Indice

1 Esercizio 1 - Multiplexer

1.1 Parte 1

L'esercizio 1.1 richiede la rappresentazione di un multiplexer 16:1 tramite la composizione di multiplexer 4:1, quindi definiamo il module del componente mux_4_1 definito in modo Dataflow, e poi per le proprietà della modularità definiamo il module mux_16_1 come composizione dei precedenti, tramite il costrutto *for..generate*.
Il multiplexer 4:1 ha come ingressi quattro bit, identificati col vettore a(0 to 3), e $\lceil \log_2(n) \rceil$ segnali di abilitazione, dove n è il numero di segnali di ingresso. In questo caso, quindi, due segnali di abilitazione ed uno di uscita.

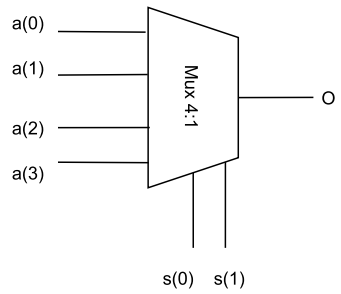

Figure 1: Mux 4:1

Il codice in VHDL per descrivere il comportamento di questo componente è il seguente:

```
architecture DataFlow of mux_4_1 is

begin
    o <= a(0) when s = "00" else
         a(1) when s = "01" else
         a(2) when s = "10" else
         a(3) when s = "11" else
         '-';

end DataFlow;
```

Figure 2: Mux 4:1 Dataflow

Composto da un costrutto *when...else* che suddivide i diversi casi e gestisce anche tutti i casi non definiti con l'ultima clausola else senza alcuna condizione.
Definito l'elemento base del progetto, si passa a comporre il multiplexer 16:1 tramite un approccio strutturale, nel quale generiamo cinque mux_4_1: i primi quattro avranno gli ingressi interfacciati con l'esterno e, tramite segnali interni, le loro uscite sono collegate come ingressi dell'ultimo multiplexer che costituisce l'uscita del sistema. La macchina completa presenta quindi 16 segnali di ingresso, 4 segnali di selezione ed un unico segnale di uscita:

