ricevente vale il funzionamento standard di un handshake semplice che abbiamo trattato nel capitolo 7. In figura 104 sia l'automa che lo schematico. Notiamo

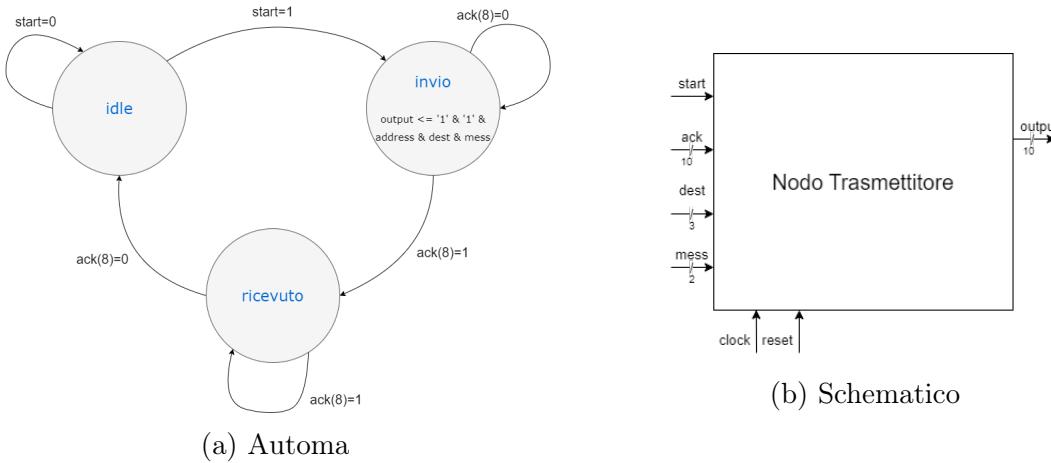


Figura 103: Automa e schematico del nodo trasmettitore

che il nodo ha solo un ingresso che è l'output relativo alla prima Omega Network (input). In output invece c'è solo il collegamento alla seconda Omega Network per inviare l'ack del messaggio ricevuto.

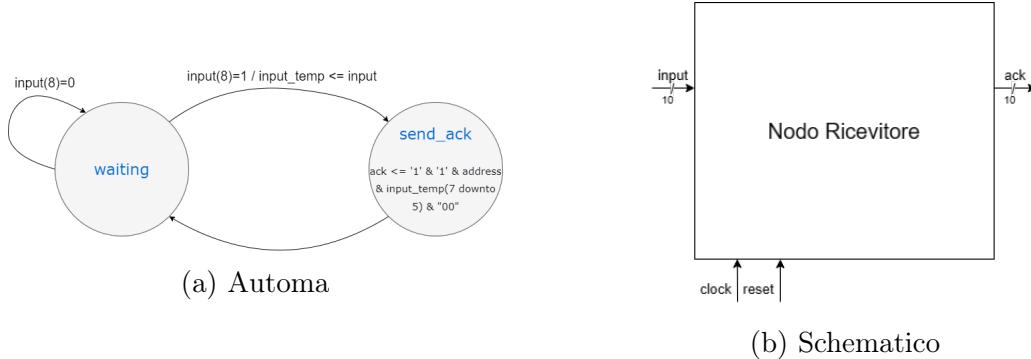


Figura 104: Automa e schematico del nodo ricevente

Sistema complessivo

Il sistema complessivo è realizzato collegando opportunamente le due Omega Network ai nodi trasmettitore e ricevitore. Tutti i nodi trasmettitore sono collegati come sorgenti della prima Omega Network e come destinatari della seconda Omega Network. Mentre tutti i nodi ricevitore sono collegati come destinatari della prima Omega Network e come sorgenti della seconda Omega Network. Lo schematico non è riportato in quanto diverrebbe di complesso rappresentare il sistema. In ingresso vi è un vettore di 8 bit per il segnale di start, in questo modo si può abilitare la trasmissione per più nodi contemporaneamente; messaggi

e destinazioni per ciascun nodo trasmettitore della rete, mentre in output c'è il pacchetto ricevuto dai nodi ricevitori.

10.3.2 Codice VHDL

Vediamo il codice che implementa le modifiche effettuate rispetto lo scorso esercizio.

Unità di controllo dello smart switch

Il codice dell'unità di controllo è simile al precedente, ma con le modifiche presentate precedentemente nello stato di full buffer. Di seguito il codice.

```

1 entity uc_switch is
2 generic (
3     level : integer := 0
4 );
5 port (
6     clock      : in std_logic;
7     reset      : in std_logic;
8     input0     : in std_logic_vector(9 downto 0);
9     input1     : in std_logic_vector(9 downto 0);
10    out_reg   : in std_logic_vector(9 downto 0);
11    src        : out std_logic;
12    dst        : out std_logic;
13    load       : out std_logic;
14    in_reg    : out std_logic_vector(9 downto 0);
15    sel        : out std_logic
16 );
17 end uc_switch;
18
19 architecture Behavioral of uc_switch is
20 type stato is (empty_buffer, full_buffer);
21 signal curr_state : stato := empty_buffer;
22 signal next_state : stato := empty_buffer;
23 begin
24 registri : process (clock)
25 begin
26 if rising_edge(clock) then
27
28     if reset='1' then
29         curr_state <= empty_buffer;
30     else
31         curr_state  <= next_state;
32     end if;
33 end if;
34 end process;
35
36 calcolo_stato : process(curr_state, input0, input1)
37 begin
38 load <= '0';
39 sel <= '0';
40 in_reg <= (others => '0');
41 src <= '0';
42 dst <= '0';

```

```

43
44 case curr_state is
45   when empty_buffer =>
46     if (input0(9)='1' and input1(9)='1') then
47       src <= input0(7-level);
48       dst <= input0(4-level);
49       in_reg <= input1;
50       load <= '1';
51       next_state <= full_buffer;
52
53     elsif (input0(9)='1') then
54       src <= input0(7-level);
55       dst <= input0(4-level);
56       next_state <= empty_buffer;
57
58     elsif (input1(9)='1') then
59       src <= input1(7-level);
60       dst <= input1(4-level);
61       next_state <= empty_buffer;
62
63   end if;
64
65   when full_buffer =>
66     if (input0(9)='1') then
67       src <= input0(7-level);
68       dst <= input0(4-level);
69       next_state <= full_buffer;
70
71     else
72       dst <= out_reg(4-level);
73       sel <= '1';
74       next_state <= empty_buffer;
75
76   end if;
77
78 end case;
79 end process;
80 end Behavioral;

```

Nodo trasmettitore

Il codice VHDL del nodo trasmettitore implementa il funzionamento tipico di un handshake semplice. L'automa visto nella sezione degli schematici viene implementato utilizzando due process, uno per il implementare il registro, mentre l'altro per implementare la parte combinatoria.

Notiamo come nello stato di invio venga costruito opportunamente il pacchetto da inviare utilizzando l'operatore di concatenazione. Il pacchetto segue la struttura che abbiamo già visto in precedenza e vengono utilizzate anche le informazioni in ingresso relative al destinatario ed al messaggio (dest e mess). Notiamo come l'indirizzo del nodo sia da fornire dall'esterno, cioè quando il nodo viene utilizzato, infatti è un parametro presente nella sezione generic dell'entità. Di seguito il codice.

```

1 entity sender_node is
2 generic ( address : std_logic_vector(2 downto 0) );
3 port (
4     clock      : in std_logic;
5     reset      : in std_logic;
6     start      : in std_logic;
7     ack        : in std_logic_vector(9 downto 0);
8     dest        : in std_logic_vector(2 downto 0);
9     mess        : in std_logic_vector(1 downto 0);
10    output       : out std_logic_vector(9 downto 0)
11 );
12 end sender_node;
13
14 architecture Behavioral of sender_node is
15 type stato is (idle, invio, ricevuto);
16 signal curr_state : stato := idle;
17 signal next_state : stato := idle;
18 begin
19
20 registri : process (clock)
21 begin
22 if rising_edge(clock) then
23
24     if reset='1' then
25         curr_state <= idle;
26     else
27         curr_state  <= next_state;
28     end if;
29 end if;
30 end process;
31
32 calcolo_stato : process(curr_state, start, ack)
33 begin
34 output <= (others => '0');
35
36 case curr_state is
37     when idle      =>
38         if (start = '1') then
39             next_state <= invio;
40         else
41             next_state <= idle;
42         end if;
43     when invio     =>
44         output <= '1' & '1' & address & dest & mess;
45         if (ack(8) = '1') then
46             next_state <= ricevuto;
47         else
48             next_state <= invio;
49         end if;
50     when ricevuto  =>
51         if (ack(8) = '0') then
52             next_state <= idle;
53         else
54             next_state <= ricevuto;
55         end if;
56
57 end case;
58 end process;
59

```

```
60 end Behavioral;
```

Nodo ricevente

Il codice VHDL del nodo ricevitore implementa il funzionamento tipico di un handshake semplice. L'automa visto nella sezione degli schematici viene implementato utilizzando due process, uno per il implementare il registro, mentre l'altro per implementare la parte combinatoria.

Notiamo come nello stato di send_ack venga costruito opportunamente il pacchetto di ack da inviare al nodo trasmittitore utilizzando la seconda. Il pacchetto costruito segue la struttura che abbiamo già visto in precedenza e viene effettuata anche un'operazione di slicing per estrarre l'indirizzo del trasmittitore che sarà l'indirizzo di destinazione dell'ack. Notiamo come l'indirizzo del nodo sia da fornire dall'esterno, cioè quando il nodo viene utilizzato, infatti è un parametro presente nella sezione generic dell'entità. Di seguito il codice.

```

1 entity receiver_node is
2 generic ( address : std_logic_vector(2 downto 0) );
3 port (
4     clk      : in std_logic;
5     reset    : in std_logic;
6     input    : in std_logic_vector(9 downto 0);
7     ack      : out std_logic_vector(9 downto 0)
8 );
9 end receiver_node;
10
11 architecture Behavioral of receiver_node is
12 type stato is (waiting, send_ack);
13 signal curr_state : stato := waiting;
14 signal next_state : stato := waiting;
15 signal input_temp : std_logic_vector(9 downto 0);
16 begin
17
18 registri : process (clk)
19 begin
20 if rising_edge(clk) then
21
22 if reset='1' then
23     curr_state <= waiting;
24 else
25     curr_state <= next_state;
26 end if;
27 end if;
28 end process;
29
30
31 calcolo_stato : process(curr_state, input)
32 begin
33 ack <= (others => '0');
34
35 case curr_state is

```

```

36
37     when waiting          =>
38         if input(8)='1' then
39             input_temp <= input;
40             next_state <= send_ack;
41         else
42             next_state <= waiting;
43         end if;
44
45     when send_ack          =>
46         ack <= '1' & '1' & address & input_temp(7 downto 5) & "00";
47         next_state <= waiting;
48
49 end case;
50 end process;
51 end Behavioral;

```

Sistema complessivo

Il sistema complessivo è una semplice composizione dei componenti realizzati. L'Omega Network è la stessa vista nello scorso esercizio, con le opportune modifiche all'unità di controllo dello switch. I nodi sender e receiver sono realizzati come visto sopra e il mapping delle porte di ciascun nodo viene fatto utilizzando un for-generate. Per definire le destinazioni, i messaggi, le destinazioni e gli output si sono utilizzati dei tipi definiti da noi. Questi sono essenziali per permettere il mapping utilizzando i for-generate e semplificano anche la scrittura dell'interfaccia del componente. Di seguito il codice per il package che include i tipi definiti.

```

1 package type_package is
2
3 type input_array is array (0 to 7) of std_logic_vector(9 downto 0);
4 type input_message is array (0 to 7) of std_logic_vector(1 downto 0);
5 type input_destination is array (0 to 7) of std_logic_vector(2 downto 0);
6
7 end package type_package;
8
9 package body type_package is
10 end package body type_package;

```

Qui invece il codice del sistema complessivo.

```

1 entity HS_omega_network is port (
2     clock      : in std_logic;
3     reset      : in std_logic;
4     start      : in std_logic_vector(7 downto 0);
5     mess       : in input_message;
6     dest       : in input_destination;
7     output     : out input_array
8 );

```

```

9  end HS_omega_network;
10
11 architecture structural of HS_omega_network is
12 signal node_in_sender : input_array;
13 signal node_in_receiver : input_array;
14 signal node_out_sender : input_array;
15 signal node_out_receiver : input_array;
16 begin
17 output <= node_out_sender;
18
19 senders : for i in 0 to 7 generate
20 sender : entity work.sender_node
21 generic map ( address => std_logic_vector(to_unsigned(i, 3)))
22 port map(
23 clock => clock,
24 reset => reset,
25 start => start(i),
26 ack => node_out_receiver(i),
27 dest => dest(i),
28 mess => mess(i),
29 output => node_in_sender(i)
30 );
31 end generate;
32
33 receivers : for i in 0 to 7 generate
34 receiver : entity work.receiver_node
35 generic map ( address => std_logic_vector(to_unsigned(i, 3)))
36 port map(
37 clk => clock,
38 reset => reset,
39 input => node_out_sender(i),
40 ack => node_in_receiver(i)
41 );
42 end generate;
43
44 omega_sender : entity work.smart_omega_network port map(
45 clock => clock,
46 reset => reset,
47 nodes_in => node_in_sender,
48 nodes_out => node_out_sender
49 );
50
51 omega_receiver : entity work.smart_omega_network port map(
52 clock => clock,
53 reset => reset,
54 nodes_in => node_in_receiver,
55 nodes_out => node_out_receiver
56 );
57
58 end structural;

```

10.3.3 Simulazione

In simulazione abbiamo testato la gestione delle collisione degli smart switch cercando di capire se rispettassero le specifiche decise. Di seguito il codice della simulazione.

```

1 mess(0) <= "01";
2 mess(1) <= "11";
3 dest(0) <= "000";
4 dest(1) <= "000";
5
6 wait for clk_period*2;
7
8 start <= "00000011";
9 wait for clk_period;
10 start <= "00000000";

```

Inviamo contemporaneamente, dal trasmettitore 0 ed 1, un messaggio verso il ricevitore 0. Quindi ci sarà sicuramente una collisione che dovrà essere gestita da uno switch.

Vediamo, in figura 105 la simulazione del sistema progettato. Notiamo che il sistema si comporta come ci si aspetterebbe dal comportamento descritto in questo capitolo. Notiamo come, in primo luogo, il nodo trasmettitore costruisca in modo corretto il pacchetto e, in secondo luogo, come la rete gestisca opportunamente la collisione, infatti i pacchetti (da trasmettitore 0 e 1) arrivano a destinazione in due momenti diversi nonostante siano partiti contemporaneamente, in particolare il pacchetto del trasmettitore 1 arriva successivamente perché il nodo dello switch memorizza sempre il pacchetto sulla seconda linea. Inoltre possiamo notare come, dopo la ricezione del pacchetto, il nodo ricevitore 0 risponda con un ack ad entrambi i nodi trasmettitori dopo un periodo di clock necessario per il passaggio di stato (ack_sender è ciò che riceve il nodo trasmettitore dalla seconda omega network).