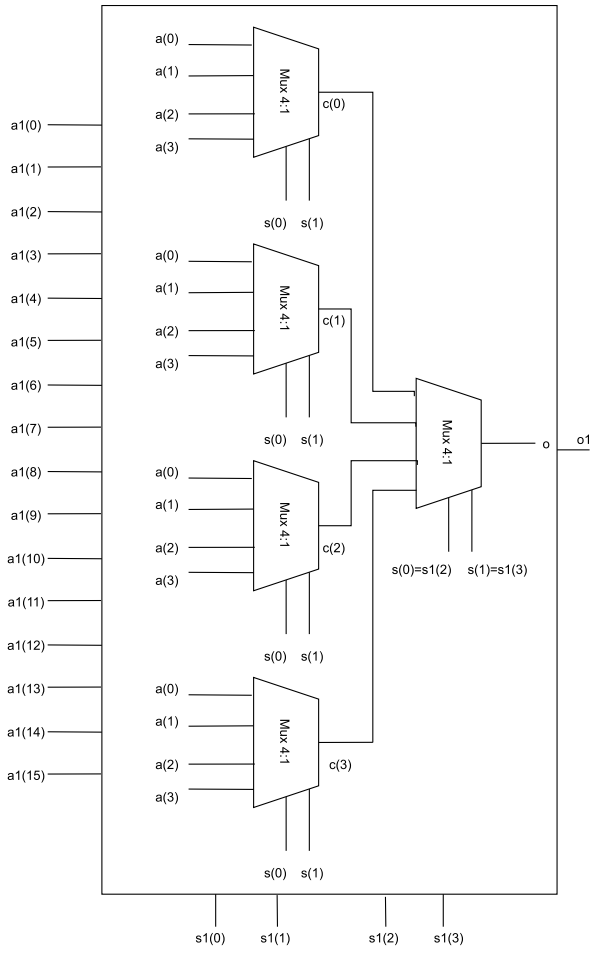

Figure 3: Mux 16:1

Lo schema sopra rappresentato è tradotto in linguaggio VHDL dal seguente codice, dove l'entità mux_16_1 è rappresentata con approccio strutturale e, tramite i segnali interni c(0 to 3), colleghiamo le uscite parziali dei primi quattro mux_4_1, identificati con la label mux_4_1_in, con i quattro ingressi dell'ultimo mux_4_1, identificato con la label mux_4_1_fin.

```
entity mux_16_1 is
    Port ( al : in STD_LOGIC_VECTOR (0 to 15);
           sl : in STD_LOGIC_VECTOR (0 to 3);
           ol : out STD_LOGIC);
end mux_16_1;

architecture Structural of mux_16_1 is
    COMPONENT mux_4_1 PORT (a : in STD_LOGIC_VECTOR (0 to 3); s : in STD_LOGIC_VECTOR (0 to 1); o : out STD_LOGIC);
    END COMPONENT;
    FOR ALL: mux_4_1 USE ENTITY WORK.mux_4_1 (Behavioral);
    signal c : STD_LOGIC_VECTOR (0 to 3);
begin
    mux_4_1_in : FOR i in 0 to 3 GENERATE m: mux_4_1
        port map (
        a(0 to 3) => al(i*4 to i*4+3),
        s(0 to 1) => sl(0 to 1),
        o => c(i)
        );
    end GENERATE;

    mux_4_1_fin : mux_4_1
        port map(
        a(0 to 3) => c(0 to 3),
        s(0 to 1) => sl(2 to 3),
        o => ol
    );
end Structural;
```

Figure 4: Mux 16:1 Dataflow

Progettato il mux_16_1, è possibile testarlo attraverso un testbench. La prima cosa che bisogna specificare è che il corpo dell'entity è vuoto, questo perché non si tratta di oggetto che realizziamo, ma serve solo per effettuare la simulazione e verificare se il sistema realizzato funziona correttamente. Il testbench effettivamente non ha né segnali d'ingresso né d'uscita, ma sfrutta per i test i segnali interni definiti nel codice. Per testare il mux_16_1 definito precedentemente, abbiamo istanziato una uut (Unit Under Test) in cui colleghiamo le varie porte ai segnali (input, selection, output).

```
SIGNAL input : STD_LOGIC_VECTOR (0 TO 15);
SIGNAL selection : STD_LOGIC_VECTOR (0 TO 3);
SIGNAL output : STD_LOGIC;

begin
mux_16_1 : mux_16 PORT MAP (a1(0 to 15)=>input(0 to 15), s1(0to 3)=>selection(0 to 3), o1=>output);

mux_16_2 : input <= "0000000000000000",
"0000000000000001" AFTER 100 NS,
"0000000000000011" AFTER 200 NS,
"1100000000000000" AFTER 300 NS;

mux_16_3 : selection <= "0000",
"1111" AFTER 100 NS,
"0011" AFTER 200 NS,
"1100" AFTER 300 NS;
end structural;
```

Figure 5: Mux 16:1 Testbench

Dopo aver effettuato queste assegnazioni, compreso di costrutto after per permettere l'evoluzione del sistema durante il tempo, si passa alla schermata di simulazione nella quale si può analizzare e studiare l'evoluzione nel tempo di ogni segnale presente nel codice, compresi eventuali segnali intermedi.