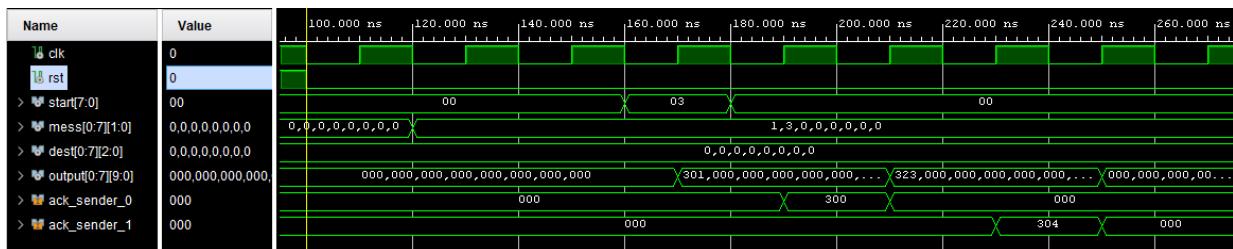


Figura 105: Simulazione del sistema

11 Macchine aritmetiche: Moltiplicatore di Booth

Il moltiplicatore di Booth é una macchina aritmetica sequenziale (data dalla presenza di una parte operativa ed una parte di controllo) con presentazione parallela (visto che sia gli ingressi che le uscite sono dati in parallelo). Il moltiplicatore implementa quello che é l'algoritmo di Booth per il calcolo di una moltiplicazione tra 2 valori espressi in complemento a 2. L'algoritmo si basa sull'omonima codifica, che a partire da un vettore X (in complemento a 2) del tipo:

$$X_{n-1}X_{n-2}\dots x_0$$

Permette di codificare ogni valore di X in un vettore Y con valori pari a:

$$Y_0 = -X_0$$

$$Y_1 = -X_1 + X_0$$

$$Y_2 = -X_2 + X_1$$

...

$$Y_{n-1} = -X_{n-1} + X_{n-2}$$

moltiplicando ogni valore di Y_i per 2^i ($i = \text{pedice}$) e sommando i vari valori ricavati si osserva che:

$$Y_{n-1}*2^{n-1} + Y_{n-2}*2^{n-2} + \dots + Y_1*2^1 + Y_0*2^0 = X_{n-1}*2^{n-1} + X_{n-2}*2^{n-2} + \dots + X_1*2^1 + X_0*2^0$$

Per come viene codificato un valore attraverso la codifica di Booth un qualsiasi valore Y_i puó acquisire un valore tra $\{1, 0, -1\}$, però utilizzando la codifica binaria questa rappresentazione non sarebbe comoda. Quindi per poter rappresentare un valore sulla base della codifica di booth si valuta il valore codificato valutando 2 bit alla volta, seguendo la tabella di codifica in figura 106.

In base al valore di codifica calcolato sul moltiplicatore é possibile eseguire una tra le seguenti operazioni:

- solo operazione di shift, nel caso si abbiano 2 valori uguali adiacenti ("11" o "00");
- operazione di addizione e di shift, nel caso si abbia ("01");
- operazione di sottrazione e di shift, nel caso si abbia ("10").

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

Figura 106: tabella di codifica di Booth

Dovendo sviluppare l'algoritmo su una macchina è necessario comprendere prima i componenti (la cui interconnessione è presente in figura 107) e successivamente come una moltiplicazione può essere eseguita su un circuito.

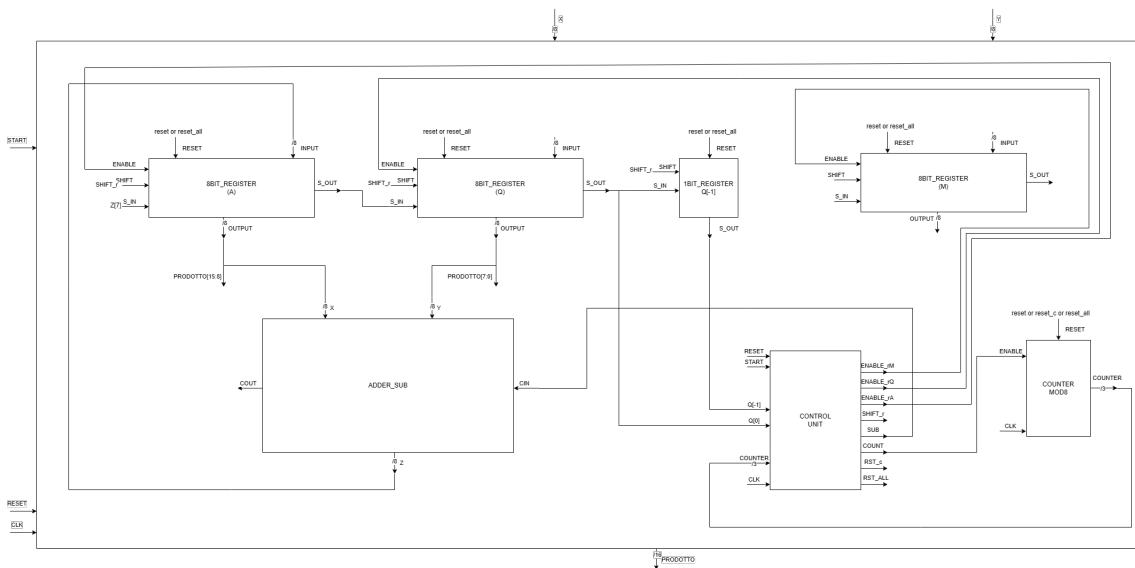


Figura 107: moltiplicatore di Booth

Per svolgere una moltiplicazione su un circuito si usano 4 shift registers:

- 1 per acquisire un operando;
- 1 per acquisire l'altro operando;
- 1 per avere gli $n/2$ bit più significativi del prodotto;
- 1 registro ad un bit per mantenere il valore $Q[-1]$.

I valori vengono salvati all'interno dei registri e la control unit segue passo passo l'algoritmo di Booth:

- salva i valori nei 2 registri per gli operandi;
- controlla il valore del contatore (per vedere quanti passaggi sono stati eseguiti);
- controlla se va eseguita la somma o la sottrazione;
- attende l'operazione;
- salva il risultato nel registro A;
- esegue uno shift sui 3 registri ($A, Q, Q[-1]$).

Dopo che il controllore esegue questo algoritmo è disponibile il valore del prodotto sull'uscita del moltiplicatore.

11.1 Progettazione in VHDL ed implementazione su board

11.1.1 Traccia

Progettare ed implementare in VHDL una macchina aritmetica sequenziale a scelta fra le seguenti:

- moltiplicatore di Robertson, per effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna;
- moltiplicatore di Booth, per effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna;
- divisore non-restoring, per effettuare la divisione intera fra due stringhe A e B di 4 bit ciascuna;
- divisore restoring, per effettuare la divisione intera fra due stringhe A e B di 4 bit ciascuna;

Opzionalmente, la macchina implementata può essere sintetizzata su FPGA e testata mediante l'utilizzo dei dispositivi di input/output (switch, bottoni, led, display) presenti sulla board di sviluppo in dotazione al gruppo. La modalità di utilizzo degli stessi è a completa discrezione degli studenti.

11.1.2 Schematici

Il moltiplicatore sviluppato (come detto in precedenza) é il moltiplicatore di Booth, il componente é stato sviluppato in modo strutturale e per il suo funzionamento é stato necessario avere sia una parte operativa contenente:

- 3 registri di 8 bit: registro A, che mantiene il risultato della somma parziale, registro Q, che mantiene sia il moltiplicatore in ingresso che i bit da non considerare nelle somme parziali ed il registro M, che mantiene il valore del moltiplicando;
- un registro di 1 bit, utilizzato per mantenere il valore di $Q[-1]$
- un contatore modulo 8, che viene utilizzato per sapere quando é da considerare terminata l'operazione di moltiplicazione;
- un adder_sub, che gestisce l'operazione di somma e differenza dei valori in ingresso;
- una control_unit, che gestisce i componenti seguendo i passi dettati dall'algoritmo di Booth.

8bit_register

Il componente é uno shift register con ingresso ed uscita sia seriale che parallelo e viene utilizzato sia per acquisire i valori in ingresso alla macchina, sia per salvare tutte le addizioni parziali della moltiplicazione, sia per mantenere il risultato della moltiplicazione in uscita alla macchina.

1bit_register

Il componente é un semplice registro ad un bit (quindi un flip flop D), il componente é utilizzato per mantenere il valore $Q[-1]$ utilizzato dalla control unit, per valutare quale operazione va effettuata sugli operandi in ingresso all'adder_sub. Il componente poteva essere evitato se il registro Q fosse stato implementato a 9 bit.

counter_mod8

Il componente é un contatore con la sola uscita di conteggio ed é utilizzato per mantenere in memoria il numero di addizioni parziali eseguite durante il processo di moltiplicazione.

full_adder

Il componente Full_adder é utilizzato per calcolare la somma tra 3 bit (2 bit di ingresso, A e B, ed 1 bit di riporto, Cin), i risultati della somma si trovano su 2 bit di uscita:

$$S = A \text{ XOR } B \text{ XOR } Cin$$

$$Cout = (A \text{ AND } B) \text{ OR } (Cin \text{ AND } (A \text{ XOR } B))$$

Il componente ha la struttura come in figura 108.

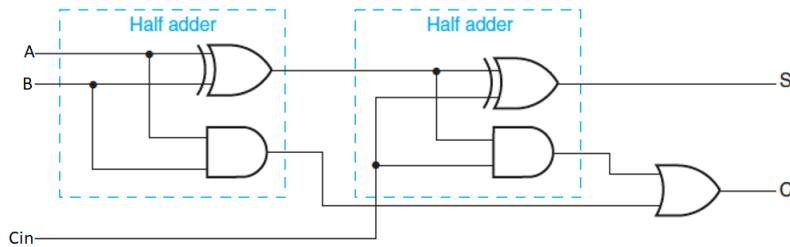


Figura 108: struttura di un full adder

ripple_carry

Il componente é il vero e proprio addizionatore utilizzato, é sviluppato sulla base di 8 blocchi di full adder (1 blocco per bit) e permette la somma di 2 valori ad 8 bit. Gli 8 full adder sono inseriti a cascata, infatti in ogni blocco in posizione i si avrà:

$$Cin_i = Cin_{i-1}$$

$$A_i = A_i$$

$$B_i = B_i$$

Questa struttura nonostante sia facile da implementare, comporta anche un aumento della tempo di esecuzione con l'aumentare dei bit degli operandi, infatti ogni singolo full_adder deve attendere la propagazione del carry dal full_adder precedente affinché possa dare il risultato corretto. Il problema é ovviato dall'adder Carry-lookahead, ma non é stato implementato in questo esercizio. Il ripple carry adder é presente in figura 109

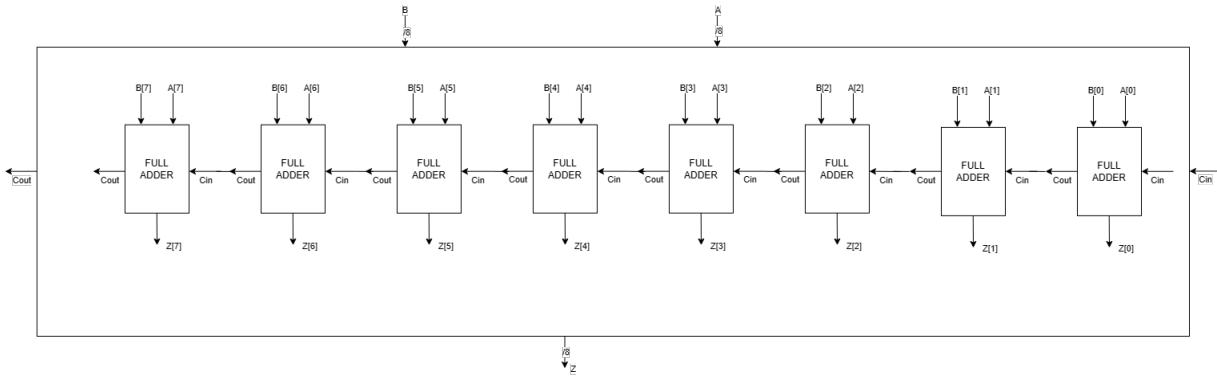


Figura 109: ripple carry adder

Adder_sub

Il componente é utilizzato come gestore dell'operazione da svolgere, infatti permette di utilizzare il ripple carry adder sia come un addizionatore che come un sottrattore. La modifica dell'operazione é possibile grazie a 2 fattori:

- l'uso di ingressi in complemento a 2;
- la possibilità di rende un numero di invertire il segno di un numero invertendo i suoi bits ed aggiungendo 1.

L'inversione dei bits é data dalle xor in cui entra il segnale y ed il segnale di sub. Il componente é presentato in figura 110

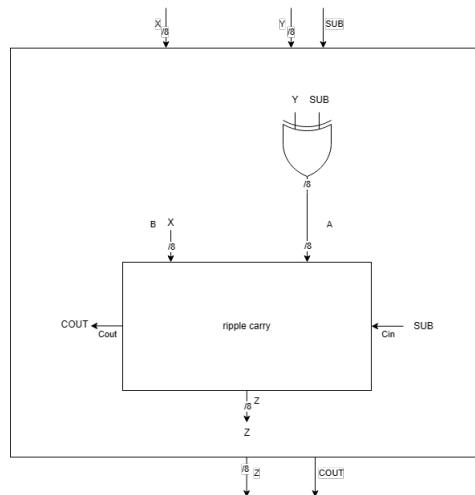


Figura 110: componente adder sub

control unit

La control unit fa da parte di controllo e gestisce i segnali di enable, shift, il segnale di conteggio ed il segnale di sub. Il componente é stato sviluppato in modo comportamentale con l'implementazione di una macchina a stati finiti con i seguenti stati:

- idle, che é lo stato in cui si attende lo start per avviare la moltiplicazione;
- prelievo, che é lo stato in cui vengono salvati gli operandi all'interno dei registri Q ed M;
- inizio, che di passaggio per permettere il corretto load dei registri Q ed M;
- operazione, che é lo stato in cui si decide se svolgere la somma o la sottrazione ed in cui il risultato viene salvato nuovamente in A;
- shift, che é lo stato in cui viene fatta l'operazione di shift sui registri A, Q e Q[-1] ed in cui viene valutato il valore del contatore e quindi se l'operazione puó considerarsi conclusa. Se l'operazione non è conclusa si incrementa il contatore e si torna nello stato inizio, altrimenti si va in idle;

La control unit potrebbe essere migliorata prevedendo anche un segnale di stop, che sarebbe stato poi messo in uscita al moltiplicatore per indicare che l'operazione è terminata e il risultato corretto è presente in uscita. Un'implementazione di questo tipo è stata utilizzata per il moltiplicatore usato nell'esercizio libero, trattato nella sezione 12.

L'automa é presente in figura 111.