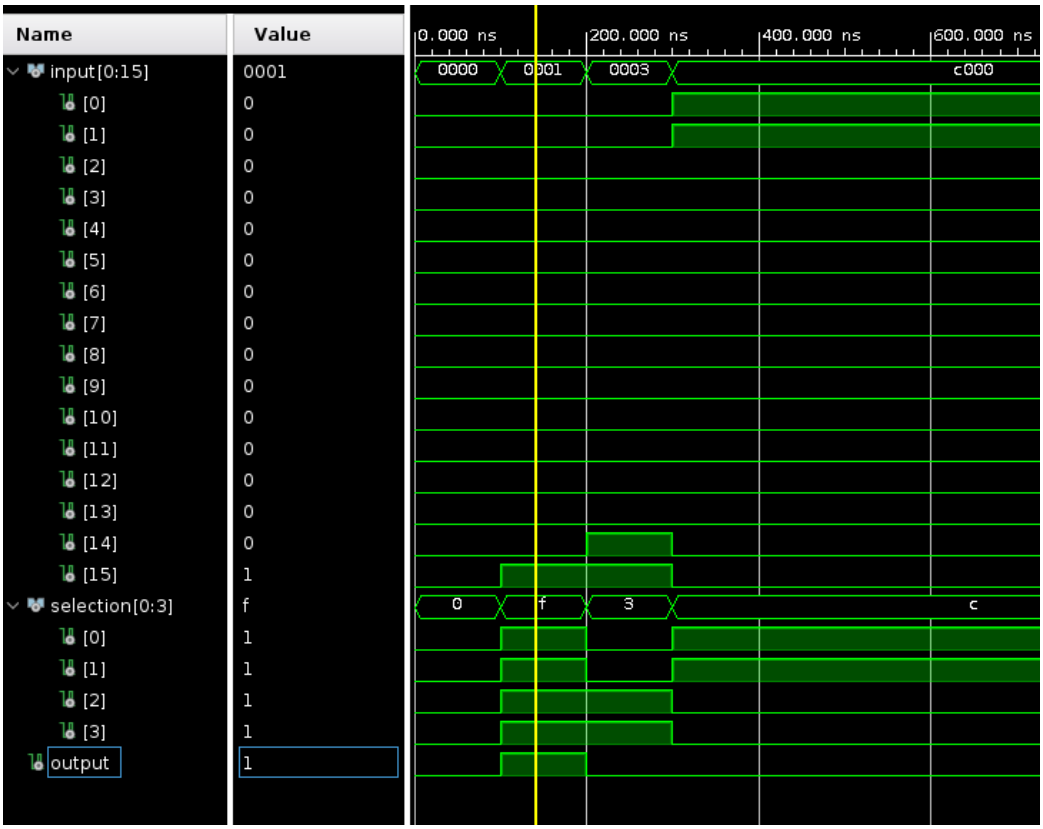

Figure 6: Mux 16:1 Simulazione

Possiamo analizzare l'evoluzione del programma: dopo circa 150ns, l'ingresso è posto a "0000000000000001" e contemporaneamente la selezione è posta a "1111", ottenendo come uscita del sistema "1". Questo è effettivamente il comportamento atteso.

1.2 Parte 2

L'esercizio 1.2 è in parte riconducibile all'esercizio precedente, in quanto la rappresentazione di una rete 16:4 può essere scomposta da una sottorete 16:1 connessa ad un demux 1:4. Basta quindi aggiungere un demultiplexer 1:4 alla rete precedente. Il demux 1:4 è realizzato con approccio Dataflow.

```
architecture DataFlow of demux_4_1 is
begin
    output <= input&"000" when enable="00" else
              '0'&input&"00" when enable="01" else
              "00"&input&'0' when enable="10" else
              "000"&input when enable="11" else
              "----";

end DataFlow;
```

Figure 7: Demux 1:4 Dataflow

Il Demux 1:4 presenta come uscita un segnale di 4 bit, di cui 3 pari a zero ed uno pari al valore in ingresso, la cui posizione è determinata a seconda dei segnali di abilitazione.