11.1.3 Codice VHDL

full_adder

Il componente full_adder é stato sviluppato in modo dataflow dando direttamente le funzioni da inviare in uscita sui segnali s e cout. Di seguito é presente il codice VHDL.

```

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;

```

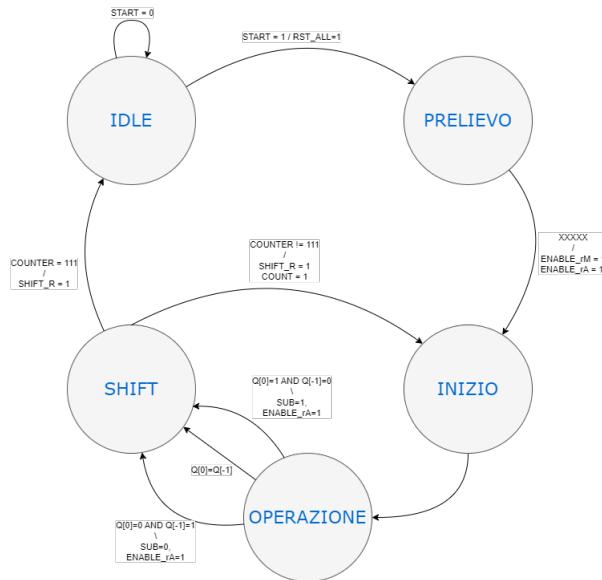


Figura 111: automa control unit

```

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;

```

ripple_carry

Il componente `ripple_carry` é stato sviluppato in modo strutturale e forma un ripple carry adder, collegando tra loro dei full adder. Di seguito é presente il codice vhdl.

```

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 RAO: 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) , Z(i));
27 END GENERATE;
28
29 RA7: full_adder port map(X(7) , Y(7) , temp(6) , c_out , Z(7));
30
31 end structural;

```

Adder_sub

Il componente adder_sub é stato sviluppato in modo strutturale e viene utilizzato per generare il complemento di Y (quando il segnale di sub in ingresso vale 1), tramite la struttura FOR...GENERATE e collega gli ingressi al componente ripple_carry. Di seguito é presente il codice vhdl del componente.

```

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 complementoy_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;

```

control unit

La control unit é stata sviluppata in modo comportamentale e gestisce tutti i componenti della parte operativa del moltiplicatore di booth (contatore, registri ed addizionatore). Di seguito é presente il codice VHDL del componente.

```

1
2
3 library IEEE;
4 use IEEE.STD_LOGIC_1164.ALL;
5
6 entity control_unit is port (
7     clock      : in std_logic;
8     reset      : in std_logic;
9     start      : in std_logic;
10    counter    : in std_logic_vector(2 downto 0);
11    q_0        : in std_logic;
12    q_meno1   : in std_logic;
13    enable_rA  : out std_logic;
14    enable_rQ  : out std_logic;
15    enable_rM  : out std_logic;
16    shift_r    : out std_logic;
17    sub        : out std_logic := '0';
18    count      : out std_logic;
19    rst_c      : out std_logic;
20    rst_all    : out std_logic
21 );
22 end control_unit;
23
24 architecture Behavioral of control_unit is
25
26 type stato is (idle, prelievo, inizio, operazione, shift);
27 signal curr_state : stato := idle;
28 signal next_state : stato := idle;
29
30 begin
31
32 registri : process (clock)
33 begin
34 if rising_edge(clock) then
35
36     if reset='1' then
37         curr_state <= idle;
38     else
39         curr_state  <= next_state;
40     end if;
41 end if;
42 end process;
43
44 calcolo_stato : process(curr_state, start)
45 begin
46
47 rst_all <= '0';
48 rst_c <= '0';
49 enable_rA <= '0';
50 enable_rQ <= '0';
51 enable_rM <= '0';

```

```

52 shift_r <= '0';
53 sub <= '0';
54 count <= '0';
55
56 case curr_state is
57   when idle          =>
58     if (start = '1') then
59       rst_all <= '1';
60       next_state <= prelievo;
61     else
62       next_state <= idle;
63     end if;
64   when prelievo      =>
65     enable_rM <= '1';
66     enable_rQ <= '1';
67     next_state <= inizio;
68   when inizio         =>
69     next_state <= operazione;
70   when operazione    =>
71     if (q_0 = q_meno1) then
72       next_state <= shift;
73     else
74       if (q_0='0' and q_meno1='1') then
75         sub <= '0';
76       elsif (q_0='1' and q_meno1='0') then
77         sub <= '1';
78       end if;
79       enable_ra <= '1';
80       next_state <= shift;
81     end if;
82   when shift          =>
83     shift_r <= '1';
84     if (counter = "111") then
85       next_state <= idle;
86     else
87       count <= '1';
88       next_state <= inizio;
89     end if;
90
91 end case;
92 end process;
93
94 end Behavioral;

```

11.1.4 Simulazione

In simulazione sono stati testati i 4 casi di testing:

- numero positivo x numero positivo $3 * 3$, in figura 112;
- numero negativo x numero negativo $-3 * (-3)$, in figura 113;
- numero positivo x numero negativo $3 * (-3)$, in figura 114;
- numero negativo x numero positivo $(-3) * 3$, in figura 115.

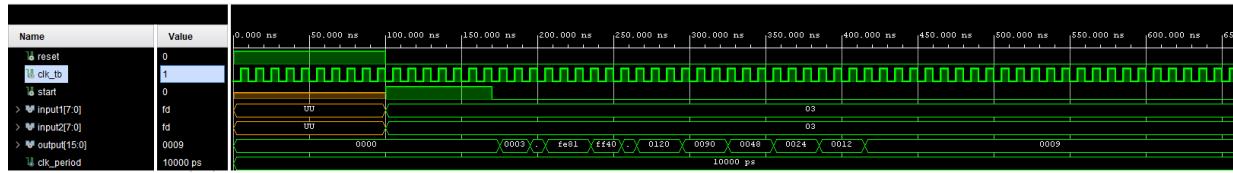


Figura 112: simulazione 1



Figura 113: simulazione 2

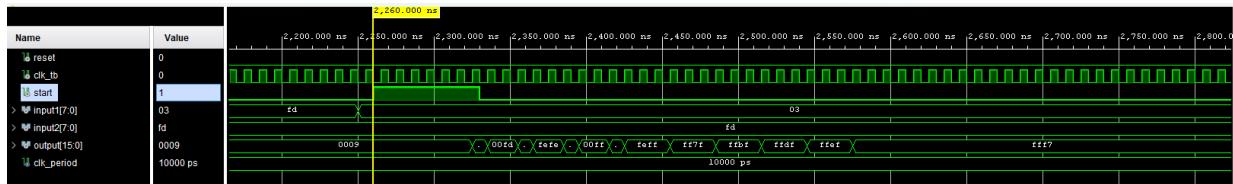


Figura 114: simulazione 3

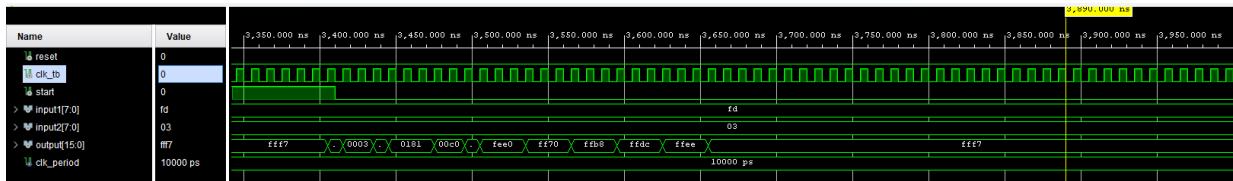


Figura 115: simulazione 4

11.2 Implementazione su board

Traccia

Opzionalmente, la macchina implementata può essere sintetizzata su FPGA e testata mediante l'utilizzo dei dispositivi di input/output (switch, bottoni, led, display) presenti sulla board di sviluppo in dotazione al gruppo. La modalità di utilizzo degli stessi è a completa discrezione degli studenti.

Schematici

Per implementare il moltiplicatore su board abbiamo deciso di utilizzare i 16 switch ed i 16 led a disposizione. I due operandi sono da 8 bit, quindi possiamo recuperarli entrambi

con un solo passaggio. Il risultato, invece, è su 16 bit quindi possiamo visualizzarlo sui led a disposizione. Abbiamo quindi utilizzato un bottone come segnale di start, quando viene premuto si presuppone che gli operandi siano disponibili ed inizia la moltiplicazione. Al termine il risultato sarà visualizzabile in binario sui led.

Per fare ciò abbiamo utilizzato un debouncer, lo stesso utilizzato nei precedenti esercizi, per pulire il bottone di start e abbiamo scritto i constraint per collegare inputs ed outputs agli elementi su board. In figura 116 lo schematico completo.

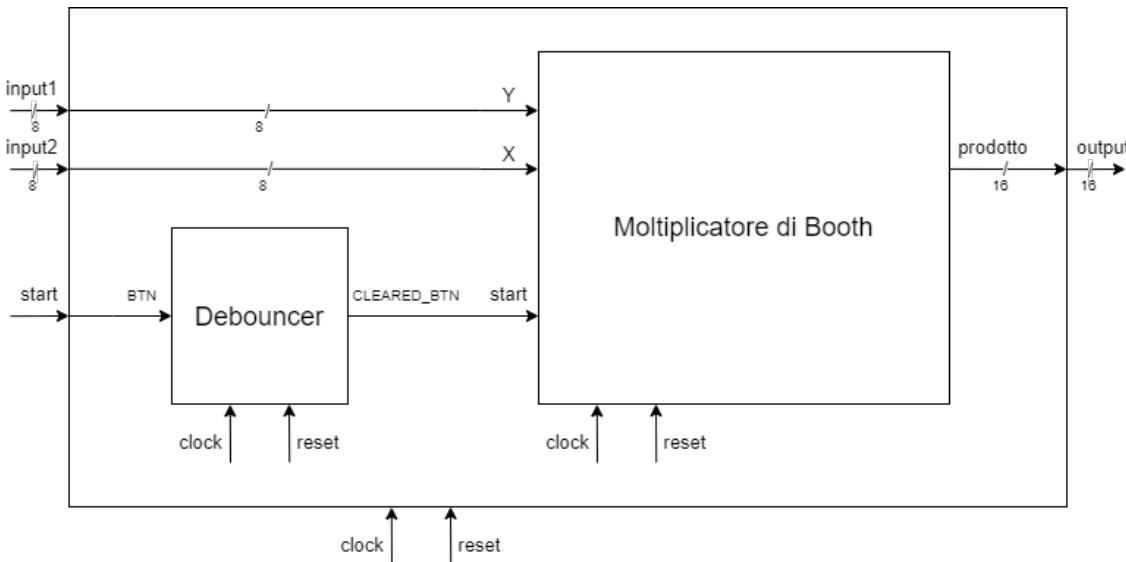


Figura 116: Moltiplicatore di booth su board

Durante la soluzione dell'esercizio però abbiamo sempre considerato che l'addizionatore utilizzato riuscisse a terminare la somma in un tempo minore di un periodo di clock, infatti l'operazione di prelievo del risultato avviane un periodo di clock dopo aver posizionato gli operandi. Questo potrebbe essere non vero nella realtà, quindi potrebbe essere necessario rallentare la dinamica del moltiplicatore per adeguarsi al tempo di calcolo dell'addizionatore. Quindi abbiamo valutato l'utilizzo di un divisore di frequenza per rallentare l'evaluzione dell'automa dell'unità di controllo e conseguentemente utilizzare il segnale del divisore in and con i segnali forniti dall'unità di controllo verso i registri ed il contatore.

Prima però di introdurre il divisore nel progetto del nostro moltiplicatore abbiamo controllato che effettivamente fosse necessario. Quindi abbiamo fatto una Time Analysis sull'intero sistema per controllare che non vi fossero path critici, cioè path con slack negativo, quindi path su cui la propagazione di un valore avviene in ritardo rispetto il tempo atteso. Quindi, dopo aver fatto l'implementazione su board del componente sopra illustrato e dopo aver

aggiunto il constraint sul clock, che è definito come un'onda quadra con periodo di 10 ns e variazione a 5 ns, abbiamo controllato il Timing Summary fornito da Vivado. E' necessario definire il clock perchè lo slack viene calcolato usando il tempo di propagazione dei segnali tra registri rispetto il periodo di clock. Se un segnale impiega meno di un periodo di clock allora lo slack è negativo quindi il segnale si propaga da un'estremità all'altra del path in meno di un clock, se impiega di più significa che il segnale non avrà correttamente raggiunto l'altra estremità del path al prossimo colpo di clock e quindi questo risulterebbe in un errore. In figura 117 è riportato il summary.

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 2,088 ns	Worst Hold Slack (WHS): 0,179 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: 164	Total Number of Endpoints: 164	Total Number of Endpoints:	77

All user specified timing constraints are met.

Figura 117: Timing Summary del moltiplicatore di Booth su board

Notiamo come non ci sia nessun path con slack negativo, quindi su tutti i path possibili all'interno dell'architettura del moltiplicatore su board il segnale viene propagato in meno di un periodo di clock, quindi lo slack sarà positivo anche su tutti i path che va da un registro qualsiasi a monte dell'adder verso un registro qualsiasi a valle dell'adder. Uno slack positivo significa Ciò significa che il nostro addizionatore riesce a completare un'addizione in meno di un clock. Quindi per questa specifica applicazione non è necessario utilizzare un divisore di frequenza.

Codice VHDL

Per implementare il moltiplicatore su board abbiamo solamente bisogno di interconnettere dei componenti. Di seguito il codice.

```

1 entity Multiplier_on_board is port (
2   clock      : in std_logic;
3   reset      : in std_logic;
4   start      : in std_logic;
5   input1     : in std_logic_vector(7 downto 0);
6   input2     : in std_logic_vector(7 downto 0);
7   output     : out std_logic_vector(15 downto 0)
8 );
9 end Multiplier_on_board;
```

```

10
11 architecture structural of Multiplier_on_board is
12 signal start_clear : std_logic;
13 signal reset_clear : std_logic;
14 begin
15
16 multiplier : entity work.Booth_multiplier port map (
17     clock      => clock,
18     reset      => reset,
19     start      => start_clear,
20     Y          => input1,
21     X          => input2,
22     prodotto   => output
23 );
24
25 deb_start : entity work.ButtonDebouncer port map (
26     RST        => '0',
27     CLK        => clock,
28     BTN        => start,
29     CLEARED_BTN => start_clear
30 );
31
32 end structural;

```

Di seguito invece riportiamo i constraint utilizzati per definire il clock e per collegare gli input ed output agli elementi su board.

```

1 ## Clock signal
2 set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { clock }]; #
3           IO_L12P_T1_MRCC_35 Sch=clk100mhz
4 create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { clock }];
5
6 ##Switches
7 set_property -dict { PACKAGE_PIN J15     IOSTANDARD LVCMOS33 } [get_ports { input1[0] }]; #
8           IO_L24N_T3_RS0_15 Sch=sw[0]
9 set_property -dict { PACKAGE_PIN L16     IOSTANDARD LVCMOS33 } [get_ports { input1[1] }]; #
10          IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
11 set_property -dict { PACKAGE_PIN M13     IOSTANDARD LVCMOS33 } [get_ports { input1[2] }]; #
12          IO_L6N_T0_D08_VREF_14 Sch=sw[2]
13 set_property -dict { PACKAGE_PIN R15     IOSTANDARD LVCMOS33 } [get_ports { input1[3] }]; #
14          IO_L13N_T2_MRCC_14 Sch=sw[3]
15 set_property -dict { PACKAGE_PIN R17     IOSTANDARD LVCMOS33 } [get_ports { input1[4] }]; #
16          IO_L12N_T1_MRCC_14 Sch=sw[4]
17 set_property -dict { PACKAGE_PIN T18     IOSTANDARD LVCMOS33 } [get_ports { input1[5] }]; #
18          IO_L7N_T1_D10_14 Sch=sw[5]
19 set_property -dict { PACKAGE_PIN U18     IOSTANDARD LVCMOS33 } [get_ports { input1[6] }]; #
20          IO_L17N_T2_A13_D29_14 Sch=sw[6]
21 set_property -dict { PACKAGE_PIN R13     IOSTANDARD LVCMOS33 } [get_ports { input1[7] }]; #
22          IO_L5N_T0_D07_14 Sch=sw[7]
23 set_property -dict { PACKAGE_PIN T8      IOSTANDARD LVCMOS18 } [get_ports { input2[0] }]; #
24          IO_L24N_T3_34 Sch=sw[8]
25 set_property -dict { PACKAGE_PIN U8      IOSTANDARD LVCMOS18 } [get_ports { input2[1] }]; #
26          IO_25_34 Sch=sw[9]
27 set_property -dict { PACKAGE_PIN R16     IOSTANDARD LVCMOS33 } [get_ports { input2[2] }]; #
28          IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
29 set_property -dict { PACKAGE_PIN T13     IOSTANDARD LVCMOS33 } [get_ports { input2[3] }]; #
30          IO_L23P_T3_A03_D19_14 Sch=sw[11]

```

```

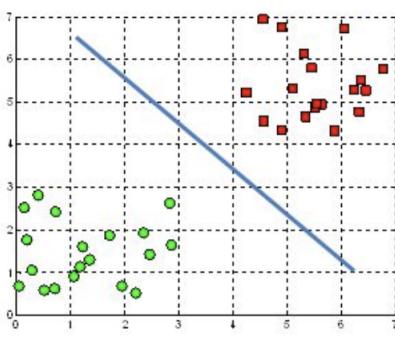
18 set_property -dict { PACKAGE_PIN H6      IOSTANDARD LVCMOS33 } [get_ports { input2[4] }]; #
19   IO_L24P_T3_35 Sch=sw[12]
20 set_property -dict { PACKAGE_PIN U12      IOSTANDARD LVCMOS33 } [get_ports { input2[5] }]; #
21   IO_L20P_T3_A08_D24_14 Sch=sw[13]
22 set_property -dict { PACKAGE_PIN U11      IOSTANDARD LVCMOS33 } [get_ports { input2[6] }]; #
23   IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
24 set_property -dict { PACKAGE_PIN V10      IOSTANDARD LVCMOS33 } [get_ports { input2[7] }]; #
25   IO_L21P_T3_DQS_14 Sch=sw[15]
26
27 ## LEDs
28 set_property -dict { PACKAGE_PIN H17      IOSTANDARD LVCMOS33 } [get_ports { output[0] }]; #
29   IO_L18P_T2_A24_15 Sch=led[0]
30 set_property -dict { PACKAGE_PIN K15      IOSTANDARD LVCMOS33 } [get_ports { output[1] }]; #
31   IO_L24P_T3_RS1_15 Sch=led[1]
32 set_property -dict { PACKAGE_PIN J13      IOSTANDARD LVCMOS33 } [get_ports { output[2] }]; #
33   IO_L17N_T2_A25_15 Sch=led[2]
34 set_property -dict { PACKAGE_PIN N14      IOSTANDARD LVCMOS33 } [get_ports { output[3] }]; #
35   IO_L8P_T1_D11_14 Sch=led[3]
36 set_property -dict { PACKAGE_PIN R18      IOSTANDARD LVCMOS33 } [get_ports { output[4] }]; #
37   IO_L7P_T1_D09_14 Sch=led[4]
38 set_property -dict { PACKAGE_PIN V17      IOSTANDARD LVCMOS33 } [get_ports { output[5] }]; #
39   IO_L18N_T2_A11_D27_14 Sch=led[5]
40 set_property -dict { PACKAGE_PIN U17      IOSTANDARD LVCMOS33 } [get_ports { output[6] }]; #
41   IO_L17P_T2_A14_D30_14 Sch=led[6]
42 set_property -dict { PACKAGE_PIN U16      IOSTANDARD LVCMOS33 } [get_ports { output[7] }]; #
43   IO_L18P_T2_A12_D28_14 Sch=led[7]
44 set_property -dict { PACKAGE_PIN V16      IOSTANDARD LVCMOS33 } [get_ports { output[8] }]; #
45   IO_L16N_T2_A15_D31_14 Sch=led[8]
46 set_property -dict { PACKAGE_PIN T15      IOSTANDARD LVCMOS33 } [get_ports { output[9] }]; #
47   IO_L14N_T2_SRCC_14 Sch=led[9]
48 set_property -dict { PACKAGE_PIN U14      IOSTANDARD LVCMOS33 } [get_ports { output[10] }]; #
49   IO_L22P_T3_A05_D21_14 Sch=led[10]
50 set_property -dict { PACKAGE_PIN T16      IOSTANDARD LVCMOS33 } [get_ports { output[11] }]; #
51   IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
52 set_property -dict { PACKAGE_PIN V15      IOSTANDARD LVCMOS33 } [get_ports { output[12] }]; #
53   IO_L16P_T2_CSI_B_14 Sch=led[12]
54 set_property -dict { PACKAGE_PIN V14      IOSTANDARD LVCMOS33 } [get_ports { output[13] }]; #
55   IO_L22N_T3_A04_D20_14 Sch=led[13]
56 set_property -dict { PACKAGE_PIN V12      IOSTANDARD LVCMOS33 } [get_ports { output[14] }]; #
57   IO_L20N_T3_A07_D23_14 Sch=led[14]
58 set_property -dict { PACKAGE_PIN V11      IOSTANDARD LVCMOS33 } [get_ports { output[15] }]; #
59   IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]
60
61 ##Buttons
62 set_property -dict { PACKAGE_PIN N17      IOSTANDARD LVCMOS33 } [get_ports { start }]; #
63   IO_L9P_T1_DQS_14 Sch=btnc
64 set_property -dict { PACKAGE_PIN M18      IOSTANDARD LVCMOS33 } [get_ports { reset }]; #
65   IO_L4N_T0_D05_14 Sch=btぬ

```

12 Esercizio libero: Rete neurale

L'intelligenza artificiale è una disciplina che studia se e in che modo si possano realizzare sistemi informatici intelligenti in grado di simulare la capacità e il comportamento del pensiero umano. Però Intelligenza Artificiale è un termine ombrello sotto cui ricadono diversi metodi, tecniche e approcci per emulare il comportamento umano. Una di queste tecniche è il Machine learning in cui vengono utilizzati metodi statistici per migliorare la performance di un algoritmo nell'identificare pattern nei dati, quindi è essenzialmente un algoritmo che permette ad una macchina di imparare da degli esempi. Un particolare algoritmo di Machine Learning sono le Artificial Neural Networks (ANN) che si basano sul modello biologico del cervello umano, quindi utilizzano al loro interno dei neuroni artificiali. Le ANN hanno bisogno di una fase di addestramento sui dati, durante la quale apprendono i pattern da riconoscere, per poi essere in grado di identificare correttamente ciò per cui sono state create. Nel Machine Learning i dati forniti alla rete neurale non sono quelli originale, ma sono modificati da un esperto di dominio (Domain expert) in modo da estrarre delle feature particolari che semplificano la fase di addestramento e le prestazioni della rete neurale. Estratte le feature dall'insieme dei dati necessari all'addestramento (anche detti training dataset) questi vengono posti in ingresso alla ANN, che tramite tecniche particolari di apprendimento (come la gradient descent) sono in grado, modificando i parametri della rete, di migliorare gradualmente le prestazioni nell'identificazione. Questo processo può essere schematizzato

A hyperplane in \mathbb{R}^2 is a line



A hyperplane in \mathbb{R}^3 is a plane

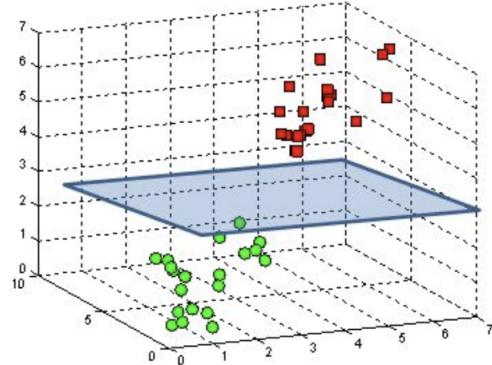


Figura 118: Iperpiani (in rosso) con 2 e 3 feature e 2 classi di appartenenza

facilmente, in quanto la rete neurale, durante la fase di addestramento, cerca di creare un buon iper-piano all'interno dello spazio delle feature, per separare "al meglio" i dati della

fase di addestramento in classi distinte che corrispondono ciascuna a delle "risposte" della rete ad un dato di ingresso. In figura 118 vediamo un esempio, notiamo come più feature vengano estratte per far apprendere la rete, più le dimensioni dello spazio in cui trovare l'iperpiano aumentano. Dopo aver completato l'addestramento la rete sarà in grado di, con dei dati di ingresso, di fornire l'identificazione corretta dei pattern e fornire il risultato atteso in output. Naturalmente, essendo le ANN un modello statistico i risultati della rete sono delle probabilità, che vanno associate alle etichette di output. L'etichetta con la probabilità più alta è il risultato della rete, ma molto spesso assumo molta importanza anche le risposte con probabilità appena più bassa, di solito infatti (dipendentemente dal problema in esame e dalle classi di uscita) si considera l'insieme di etichette con le probabilità più alte tra tutte quelle in uscita dalla rete.

Dopo quest'introduzione generale alle ANN vediamo in particolare come funziona un singolo neurone. In figura 119 vediamo un schematizzazione del formalismo matematico che permette di simulare il neurone dei mammiferi. Per il singolo neurone i valori in input vengono pesati utilizzando i pesi w_i , che sono dei parametri appresi durante la fase di addestramento; effettuata la pesatura di ogni input con i pesi corrispondenti, i risultati vengono sommati tra loro e il valore ottenuto viene messo in ingresso ad una funzione non lineare, detta funzione di attivazione, che, utilizzando una soglia, azzera o assegna un certo valore (che dipende dalla funzione utilizzata) all'input. La soglia utilizzata, anche detta bias, può anche essere sommata alla sommatoria degli input pesati, in modo che la funzione non lineare possa essere centrata sull'origine degli assi. Tutto il formalismo è derivato da considerazioni biologiche fatte attraverso esperimenti sui mammiferi.

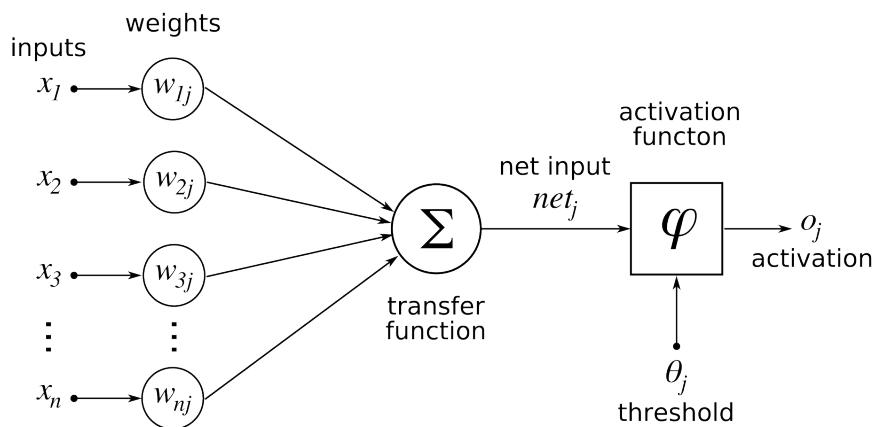


Figura 119: Neurone artificiale

I neuroni sono poi sistemati in strutture che li interconnettono tra loro. Uno schema

molto utilizzato di interconnessione è il feed-forward in cui l'informazione viaggia in una sola direzione ed i neuroni sono organizzati in layer. Un'input layer su cui si presentano i dati in ingresso, un output layer su cui viene presentato il risultato delle rete e degli hidden layer necessari per elaborare correttamente l'informazione di ingresso. In figura 120 vi è un esempio di una semplice rete neurale con 3 neuroni in input, 4 nell'hidden layer e 2 neuroni in output. L'informazione scorre dal layer di input a quello di output e notiamo come ogni neurone di un layer sia solamente connesso con i neuroni del layer precedente. In particolare in figura i layer sono fully-connected, cioè ogni neurone di un layer è connesso con tutti i neuroni del layer precedente. Ogni collegamento tra neuroni ha un peso e ogni neurone ha il proprio bias, che saranno quelli utilizzati da ogni neurone per svolgere il calcolo sopra descritto.

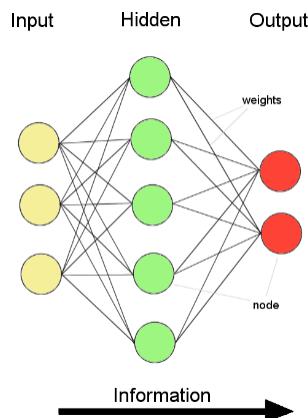


Figura 120: Esempio di semplice rete neurale

12.1 Progettazione in VHDL

Traccia

Progettare, implementare in VHDL e testare mediante simulazione (e, optionalmente, mediante sintesi su board), un sistema le cui specifiche siano definite dallo studente e rientrino in una delle seguenti tipologie: Progetto di sistemi che assolvono a specifici compiti noti (Esempio: si potrebbe pensare di implementare una specifica macchina aritmetica non trattata a lezione, una funzione crittografica, una rete neurale, ecc.)

12.1.1 Schematici

Abbiamo quindi deciso di implementare una ANN per il nostro progetto libero. In particolare abbiamo utilizzato un particolare dataset, "Statlog (Shuttle) Data Set", per addestrare una rete a fornire lo stato dello shuttle in base alla rivelazioni di 9 sensori presenti in esso. In particolare gli stati possibili sono 7 e sono:

1. Rad Flow
2. Fpv Close
3. Fpv Open
4. High
5. Bypass
6. Bpv Close
7. Bpv Open

Quindi inseriremo le misurazioni dei 9 sensori in ingresso ed in uscita otterremo lo stato dello shuttle corrispondente. In figura 121 un estratto del dataset utilizzato.

	A	B	C	D	E	F	G	H	I	J	Status
1	Sensor 1	Sensor 2	Sensor 3	Sensor 4	Sensor 5	Sensor 6	Sensor 7	Sensor 8	Sensor 9		
2	59	125	112	129	88	131	125	164	155	2	
3	72	124	141	129	77	131	140	182	169	4	
4	67	124	122	129	98	131	128	157	147	1	
5	26	124	110	129	88	131	147	164	149	1	
6	26	124	116	129	91	131	152	163	147	1	
7	149	124	133	129	79	131	85	178	179	5	
8	75	124	120	129	75	131	122	180	171	4	
9	72	124	147	129	99	131	147	161	147	1	
10	67	124	112	129	88	131	118	164	156	4	
11	26	124	159	129	88	131	187	174	149	1	
12	26	124	114	129	82	131	150	171	156	1	
13	46	124	126	129	96	131	143	160	146	1	
14	28	124	112	129	92	131	145	160	146	1	
15	26	124	114	129	76	131	148	178	162	1	
16	36	124	157	129	92	131	178	169	147	1	
17	36	124	135	129	92	131	160	165	147	1	
18	52	124	128	129	96	131	143	160	146	1	
19	57	124	116	129	96	131	130	158	147	1	
20	57	124	122	129	97	131	135	158	146	1	
21	26	116	167	129	92	131	193	173	146	1	
22	44	124	112	129	95	131	135	158	146	1	
23	203	124	173	129	105	131	82	160	161	5	
24	26	124	110	129	88	131	147	164	149	1	
25	57	124	157	129	97	131	167	165	146	1	
26	46	124	131	129	95	131	152	162	146	1	
27	26	124	118	129	76	131	152	178	162	1	
28	72	124	153	129	98	131	150	163	148	4	
29	72	124	129	130	99	131	132	158	146	1	

Figura 121: Estratto del dataset utilizzato

Dato il dataset di partenza, abbiamo utilizzato un modello di rete molto semplice. In figura 122 vi è la rete utilizzata. Sono presenti:

1. Un input layer, che presenta 9 neuroni, corrispondenti alle 9 misurazioni dei 9 sensori

2. Un hidden layer, di 10 neuroni necessario per permettere un corretto apprendimento della rete, questo layer è fully-connected
3. Un output layer, che presenta 7 neuroni corrispondenti ai 7 possibili stati dello shuttle

Lo stato associato al neurone con il valore più alto tra tutti quelli nell'output layer sarà quello predetto dalla rete date le misurazioni in ingresso.