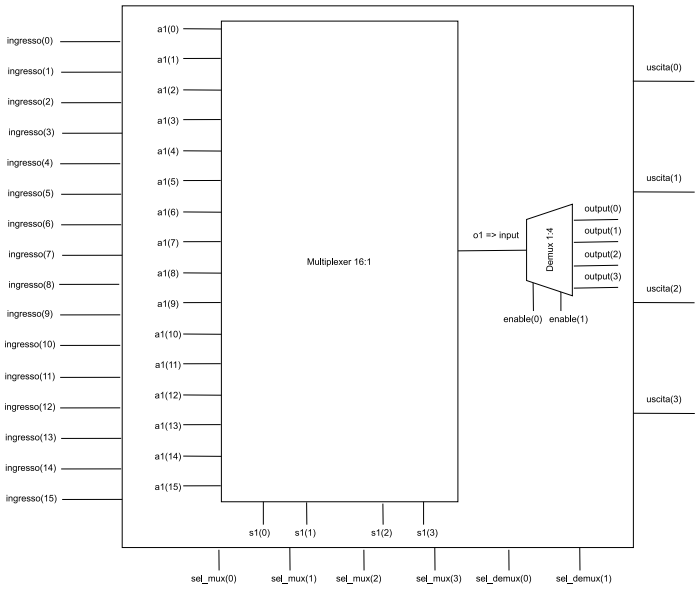

Figure 8: Rete 16:1

Una volta eseguite le interconnessioni tramite un unico segnale interno, utilizzato per collegare l'uscita del mux_16_1 con l'ingresso del demux_1_4, si ottiene la rete 16:4: tale rete presenta 16 segnali di ingressi totali, 6 di selezione (di cui 2 utilizzati per il demux) e 4 segnali di uscita.

```
signal interco : STD_LOGIC;
begin
    mux : mux_16_1
    PORT MAP (
    a1(0 to 15) => ingresso(0 to 15),
    s1( 0 to 3) => sel_mux(0 to 3),
    o1 => interco
    );
    demux : demux_4_1
    PORT MAP (
    input => interco,
    enable( 0 to 1) => sel_demux(0 to 1),
    output(0 to 3) => uscita ( 0 to 3)
    );


end Structural;
```

Figure 9: Rete 16:4 Structural

Anche dopo aver testato il singolo componente, è comunque necessario ripetere il test per la macchina completa, sia perché potrebbero essere presenti errori e problemi derivanti da implementazione di nuove funzioni, sia perché anche nella composizione di una macchina più complessa sono presenti intrinsecamente problemi legati alla coesione dei vari moduli.

```
begin
rete_16_4_1: rete_16_4 port map(
    ingresso (0 to 15) => i(0 to 15),
    uscita (0 to 3) => u (0 to 3),
    sel_mux(0 to 3) => s_mux(0 to 3),
    sel_demux(0 to 1) => s_demux(0 to 1));

rete_16_4_2: i <= "0000001000000000",
"0000000000000000" after 100ns,
"1000000000000000" after 200ns,
"1110111111111111" after 300ns;

rete_16_4_3: s_mux <= "1001",
"1111" after 100ns,
"0000" after 200ns,
"1100" after 300ns;

rete_16_4_4: s_demux <= "00",
"10" after 100ns;
```