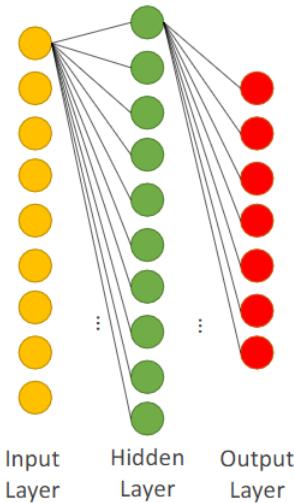


Figura 122: Rete utilizzata

Per questa rete neurale la funzione di attivazione prevista per ciascun nodo è la ReLU, una linerità molto utilizzata nel campo del Machine Learning.

In particolare abbiamo utilizzato dei parametri di un addestramento della rete presentata sul task di classificazione dello stato dello shuttle, quindi abbiamo già a disposizione i pesi di ciascun collegamento ed i bias associato a ciascun neurone della rete. In figura 123 un esempio di alcuni valori dei pesi della rete.

	Input 1	Input 2	Input 3	Input 4	Input 5
Neurone1	27	-46	9	15	-15
Neurone2	61	54	-46	-50	-29
Neurone3	-45	37	29	-39	18
Neurone4	45	-3	47	-53	31
Neurone5	17	26	16	-54	-40
Neurone6	-78	-34	19	42	-8

Figura 123: Estratto dei valori dei pesi

Avendo la rete già addestrata ci poniamo come obiettivo quello di utilizzare la rete neurale per svolgere delle inferenze. Quindi vogliamo poter porre in ingresso le misurazioni ed

ottenere lo stato dello shuttle in uscite. Per fare ciò abbiamo bisogno di implementare l'intera struttura della rete. Il componente con più intelligenza all'interno è il singolo neurone, per implementarlo abbiamo bisogno di implementare l'operazione che viene fatta al suo interno, che ripostiamo di seguito:

$$y_i = g(b + \sum_{i=0}^n x_i * w_i) \quad (2)$$

Quindi sarà necessario un sommatore, un moltiplicatore e un componente che implementi una funzione g , che, come precedentemente accennato, è la ReLU. Inoltre per permettere il corretto funzionamento del neurone saranno necessarie anche:

- 2 mux, per permettere di selezione un solo input alla volta e per selezionare la somma del bias
- Un registro per mantenere la somma parziale
- Un contatore, per contare e controllare di compiere il numero di operazioni corretto
- Una ROM, per mantenere i pesi di ciascun collegamento con i neuroni
- Un unità di controllo, per gestire tutto il funzionamento del nodo

Moltiplicatore e addizionatore sono già stati utilizzati e descritti nella sezione 11, anche registri, contatore e ROM sono già stati descritti e utilizzati nei precedenti esercizi. Per tutti questi componenti utilizzeremo le implementazioni già trattate, modificando solamente le dimensioni dei dati in input e output se necessario. In particolare distinguamo neuroni dell'hidden layer e del output layer in quanto il numero di collegami ad altri neuroni ed il numero di bit necessari per le operazioni sono diversi tra i due layer. Una volta definiti i neuroni sarà semplice definire hidden layer e output layer e poi unirli per ottenere la rete neurale completa. In particolare, utilizzando questa nostra implementazione faremo un modo che i singoli neuroni in un layer lavorino in modo parallelo, aumentando la velocità di risposta della rete. Mentre i due layer devono lavorare necessariamente sequenzialmente, infatti l'output layer necessita dei risultati dell'hidden layer per svolgere le sue elaborazioni

ReLU

La ReLU (Rectified Linear Unit) è una semplice funzione non lineare, che associa ad un ingresso intero il suo valore se positivo, mentre associa 0 se l'ingresso è negativo.

$$y = \max(0, x)$$

Quindi presenta un singolo ingresso e una singola uscita entrambi della stessa dimensione. In figura 124 la funzione matematica e lo schematico.

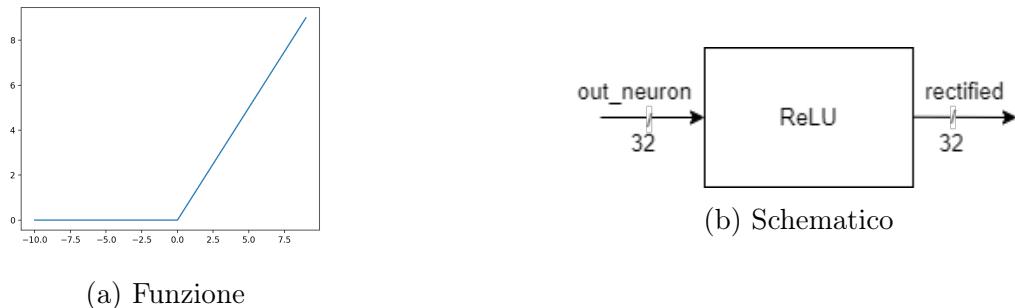


Figura 124: Funzione e schematico della ReLU

Unità di controllo dell'hidden neuron

L'unità di controllo dovrà essere in grado di controllare tutti gli elementi nel sistema. In ingresso ha il segnale di start per iniziale le elaborazioni, il segnale di mult_stop che indica la terminazione di una moltiplicazione e il valore del contatore per determinare quando le operazioni saranno terminate. Le uscite sono: il segnale di read per la ROM, il segnale di inizio per il moltiplicatore il segnale di selezione per un MUX, il segnale di load del registro, il segnale di conteggio per il contatore, un segnale di reset di tutte le componenti ed un segnale di end_op che indica la fine di tutte le operazioni del neurone. In figura 125 l'automa dell'unità di controllo e lo schematico. E' da tenere presente che tutti i segnali in uscita vengono azzerati sempre, a meno che non sia indicato esplicitamente. Nel successivo paragrafo verrà fornito il contesto in cui opera l'UC e l'automa verrà chiarito.

Neurone dell'hidden layer

A questo punto possiamo definire l'intera struttura di un neurone dell'hidden layer componendo opportunamente i componenti citati. In figura 126 lo schematico completo. Gli ingressi sono tutti inseriti in un mux, mentre i pesi del nodo sono posizionati in una ROM, la selezione del mux e l'indirizzo da cui leggere per la ROM sono forniti dall'uscita di conteggio di un contatore modulo 9 (perchè gli ingressi sono 9). Per il singolo neurone abbiamo deciso di implementare le operazioni di prodotto e somma in modo sequenziale, il risultato parziale di un prodotto e successiva somma viene salvato in registro, per la somma successiva si utilizzerà il valore di questo registro come operando. In questo modo implementiamo la

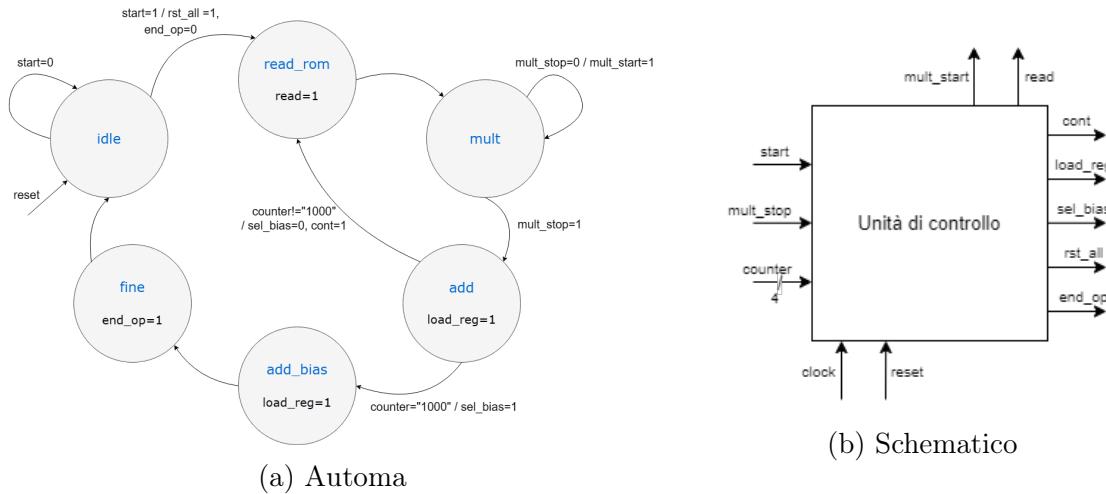


Figura 125: Automa e schematico dell'unità di controllo

sommatoria del prodotto senza complicare troppo la struttura del nodo. Peso e input scelti tramite il contatore entrano nel moltiplicatore che inizia l'operazione quando la UC dà lo start. A fine operazione viene fatta la somma e la UC può recuperare il valore della somma inserendolo nel registro (tramite il load). Questo viene ripetuto per nove volte, infine viene effettuata un'ultima somma, stavolta però si sommano valore del registro e il bias, che vengono selezionati usando altri due multiplexer. A questo punto rimane solo da applicare la funzione ReLU e successivamente viene alzato il segnale di end_op dalla UC. Nello schematico alcuni collegami sono effettuati con l'operatore " $=>$ " per non rendere più chiaro il tutto.

Hidden layer

L'hidden layer è realizzato semplicemente utilizzando 10 neuroni, ognuno prende in ingresso tutti gli input e lo start, mentre l'uscita del layer sono: tanti bit di end_op e tanti valori (su 32 bit) quanti sono i neuroni nel layer. In figura 127 lo schematico.

Unità di controllo dell'output neuron

L'unità di controllo dell'output neuron è praticamente la stessa dell'hidden layer, l'unica differenza riguarda l'ingresso di start. Infatti un neurone dell'output layer deve aspettare che tutti i neuroni dell'hidden layer abbiano terminato, quindi l'ingresso di start è un vettore di 10 elementi. In figura 128 l'automa dell'unità di controllo e lo schematico. E' da tenere presente che tutti i segnali in uscita vengono azzerati sempre, a meno che non sia indica-

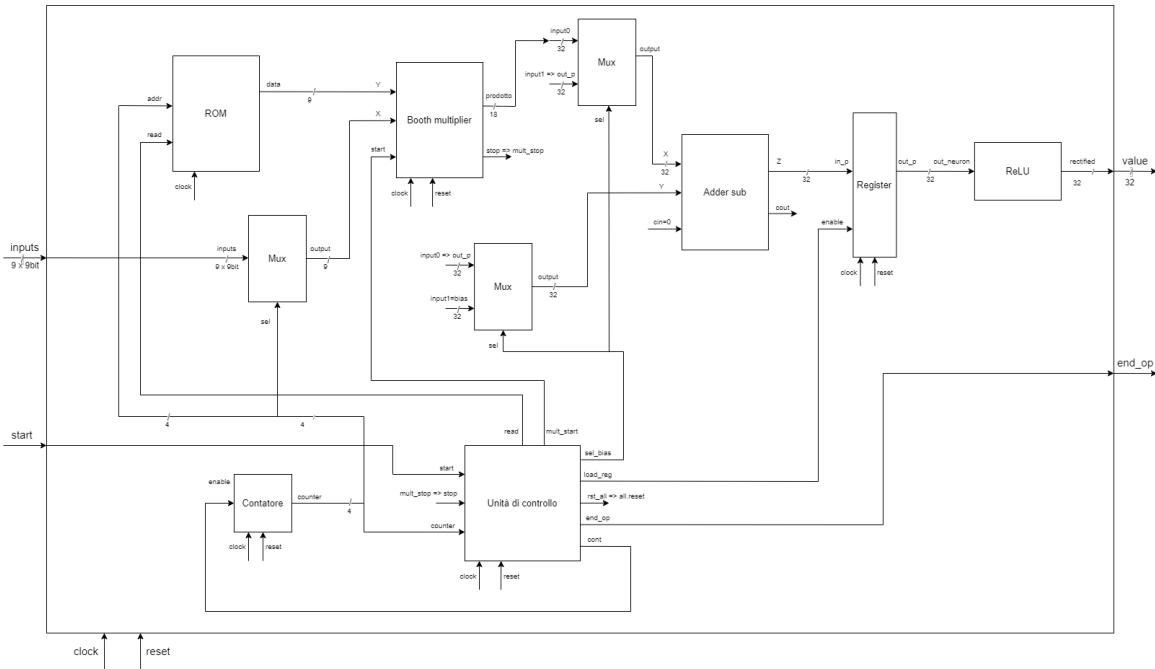


Figura 126: Hidden Neuron

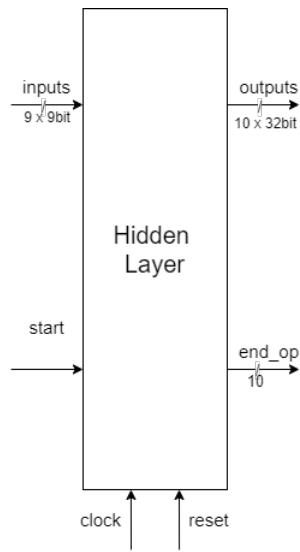


Figura 127: Hidden layer

to esplicitamente. Notiamo come l'automa rimanga bloccato nello stato fine, solo quando avviene il reset torna nello stato di idle, questo verrà spiegato in seguito. Nel successivo paragrafo verrà fornito il contesto in cui opera l'UC e l'automa verra chiarito.

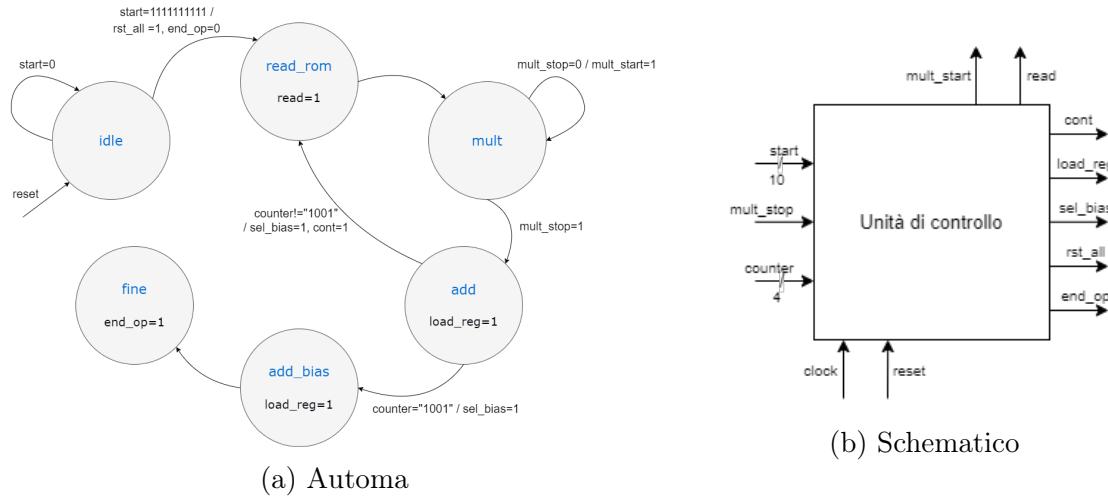


Figura 128: Automa e schematico dell'unità di controllo

Neurone dell'output layer

Anche il neurone dell'output layer è molto simile a quello dell'hidden layer. Gli unici cambiamenti effettuati riguardano il dimensionamento di moltiplicatore, addizionare, registro e i conseguenti collegamenti. Questo è stato fatto perché i valori in input che vede il neurone sono su 32 bit, in quanto uscite dei neuroni dell'hidden layer. Di conseguenza l'uscita del neurone sarà su 64 bit. In figura 129 lo schematico.

Output layer

L'output layer è realizzato semplicemente utilizzando 7 neuroni, in quanto si associa ad ogni stato dello shuttle un valore in uscita. Ogni neurone prende in ingresso tutti gli input e il vettore di start, mentre l'uscita del layer sono: tanti bit di end_op e tanti valori (su 64 bit) quanti sono i neuroni nel layer. In figura 130 lo schematico.

Rete Neurale

La Rete Neurale completa è stata realizzata interconnettendo, come visto in figura 122, i layer realizzati con gli input ed output della rete.

Abbiamo deciso di lasciare in output alla rete l'uscita di tutti i neuroni dell'output layer. Questo perchè solitamente, non si presenta solamente la risposta più probabile della rete (cioè, in questo caso, quella con il valore più alto) ma si presentano le top 2 o 3 opzioni (le top con valore più alto, che corrispondono agli stati più probabili) fornite della rete, questo perchè possiamo renderci conto di quanto la rete sia sicura di una certa risposta rispetto le

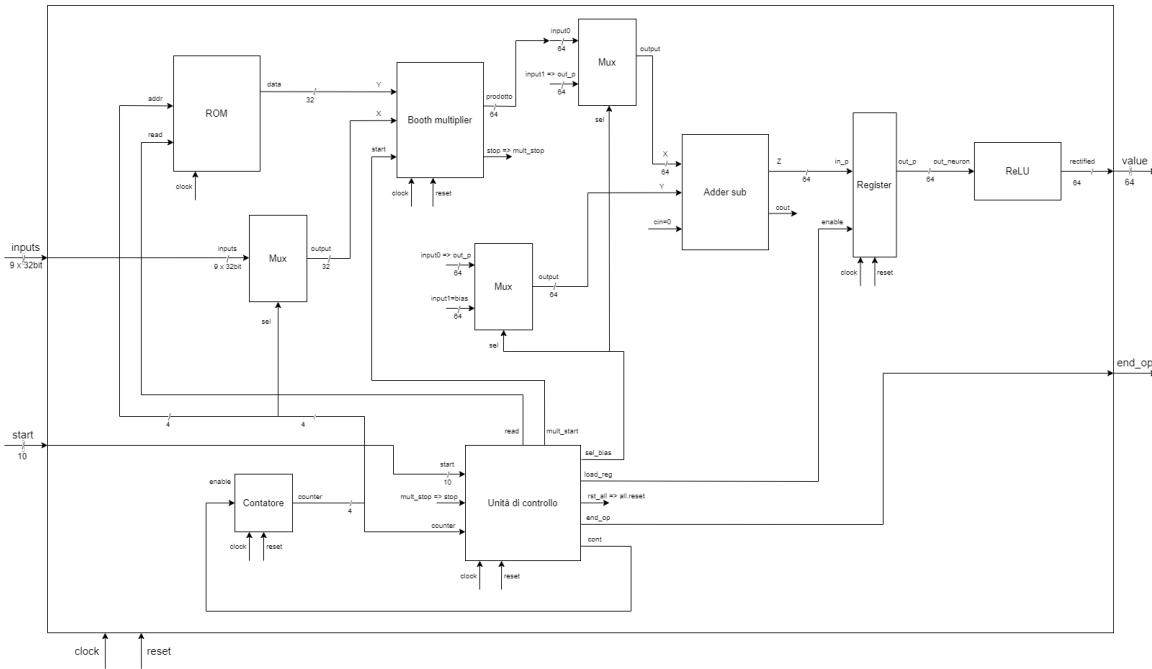


Figura 129: Output Neuron

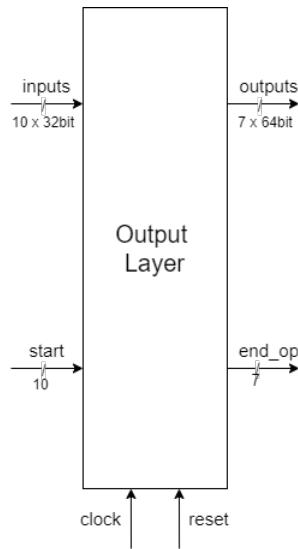


Figura 130: Output layer

altre opzioni osservando i valori associati alle etichette di output (nel nostro caso gli stati dello shuttle). Nel caso sia necessario è possibile inserire un componente che, valutando le uscite della rete, fornisce un singolo stato in output, che corrisponde al valore più alto in uscita. Un componente di questo tipo può essere semplicemente realizzato usando un comparatore, un contatore e un'unità di controllo che gestisca le comparazioni tra le uscite dei neuroni e

un massimo temporaneo che viene aggiornato quando un'uscita è più grande.

In figura 131 lo schematico della rete completa.

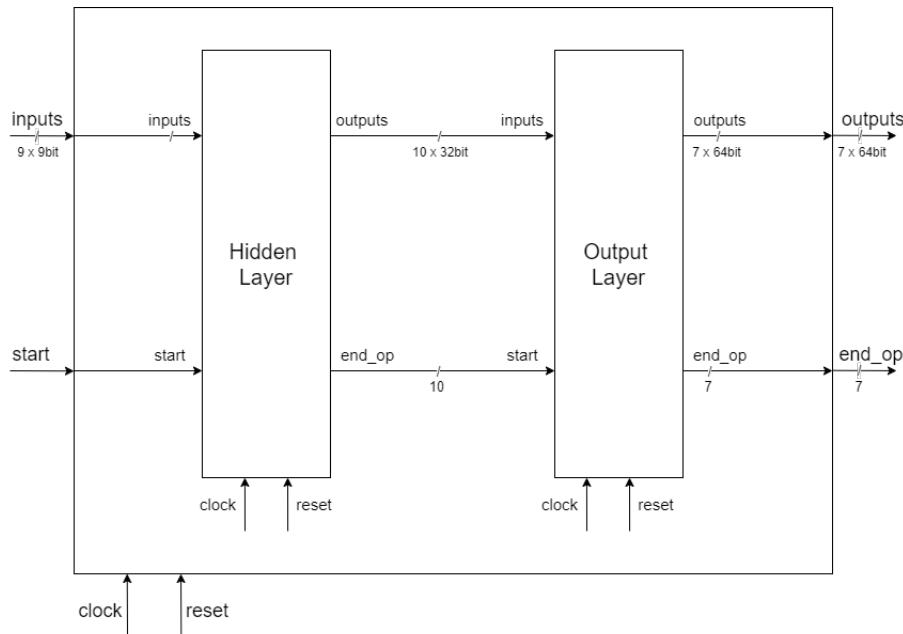


Figura 131: Rete neurale

12.1.2 Codice VHDL

Nell'implementazione in VHDL, per definire facilmente alcuni segnali, abbiamo deciso di scrivere dei package che includessero i tipi necessari e che ci permettessero di usarli in qualsiasi file del progetto includendo semplicemente un package. I package sono stati inoltre utilizzati per mantenere, come costanti, i valori di i pesi e bias di ciascun nodo in binario. Questi valori saranno passati durante il mapping dei nodi utilizzando un generic, in modo che ogni nodo possa avere la propria ROM con i valori corretti. In questo modo evitiamo di dover inserire manualmente i valori di ciascun ROM in ciascun nodo e possiamo mappare i nodi in un layer utilizzando un for-generate, molto più semplice e veloce che farlo nodo per nodo. Abbiamo differenziato due package a seconda del layer in cui ci troviamo in modo da evitare confusione tra i tipi e le costanti relative ai pesi e bias. Di seguito il codice del package relativo ai tipi usati nell'hidden layer.

```

1 package type_package is
2
3 type hidden_input is array (0 to 9) of std_logic_vector(8 downto 0);
4 type hidden_output is array (0 to 9) of std_logic_vector(31 downto 0);
  
```

```

5 type hidden_bias is array (0 to 9) of std_logic_vector(31 downto 0);
6
7 type rom_type is array (0 to 8) of std_logic_vector(8 downto 0);
8 type array_rom_type is array (0 to 9) of rom_type;
9
10 constant neurone_hidden_0 : rom_type := (
11 "000011011", "111010010", "000001001", "000001111", "111110001", "111011111",
12 "000010100", "000011110", "111100111"
13 );
14
15 constant neurone_hidden_1 : rom_type := (
16 "000111101", "000110110", "111010010", "111001110", "111100011", "000010001",
17 "111011111", "111101111", "000110001"
18 );
19
20 constant neurone_hidden_2 : rom_type := (
21 "111010011", "000100101", "000011101", "111011001", "000010010", "000000011",
22 "111110011", "000010100", "111111010"
23 );
24
25 constant neurone_hidden_3 : rom_type := (
26 "000101101", "111111101", "000101111", "111001011", "000011111", "111011000",
27 "000010100", "111100010", "000000100"
28 );
29
30 constant neurone_hidden_4 : rom_type := (
31 "000010001", "000011010", "000010000", "111001010", "111011000", "111001101",
32 "000100100", "000100110", "000100101"
33 );
34
35 constant neurone_hidden_5 : rom_type := (
36 "110110010", "111011110", "000010011", "000101010", "111111000", "000000101",
37 "000111000", "111111001", "111110011"
38 );
39
40 constant neurone_hidden_6 : rom_type := (
41 "111011100", "111101100", "000010001", "000001011", "111101111",
42 "000101001", "111111011", "111010011", "000010010"
43 );
44
45 constant neurone_hidden_7 : rom_type := (
46 "000000100", "000101001", "111101011", "111101101", "000001110", "111101010",
47 "111010110", "111100110", "000001101"
48 );
49
50 constant neurone_hidden_8 : rom_type := (
51 "000101011", "000011000", "000001000", "000101001", "000001111", "000000000",
52 "111001111", "000000100", "111101001"
53 );
54
55 constant neurone_hidden_9 : rom_type := (
56 "000110101", "001000001", "111100101", "111011010", "000100110", "111101111",
57 "111110011", "000001101", "000010111"
58 );
59
60 constant array_hidden : array_rom_type := (
61 neurone_hidden_0,
62 neurone_hidden_1,
63 neurone_hidden_2,
64 neurone_hidden_3,
```

```

64 neurone_hidden_4 ,
65 neurone_hidden_5 ,
66 neurone_hidden_6 ,
67 neurone_hidden_7 ,
68 neurone_hidden_8 ,
69 neurone_hidden_9
70 );
71
72 constant value_hidden_bias : hidden_bias:= (
73 "11111111111111111111111111000",
74 "0000000000000000000000000000010",
75 "111111111111111111111111110010",
76 "111111111111111111111111111110",
77 "1111111111111111111111111111000",
78 "0000000000000000000000000000010",
79 "11111111111111111111111111110101",
80 "11111111111111111111111111111011",
81 "111111111111111111111111111110111",
82 "00000000000000000000000000000110"
83 );
84
85 end package type_package;
86
87 package body type_package is
88 end package body type_package;

```

Di seguito, invece, il codice del package relativo ai tipi usati nell'output layer:

```

1 package type_package_output is
2
3 type uscite is array (0 to 6) of std_logic_vector(63 downto 0);
4 type output_bias is array (0 to 6) of std_logic_vector(6 downto 0);
5
6 type rom_type_output is array (0 to 9) of std_logic_vector(31 downto 0);
7 type array_rom_type_output is array (0 to 6) of rom_type_output;
8
9 constant neurone_output_0 : rom_type_output := (
10 "0000000000000000000000000000000000",
11 "0000000000000000000000000000000000",
12 "1111111111111101001011101010000",
13 "0000000000000000000000000000000000",
14 "1111111111111100001000101110000",
15 "000000000000000011001011010000100",
16 "0000000000000000000000000000000000",
17 "0000000000000000000000000000000000",
18 "111111111111110100001000101000",
19 "000000000000011001001111111100"
20 );
21
22 constant neurone_output_1 : rom_type_output := (
23 "0000000000000000000000000000000000",
24 "0000000000000000000000000000000000",
25 "0000000000000010001000110100000",
26 "0000000000000000000000000000000000",
27 "00000000000000100010100110011100",
28 "1111111111111101111111001100011",
29 "0000000000000000000000000000000000",
30 "0000000000000000000000000000000000"

```

```

31 "111111111111101110001100001110",
32 "00000000000000010101101101100100"
33 );
34
35 constant neurone_output_2 : rom_type_output := (
36 "00000000000000000000000000000000",
37 "00000000000000000000000000000000",
38 "1111111111111011010101100000",
39 "00000000000000000000000000000000",
40 "111111111111100101010000110110",
41 "00000000000000011010010001010001",
42 "00000000000000000000000000000000",
43 "00000000000000000000000000000000",
44 "111111111111101001001000010001",
45 "0000000000000100110011111011010"
46 );
47
48 constant neurone_output_3 : rom_type_output := (
49 "00000000000000000000000000000000",
50 "00000000000000000000000000000000",
51 "00000000000000000000000000000000",
52 "00000000000000000000000000000000",
53 "00000000000000000000000000000000",
54 "111111111111100010010101110011",
55 "00000000000000000000000000000000",
56 "00000000000000000000000000000000",
57 "111111111111100110001001010111",
58 "00000000000000001001110111110110"
59 );
60
61 constant neurone_output_4 : rom_type_output := (
62 "00000000000000000000000000000000",
63 "00000000000000000000000000000000",
64 "00000000000000000000000000000000",
65 "00000000000000000000000000000000",
66 "1111111111110100011001101010010",
67 "00000000000000000000000000000000",
68 "00000000000000000000000000000000",
69 "00000000000000000000000000000000",
70 "111111111111100110001001010111",
71 "00000000000000001001110111110110"
72 );
73
74 constant neurone_output_5 : rom_type_output := (
75 "00000000000000000000000000000000",
76 "00000000000000000000000000000000",
77 "00000000000000000000000000000000",
78 "00000000000000000000000000000000",
79 "00000000000000000000000000000000",
80 "11111111111101110111100110111",
81 "00000000000000000000000000000000",
82 "00000000000000000000000000000000",
83 "111111111111100100001010000100",
84 "0000000000000000111110010110000"
85 );
86
87 constant neurone_output_6 : rom_type_output := (
88 "00000000000000000000000000000000",
89 "00000000000000000000000000000000",
90 "11111111111110101111001010000"

```

Vediamo ora il codice che implementa la rete neurale descritta nella sezione precedente.