Figure 10: Rete 16:4 Testbench

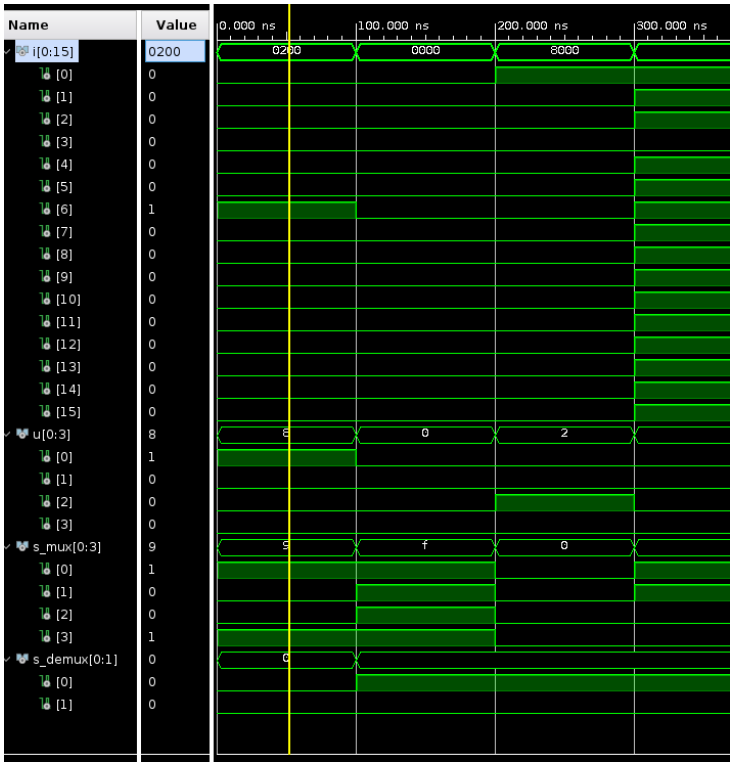

Figure 11: Rete 16:4 Simulazione

1.3 Parte 3

Come esercizio finale, è stato necessario adattare la rete per la sintesi sulla FPGA. Attraverso l'utilizzo di un file di constraint ideato per la board Nexys A7-50t, possiamo definire i collegamenti da effettuare sulla scheda tra le diverse periferiche disponibili e le componenti presenti all'interno della rete.
In questo caso sono necessari 6 switch per le linee di abilitazione dei multiplexer e demultiplexer. Poichè gli switch sono in totale 16, è stato necessario dare un input predefinito alla rete ed utilizzare gli switch unicamente per la selezione. Inoltre, sempre dal file di costraint, sono stati abilitati anche 4 led e connessi ai quattro bit di uscita, come indicato nel seguente file.