ReLU

L'implementazione della ReLU segue perfettamente la sua definizione matematica. In uscita abbiamo 0 se il valore è negativo, cioè se il suo bit più significativo è 1 (rappresentazione in complemento a 2), mentre abbiamo il valore in ingresso se il valore è positivo, cioè bit più significativo uguale a 0. Abbiamo utilizzato il generic per far lavorare il componente con segnali di dimensioni diverse in ingresso. Di seguito il codice,

```
1 entity ReLU is
2 generic (
3     bits          : integer
4 );
5 port (
6     out_neuron    : in std_logic_vector(bits-1 downto 0);
7     rectified     : out std_logic_vector(bits-1 downto 0)
8 );
9 end ReLU;
```

```

11 architecture dataflow of ReLU is
12 begin
13 rectified <= (others => '0') when out_neuron(bits-1) = '1' else
14           out_neuron when out_neuron(bits-1) = '0' else
15           out_neuron;
16
17 end dataflow;

```

Unità di controllo dell'hidden layer

Nella precedente sezione abbiamo presentato il funzionamento dell'unità di controllo allegando anche l'automa. L'implementazione è la classica a due process per implementare gli automi: un process combinatorio e un process che implementa il registro per il cambiamento di stato. Di seguito riportiamo il codice.

```

1 entity uc_neurone_hidden is port (
2     clock      : in std_logic;
3     reset      : in std_logic;
4     start      : in std_logic;
5     mult_stop  : in std_logic;
6     counter    : in std_logic_vector(3 downto 0);
7     mult_start : out std_logic;
8     cont       : out std_logic;
9     read       : out std_logic;
10    load_reg   : out std_logic;
11    sel_bias   : out std_logic;
12    rst_all    : out std_logic;
13    end_op     : out std_logic
14 );
15 end uc_neurone_hidden;
16
17 architecture Behavioral of uc_neurone_hidden is
18 type stato is (idle, read_rom, mult, add, add_bias, fine);
19 signal curr_state : stato := idle;
20 signal next_state : stato := idle;
21 begin
22
23 registri : process (clock)
24 begin
25 if rising_edge(clock) then
26
27     if reset='1' then
28         curr_state <= idle;
29     else
30         curr_state <= next_state;
31     end if;
32 end if;
33 end process;
34
35 calcolo_stato : process( curr_state, start, mult_stop)
36 begin
37 rst_all <= '0';
38 cont <= '0';
39 read <= '0';

```

```

40 mult_start <='0';
41 load_reg <= '0';
42 sel_bias <= '0';
43
44 case curr_state is
45   when idle          =>
46     if (start = '1') then
47       end_op <= '0';
48       rst_all <= '1';
49       next_state <= read_rom;
50     else
51       next_state <= idle;
52     end if;
53   when read_rom      =>
54     read <= '1';
55     next_state <= mult;
56   when mult          =>
57     if (mult_stop = '1') then
58       next_state <= add;
59     else
60       mult_start <='1';
61       next_state <= mult;
62     end if;
63   when add           =>
64     load_reg <= '1';
65     if (counter = "1000") then
66       sel_bias <= '1';
67       next_state <= add_bias;
68     else
69       cont <= '1';
70       next_state <= read_rom;
71     end if;
72   when add_bias      =>
73     load_reg <= '1';
74     next_state <= fine;
75   when fine          =>
76     end_op <= '1';
77     next_state <= idle;
78
79 end case;
80 end process;
81 end Behavioral;

```

Neurone dell'hidden layer

Il neurone è realizzato semplicemente tramite composizione dei componenti seguendo lo schematico visto nella sezione precedente. Notiamo come mux, registro, ReLu e ROM utilizzino dei generic per il mapping delle porte. Per i primi tre viene utilizzato in modo da definire i bit su cui lavora il componente, mentre l'ultimo, cioè per la ROM, viene utilizzato per passare i valori che saranno memorizzati nella ROM stessa. Nel codice notiamo inoltre un'altra particolarità, infatti l'adder somma 32 bit, mentre il moltiplicatore ne ha in uscita 18, quindi è stato necessario effettuare un padding con tutti 0 o 1, a seconda del segno del valore

in uscita dal moltiplicatore, prima di effettuare la somma. Abbiamo preferito usare un adder che lavora con 32 bit in modo da evitare di complicare la struttura del neurone per gestire il riporto dell'addizionatore. Notiamo inoltre come il generic della ROM si propaghi anche al neurone, perché i valori verranno assegnati al livello del layer. Il neurone possiede anche il valore del bias in generic, in quanto andrà fornito dal layer. Di seguito il codice.

```

1 entity hidden_neuron is
2 generic (
3     weights : rom_type;
4     bias      : std_logic_vector(31 downto 0)
5 );
6 port (
7     clock      : in std_logic;
8     reset      : in std_logic;
9     start      : in std_logic;
10    input0     : in std_logic_vector(8 downto 0);
11    input1     : in std_logic_vector(8 downto 0);
12    input2     : in std_logic_vector(8 downto 0);
13    input3     : in std_logic_vector(8 downto 0);
14    input4     : in std_logic_vector(8 downto 0);
15    input5     : in std_logic_vector(8 downto 0);
16    input6     : in std_logic_vector(8 downto 0);
17    input7     : in std_logic_vector(8 downto 0);
18    input8     : in std_logic_vector(8 downto 0);
19    end_op     : out std_logic;
20    value      : out std_logic_vector(31 downto 0)
21 );
22 end hidden_neuron;
23
24 architecture structural of hidden_neuron is
25 signal sig_read          : std_logic;
26 signal sig_counter        : std_logic_vector(3 downto 0);
27 signal sig_cont           : std_logic;
28 signal sig_weights        : std_logic_vector(8 downto 0);
29 signal sig_input          : std_logic_vector(8 downto 0);
30 signal sig_mult_start    : std_logic;
31 signal sig_mult_stop     : std_logic;
32 signal sig_prodotto       : std_logic_vector(17 downto 0);
33 signal sig_in_reg         : std_logic_vector(31 downto 0);
34 signal sig_out_reg        : std_logic_vector(31 downto 0);
35 signal sig_adder1         : std_logic_vector(31 downto 0);
36 signal sig_adder2         : std_logic_vector(31 downto 0);
37 signal sig_sel_bias       : std_logic;
38 signal sig_load_r         : std_logic;
39 signal sig_reset          : std_logic;
40
41 signal temp               : std_logic_vector(31 downto 0);
42
43 begin
44
45 controllo : entity work.uc_neurone_hidden port map (
46     clock      => clock,
47     reset      => reset,
48     start      => start,
49     mult_stop   => sig_mult_stop,
50     counter    => sig_counter,

```

```

51     mult_start  => sig_mult_start ,
52     cont        => sig_cont ,
53     read         => sig_read ,
54     load_reg    => sig_load_r ,
55     sel_bias    => sig_sel_bias ,
56     rst_all     => sig_reset ,
57     end_op      => end_op
58 );
59
60 pesi : entity work.ROM
61 generic map (
62     rom_generic => weights
63 )
64 port map (
65     clk  => clock,
66     read => sig_read,
67     addr => sig_counter,
68     data  => sig_weights
69 );
70
71 contatore : entity work.counter_mod9 port map (
72     clock  => clock,
73     reset   => reset or sig_reset ,
74     enable  => sig_cont,
75     counter => sig_counter
76 );
77
78 mux_input : entity work.mux_16_1 port map (
79     input0  => input0,
80     input1  => input1,
81     input2  => input2,
82     input3  => input3,
83     input4  => input4,
84     input5  => input5,
85     input6  => input6,
86     input7  => input7,
87     input8  => input8,
88     s       => sig_counter,
89     output   => sig_input
90 );
91
92 moltiplicatore : entity work.Booth_multiplier port map (
93     clock      => clock,
94     reset      => reset or sig_reset ,
95     start      => sig_mult_start ,
96     Y          => sig_input ,
97     X          => sig_weights ,
98     prodotto   => sig_prodotto ,
99     stop       => sig_mult_stop
100 );
101
102 temp <= "0000000000000000" & sig_prodotto when sig_prodotto(17) = '0' else
103     "1111111111111" & sig_prodotto when sig_prodotto(17) = '1' else
104     (others => '0');
105
106 mux_adder : entity work.mux_2_1
107 generic map (
108     bits  => 32
109 )
110 port map (

```

```

111     input0  => temp,
112     input1  => sig_out_reg,
113     s       => sig_sel_bias,
114     output   => sig_adder1
115 );
116
117 mux_bias : entity work.mux_2_1
118 generic map (
119     bits => 32
120 )
121 port map (
122     input0  => sig_out_reg,
123     input1  => bias,
124     s       => sig_sel_bias,
125     output   => sig_adder2
126 );
127
128 addizionatore : entity work.adder_sub_32 port map (
129     X      => sig_adder1,
130     Y      => sig_adder2,
131     cin    => '0',
132     Z      => sig_in_reg,
133     cout   => open
134 );
135
136 registroo : entity work.registro
137 generic map (
138     bits => 32
139 )
140 port map (
141     clock   => clock,
142     reset   => reset or sig_reset,
143     enable  => sig_load_r,
144     in_p    => sig_in_reg,
145     out_p   => sig_out_reg
146 );
147
148 funz_attivazione : entity work.ReLU
149 generic map (
150     bits => 32
151 )
152 port map (
153     out_neuron  => sig_out_reg,
154     rectified   => value
155 );
156
157 end structural;

```

Hidden layer

L'hidden layer è composto da 10 neuroni hidden che vengono tutti inizializzati tramite un for-generate. Questo è stato possibile grazie ai tipi e alle costanti definite nel package. Infatti qui assegnamo i pesi ed i bias ad ogni neurone. Di seguito il codice.

```
1 entity hidden_layer is port (
```

```

2      clock    : in std_logic;
3      reset    : in std_logic;
4      start    : in std_logic;
5      input0   : in std_logic_vector(8 downto 0);
6      input1   : in std_logic_vector(8 downto 0);
7      input2   : in std_logic_vector(8 downto 0);
8      input3   : in std_logic_vector(8 downto 0);
9      input4   : in std_logic_vector(8 downto 0);
10     input5   : in std_logic_vector(8 downto 0);
11     input6   : in std_logic_vector(8 downto 0);
12     input7   : in std_logic_vector(8 downto 0);
13     input8   : in std_logic_vector(8 downto 0);
14
15     output   : out hidden_output;
16
17     end_op   : out std_logic_vector(9 downto 0)
18 );
19 end hidden_layer;
20
21 architecture structural of hidden_layer is
22
23 component hidden_neuron is
24 generic (
25   weights : rom_type;
26   bias    : std_logic_vector(31 downto 0)
27 );
28 port (
29   clock    : in std_logic;
30   reset    : in std_logic;
31   start    : in std_logic;
32   input0   : in std_logic_vector(8 downto 0);
33   input1   : in std_logic_vector(8 downto 0);
34   input2   : in std_logic_vector(8 downto 0);
35   input3   : in std_logic_vector(8 downto 0);
36   input4   : in std_logic_vector(8 downto 0);
37   input5   : in std_logic_vector(8 downto 0);
38   input6   : in std_logic_vector(8 downto 0);
39   input7   : in std_logic_vector(8 downto 0);
40   input8   : in std_logic_vector(8 downto 0);
41   end_op   : out std_logic;
42   value    : out std_logic_vector(31 downto 0)
43 );
44 end component;
45
46 begin
47
48 neuroni0to9: for i in 0 to 9 generate
49   neurone : hidden_neuron
50   generic map (
51     weights => array_hidden(i),
52     bias    => value_hidden_bias(i)
53   )
54   port map (
55     clock    => clock,
56     reset    => reset,
57     start    => start,
58     input0   => input0,
59     input1   => input1,
60     input2   => input2,
61     input3   => input3,

```

```

62      input4  => input4,
63      input5  => input5,
64      input6  => input6,
65      input7  => input7,
66      input8  => input8,
67      end_op   => end_op(i),
68      value    => output(i)
69  );
70 end generate neuroni0to9;
71
72 end structural;

```

Unità di controllo dell'output layer

Nella precedente sezione abbiamo presentato il funzionamento dell'unità di controllo allegando anche l'automa. L'implementazione è la classica a due process per implementare gli automi: un process combinatorio e un process che implementa il registro per il cambiamento di stato. Di seguito riportiamo il codice.

```

1 entity uc_neurone_output is port (
2     clock      : in std_logic;
3     reset      : in std_logic;
4     start      : in std_logic_vector(9 downto 0);
5     mult_stop  : in std_logic;
6     counter    : in std_logic_vector(3 downto 0);
7     mult_start : out std_logic;
8     cont       : out std_logic;
9     read       : out std_logic;
10    load_reg   : out std_logic;
11    sel_bias   : out std_logic;
12    rst_all    : out std_logic;
13    end_op     : out std_logic
14 );
15 end uc_neurone_output;
16
17 architecture Behavioral of uc_neurone_output is
18 type stato is (idle, read_rom, mult, add, add_bias, fine);
19 signal curr_state : stato := idle;
20 signal next_state : stato := idle;
21 begin
22
23 registri : process (clock)
24 begin
25 if rising_edge(clock) then
26
27     if reset='1' then
28         curr_state <= idle;
29     else
30         curr_state <= next_state;
31     end if;
32 end if;
33 end process;
34
35 calcolo_stato : process(curr_state, start, mult_stop)

```

```

36 begin
37   rst_all <= '0';
38   cont <= '0';
39   read <= '0';
40   mult_start <='0';
41   load_reg <= '0';
42   sel_bias <= '0';
43
44 case curr_state is
45   when idle          =>
46     if (start = "1111111111") then
47       end_op <= '0';
48       rst_all <= '1';
49       next_state <= read_rom;
50     else
51       next_state <= idle;
52     end if;
53   when read_rom      =>
54     read <= '1';
55     next_state <= mult;
56   when mult          =>
57     if (mult_stop = '1') then
58       next_state <= add;
59     else
60       mult_start <='1';
61       next_state <= mult;
62     end if;
63   when add            =>
64     load_reg <= '1';
65     if (counter = "1001") then
66       sel_bias <= '1';
67       next_state <= add_bias;
68     else
69       cont <= '1';
70       next_state <= read_rom;
71     end if;
72   when add_bias       =>
73     load_reg <= '1';
74     next_state <= fine;
75   when fine           =>
76     end_op <= '1';
77
78 end case;
79 end process;
80 end Behavioral;
```

Neurone dell'output layer

Il neurone è realizzato semplicemente tramite composizione dei componenti seguendo lo schematico visto nella sezione precedente. Notiamo come mux, registro, ReLu e ROM utilizzino dei generic per il mapping delle porte. Per i primi tre viene utilizzato in modo da definire i bit su cui lavora il componente, mentre l'ultimo, cioè per la ROM, viene utilizzato per passare i valori che saranno memorizzati nella ROM stessa. Qui non è stato necessario effettuare il padding all'uscita del moltiplicatore, in quanto, avendo sovradimensionato l'adder

dell'hidden layer siamo sicuri che il risultato della somma dei valori in questo layer sia sempre rappresentabile su 64 bit. Quindi basta un moltiplicatore da 32 bit un adder da 64 bit. E' stato invece necessario effettuare il padding fino a 32 bit del bias, in quanto fornito su 7 bit. Notiamo inoltre come il generic della ROM si propaghi anche al neurone, perchè i valori verranno assegnati al livello del layer. Di seguito il codice. Il neurone possiede anche il valore del bias in generic, in quanto andrà fornito dal layer. Di seguito il codice.

```

1 entity output_neuron is
2 generic (
3     weights : rom_type_output;
4     bias     : std_logic_vector(6 downto 0)
5 );
6 port (
7     clock    : in std_logic;
8     reset    : in std_logic;
9     start    : in std_logic_vector(9 downto 0);
10    inputs   : in hidden_output;
11    end_op   : out std_logic;
12    value    : out std_logic_vector(63 downto 0)
13 );
14 end output_neuron;
15
16 architecture structural of output_neuron is
17 signal sig_read          : std_logic;
18 signal sig_counter        : std_logic_vector(3 downto 0);
19 signal sig_cont           : std_logic;
20 signal sig_weights        : std_logic_vector(31 downto 0);
21 signal sig_input          : std_logic_vector(31 downto 0);
22 signal sig_mult_start    : std_logic;
23 signal sig_mult_stop     : std_logic;
24 signal sig_prodotto       : std_logic_vector(63 downto 0);
25 signal sig_in_reg         : std_logic_vector(63 downto 0);
26 signal sig_out_reg        : std_logic_vector(63 downto 0) := (others=>'0');
27 signal sig_adder1         : std_logic_vector(63 downto 0);
28 signal sig_adder2         : std_logic_vector(63 downto 0);
29 signal sig_sel_bias       : std_logic := '0';
30 signal sig_load_r         : std_logic;
31 signal sig_reset          : std_logic;
32
33 signal temp_bias          : std_logic_vector(63 downto 0);
34
35 begin
36
37 controllo : entity work.uc_neurone_output port map (
38     clock      => clock,
39     reset      => reset,
40     start      => start,
41     mult_stop  => sig_mult_stop,
42     counter    => sig_counter,
43     mult_start => sig_mult_start,
44     cont       => sig_cont,
45     read       => sig_read,
46     load_reg   => sig_load_r,
47     sel_bias   => sig_sel_bias,
48     rst_all    => sig_reset,

```



```

107     output  => sig_adder1
108 );
109
110 mux_bias : entity work.mux_2_1
111 generic map (
112     bits => 64
113 )
114 port map (
115     input0  => sig_out_reg,
116     input1  => temp_bias,
117     s       => sig_sel_bias,
118     output   => sig_adder2
119 );
120
121 addizionatore : entity work.adder_sub_64 port map (
122     X      => sig_adder1,
123     Y      => sig_adder2,
124     cin    => '0',
125     Z      => sig_in_reg,
126     cout   => open
127 );
128
129 registroo : entity work.registro
130 generic map (
131     bits => 64
132 ) port map (
133     clock   => clock,
134     reset   => reset or sig_reset,
135     enable  => sig_load_r,
136     in_p    => sig_in_reg,
137     out_p   => sig_out_reg
138 );
139
140 funz_attivazione : entity work.ReLU
141 generic map (
142     bits => 64
143 )
144 port map (
145     out_neuron  => sig_out_reg,
146     rectified   => value
147 );
148
149 end structural;

```

Output layer

L'output layer è composto da 7 neuroni di output che vengono tutti inizializzati tramite un for-generate. Questo è stato possibile grazie ai tipi e alle costanti definite nel package. Infatti qui assegnamo i pesi ed i bias ad ogni neurone. Di seguito il codice.

```

1 entity output_layer is port (
2     clock   : in std_logic;
3     reset   : in std_logic;
4     inputs  : in hidden_output;
5     start   : in std_logic_vector(9 downto 0);

```

```

6      output  : out uscite;
7      end_op  : out std_logic_vector(6 downto 0)
8 );
9 end output_layer;
10
11 architecture structural of output_layer is
12
13 component output_neuron is
14 generic (
15   weights : rom_type_output;
16   bias     : std_logic_vector(6 downto 0)
17 );
18
19 port (
20   clock    : in std_logic;
21   reset    : in std_logic;
22   start    : in std_logic_vector(9 downto 0);
23   inputs   : in hidden_output;
24   end_op   : out std_logic;
25   value    : out std_logic_vector(63 downto 0)
26 );
27
28 begin
29
30 neuroni0to6: for i in 0 to 6 generate
31   neurone : output_neuron
32   generic map (
33     weights => array_output(i),
34     bias     => value_output_bias(i)
35   )
36   port map (
37     clock    => clock,
38     reset    => reset,
39     start    => start,
40     inputs   => inputs,
41     end_op   => end_op(i),
42     value    => output(i)
43   );
44
45 end generate neuroni0to6;
46
47 end structural;

```

Rete Neurale

La rete neurale è una semplice composizione dei due layer, come è stato visto nello schematico riportato nella sezione precedente. In particolare abbiamo aggiunto un po' di logica per la presentazione dei risultati. Infatti i risultati vengono mostrati solo quando la rete ha terminato l'elaborazione, cioè quando tutti gli end_op dell'output layer sono alti. Inoltre vediamo che l'output layer si resetta quando viene dato lo start alla rete. In questo modo continuiamo a presentare in uscita il risultato corrispondente all'ultimo input, questo finché non viene dato il segnale di start alla rete, che resetta l'output layer e quindi i neuroni al suo

interno. Questo era il motivo per cui l'automa dell'unità di controllo del neurone dell'output layer rimaneva bloccato nello stato fine, per poi tornare in idle solo quando avveniva un reset. Come già accennato potrebbe essere possibile aggiungere il blocco che restituisce in uscita solamente il massimo tra i valori in uscita dai neuroni dell'output layer. La struttura di questo blocco è stata discussa nella sezione precedente. Di seguito il codice della rete neurale.

```

1 entity rete_neurale is port (
2     clock      : in std_logic;
3     reset      : in std_logic;
4     start      : in std_logic;
5     input0     : in std_logic_vector(8 downto 0);
6     input1     : in std_logic_vector(8 downto 0);
7     input2     : in std_logic_vector(8 downto 0);
8     input3     : in std_logic_vector(8 downto 0);
9     input4     : in std_logic_vector(8 downto 0);
10    input5     : in std_logic_vector(8 downto 0);
11    input6     : in std_logic_vector(8 downto 0);
12    input7     : in std_logic_vector(8 downto 0);
13    input8     : in std_logic_vector(8 downto 0);
14
15    output     : out uscite
16 );
17 end rete_neurale;
18
19 architecture structural of rete_neurale is
20 signal conn_data          : hidden_output;
21 signal conn_end_op_h      : std_logic_vector(9 downto 0);
22 signal conn_end_op_o      : std_logic_vector(6 downto 0);
23 signal sig_output         : uscite;
24 begin
25
26 layer_nascosto : entity work.hidden_layer port map (
27     clock      => clock,
28     reset      => reset,
29     start      => start,
30     input0     => input0,
31     input1     => input1,
32     input2     => input2,
33     input3     => input3,
34     input4     => input4,
35     input5     => input5,
36     input6     => input6,
37     input7     => input7,
38     input8     => input8,
39     output     => conn_data,
40     end_op     => conn_end_op_h
41 );
42
43 layer_output : entity work.output_layer port map (
44     clock      => clock,
45     reset      => reset or start,
46     inputs     => conn_data,
47     start      => conn_end_op_h,
48     output     => sig_output,

```

```

49     end_op  => conn_end_op_o
50 );
51
52 output <=  sig_output when conn_end_op_o = "1111111" else
53     (others => x"0000000000000000");
54
55 end structural;

```

12.1.3 Simulazione

Per la simulazione abbiamo testato prima di tutto i neuroni singolarmente, per poi testare la rete neurale nel suo complesso

Hidden Neuron

Per testare il neurone dell'hidden layer abbiamo usato sia pesi che bias reali (cioè usati anche nella rete effettiva), riportati usando segnali durante il mapping delle porte. In particolare abbiamo usato pesi e bias del neurone 1 dell'hidden layer. Gli input inseriti sono molto semplici (tutti 1), utilizzati solamente per testare che la rete effettui correttamente i calcoli. Di seguito il codice della simulazione.

```

1 input0 <= "000000001";
2 input1 <= "000000001";
3 input2 <= "000000001";
4 input3 <= "000000001";
5 input4 <= "000000001";
6 input5 <= "000000001";
7 input6 <= "000000001";
8 input7 <= "000000001";
9 input8 <= "000000001";
10
11 start <= '1';
12 wait for clk_period*3;
13 start <='0';

```

Dato che il bias del neurone 1 è 2 ed i pesi sono i seguenti:

$$61, 54, -46, -50, -29, 17. - 33, -17, 49$$

Implementando l'operazione tipica del neurone presentata ad inizio della sezione il risultato dovrebbe essere 8 che in esadecimale è sempre 8. Dalla simulazione in figura 132 vediamo che effettivamente il risultato quando end_op viene alzato è 8. Inoltre è da notare come i risultato intermedi del neurone non siano mai negativi, questo perchè c'è la ReLU in uscita dal neurone, che fa in modo che i risultati negativi diventino 0 (per questo ci sono molti

risultati intermedi uguali a zero). Notiamo come per presentare il risultato in uscita la rete impieghi circa $3\mu s$ con un clock con periodo $20ns$.

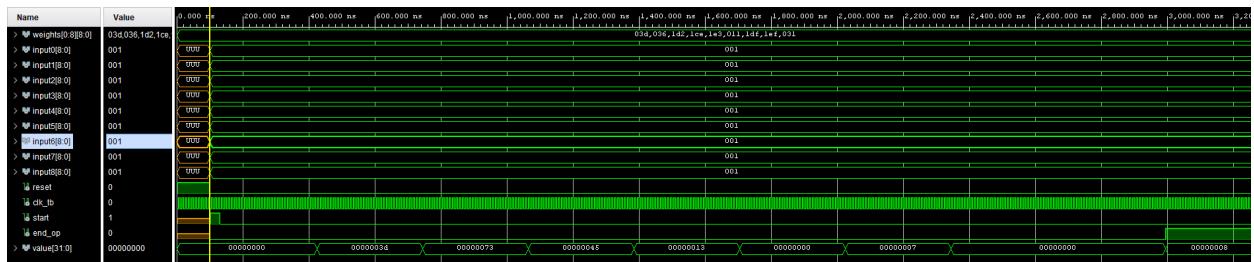


Figura 132: Simulazione dell'hidden neuron

Output Neuron

Per testare il neurone dell'output layer abbiamo usato sia pesi che bias reali (cioè quelli usati anche nella effettiva), riportati usando segnali durante il mapping delle porte. In particolare abbiamo usato pesi e bias del neurone 0 dell'output layer. Gli input inseriti sono molto semplici (tutti 1), utilizzati solamente per testare che la rete effettui correttamente i calcoli. Il segnale start è un vettore perchè un neurone dell'output layer si attiva quando tutti quelli dell'hidden layer hanno terminato. Gli input sono stati definiti direttamente come valore di default del segnale, ma come riportato sopra sono tutti uguali ad 1. Di seguito il codice della simulazione.

```

1 start <= (others => '1');
2 wait for clk_period*3;
3 start <= (others => '0');
```

Dato che il bias del neurone 0 è 12 ed i pesi sono i seguenti:

$$0, 0, -92336, 0, -257680, 208260.0, 0, 4 - 97752, 468988$$

Implementando l'operazione tipica del neurone presentata ad inizio della sezione il risultato dovrebbe essere 229.492 che in esadecimale è 38.074. Dalla simulazione in figura 133 vediamo che effettivamente il risultato quando end_op viene alzato è 8. Notiamo che qui, addirittura fino alla fine del calcolo i risultati del neurone sono sempre negativi, quindi in output è presente 0 a causa della ReLU. Solamente alla fine viene sommato un grande numero positivo, quindi il risultato diventa positivo e ritroviamo effettivamente il risultato è corretto. Notiamo come per presentare il risultato in uscita la rete impieghi circa $10\mu s$ con un clock con periodo

20ns, il neurone impiega più tempo perchè i valori con cui operare (pesi e valori dei neuroni) sono rappresentati su più bit.

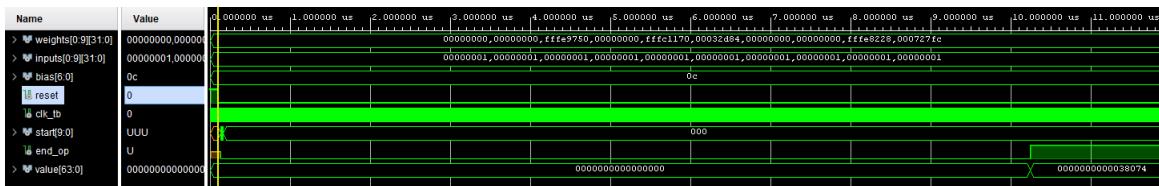


Figura 133: Simulazione dell'output neuron

Rete neurale

Per testare la rete completa abbiamo usato l'intero progetto, quindi abbiamo usato la rete neurale come una black box, senza dover impostare pesi e bias dall'esterno. Anche in questo caso gli input sono molto semplici (tutti 1), però non è stato calcolato il risultato finale in quanto il calcolo del risultato completo dell'intera rete è molto tedioso. Avendo però controllato il corretto funzionamento dei singoli neuroni è certo che il risultato della rete sia quello corretto, cioè quello che ci si aspetterebbe dati gli ingressi.

Per testare che i risultati della rete vengano presentati solo alla fine del calcolo e che gli stessi vengano resettati appena vi è un nuovo segnale di start abbiamo posto in ingresso alla rete due inputs molto semplici. Di seguito il codice della simulazione.

```

1  input0  <= "0000000001";
2  input1  <= "0000000001";
3  input2  <= "0000000001";
4  input3  <= "0000000001";
5  input4  <= "0000000001";
6  input5  <= "0000000001";
7  input6  <= "0000000001";
8  input7  <= "0000000001";
9  input8  <= "0000000001";

10
11 start <= '1';
12 wait for clk_period*3;
13 start <='0';

14
15 wait for 20000 ns;

16
17 input0  <= "000000010";
18 input1  <= "000000010";
19 input2  <= "000000010";
20 input3  <= "000000010";
21 input4  <= "000000010";
22 input5  <= "000000010";
23 input6  <= "000000010";
24 input7  <= "000000010";

```

```

25 input8 <= "000000010";
26
27 start <= '1';
28 wait for clk_period*3;
29 start <='0';

```

Dalla simulazione in figura 134 vediamo che l'uscita rimane nulla fino al termine dell'inferenza della rete, solo a quel punto abbiamo in uscita i risultati della rete, cioè i valori di tutti i neuroni dell'output layer. Notiamo inoltre che la rete presenta il risultato dell'inferenza dopo circa $13\mu s$ che è la somma del tempo necessario all'elaborazioni dei neuroni di hidden e output layer, quindi effettivamente i neuroni lavorano in parallelo.

Come già detto si potrebbe inserire un componente che metta in uscita solo il più alto di questo valori, in quanto sarebbe l'effettiva risposta della rete. Per favorire l'analisi dei risultati abbiamo deciso di non inserire un blocco di questo tipo sull'uscita della rete neurale. Notiamo inoltre come con il nuovo segnale di start i valori in output si azzerino di nuovo e assumano un nuovo valore solo al termine dell'inferenza della rete, rispettando quindi i nostri vincoli.

Notiamo inoltre come il valore più alto sia spesso relativo al primo output (quindi la prima classe), si vede ciò anche in questi due esempi, questo accade perchè il dataset utilizzato è molto sbilanciato, infatti circa l'80% del dataset appartiene alla prima classe, quindi la rete tende in generale a fornire come output più alto il primo. Questo a meno che la rete non riconosca pattern specifici che ha appreso durante la fase di addestramento e che sono collegati ad altre classi risponderà sempre che ci si trova nel primo stato. Purtroppo non siamo riusciti a trovare una pubblicazione del dataset, quindi non abbiamo potuto testare dei valori di input che avessero senso fisico per controllare se il risultato della rete coincidesse con quello corretto riportato nel dataset stesso.

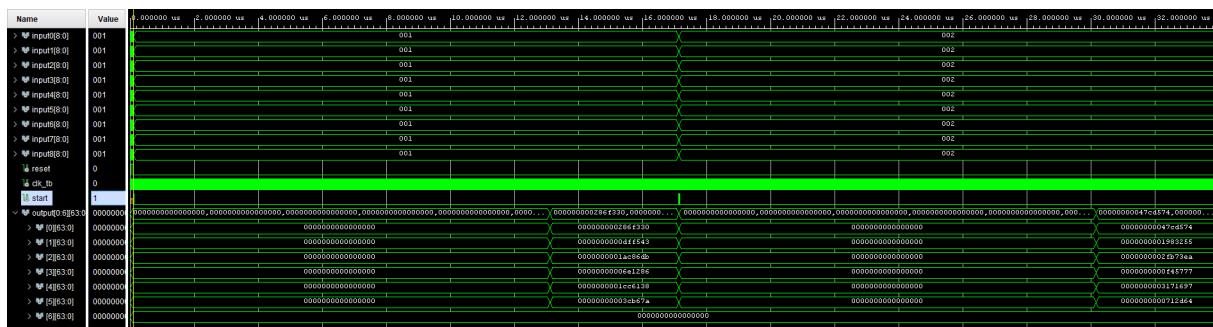


Figura 134: Simulazione della Rete Neurale

Riferimenti bibliografici

- [1] Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. Pearson, Boston, MA, 6 edition, 2012.