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

### LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { uscita[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { uscita[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { uscita[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { uscita[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { output[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { output[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
```

Figure 12: Costraints

Quindi, gli switch ***J15-L16-M13-R15*** sono stati mappati ai quattro bit di selezione dei multiplexer, mentre gli switch ***R17-T18*** ai due bit di selezione dei multiplexer, ed, infine, i led H17-K15-J13-N14 come rappresentazione visiva dei quattro bit di uscita al sistema complessivo.
Il sistema finale è quello raffigurato nella seguente figura

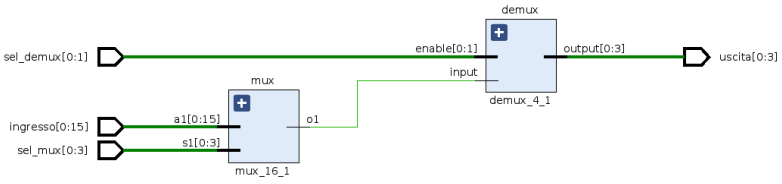

Figure 13: Rete 16:4 Analysis

2 Esercizio 2 - Encoder BCD

2.1 Traccia

Progettare, implementare in VHDL e testare mediante simulazione una rete che, data in ingresso una stringa binaria X di 10 bit X9 X8 X7 X6 X5 X4 X3 X2 X1 X0 che corrisponde alla rappresentazione decodificata di una cifra decimale (cioè, una rappresentazione in cui ogni stringa contiene un solo bit alto), fornisce in uscita la rappresentazione Y della cifra mediante codifica Binary-Coded Decimali (BCD).
Input: 0000000001 ⇒ Output: 0000 (cifra 0)
Input: 0000000010 ⇒ Output: 0001 (cifra 1)
Input: 0000000100 ⇒ Output: 0010 (cifra 2)
....
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).
Utilizzare un display a 7 segmenti per visualizzare la cifra decimale codificata da Y (pilotare opportunamente i catodi del display per visualizzare la cifra).

2.2 Soluzione

La rete è stata realizzata con vari componenti secondo un approccio strutturale. Essa ha un ingresso, cioè il valore della stringa X da 10 bit, e 2 uscite, utili per visualizzare la cifra codificata su un display a 7 segmenti. La rete utilizza i seguenti componenti:

- Un encoder: a sua volta composto da un arbitro a priorità e da un encoder 10:4;
- Un display manager per la visualizzazione dell'output;

Si è partiti, dunque, da una descrizione dataflow dei componenti base, per poi procedere con una descrizione strutturale dell'encoder ed una descrizione comportamentale del display manager. Infine, mettendo insieme questi ultimi 2 componenti, si è descritto l'intero sistema, chiaramente a livello strutturale.

Arbitro a priorità:

Il componente relativo all'arbitro di priorità dispone di un vettore di ingresso di 10 bit e di un vettore di uscita di altrettanti bit, in cui l'uscita avrà tutti 0 e un solo bit alto nella prima posizione in cui è stato trovato un 1 (a partire dalla posizione più significativa).

Encoder 10:4:

Il componente relativo all'encoder 10:4 presenta in ingresso un vettore di 10 bit ed in uscita un vettore di 4 bit, che rappresenta il numero in binario della prima posizione con bit alto in ingresso (valore compreso nel range [0,9]).

Facendo uso dei due componenti appena descritti, si è realizzato un encoder, il quale facendo uso di un segnale interno che fa da interconnessione tra l'uscita dell'arbitro e l'ingresso dell'encoder 10:4, prende in ingresso un vettore di 10 bit e restituisce in uscita un vettore di 4 bit effettuando la codifica Binary-Coded Decimal (BCD).

Display Manager:

Il componente display manager prende in ingresso un vettore di 4 bit e lo rappresenta sul display a 7 segmenti con 2 uscite. Al suo interno vengono definite delle costanti su 7 bit relative ai segmenti di una cifra del display da illuminare, in questo modo si visualizza un determinato valore in esadecimale. La prima uscita è fissa, al fine di illuminare costantemente solo la prima cifra del display, poiché il valore da mostrare è rappresentabile con una sola cifra. La seconda uscita determina i segmenti della cifra da illuminare, pertanto dipende dall'ingresso.
La rete complessiva dispone dell'encoder e del display manager. Il vettore di ingresso di 10 bit andrà nell'encoder e la sua uscita su 4 bit, mediante un segnale di interconnessione, andrà in ingresso al display manager che mostra il valore di uscita sul display.

2.3 Codice

Arbitro

L'architettura è stata descritta a livello dataflow e, scorrendo un vettore a partire dalla posizione 9 fino a 0, se tra i 10 bit uno solo è alto, l'uscità sarà una stringa con tutti 0 e solo un bit alto nella posizione in cui lo era nel vettore di ingresso. Nel caso in cui il vettore di ingresso presenta più bit uguali a 1, la stringa in uscita avrà il solo bit alto nella prima posizione, partendo dalla 9, in cui trova un 1.

```
entity arbitro is
    Port (
        x: in std_logic_vector(9 downto 0);
        y: out std_logic_vector(9 downto 0)
    );
end arbitro;

architecture dataflow of arbitro is

begin
        y <= "1000000000" when x(9) = '1' else
             "0100000000" when x(8) = '1' else
             "0010000000" when x(7) = '1' else
             "0001000000" when x(6) = '1' else
             "0000100000" when x(5) = '1' else
             "0000010000" when x(4) = '1' else
             "0000001000" when x(3) = '1' else
             "0000000100" when x(2) = '1' else
             "0000000010" when x(1) = '1' else
             "0000000001" when x(0) = '1' else
             "----------";

end dataflow;
```

Figure 14: Codice Arbitro

Encoder 10:4

L'encoder 10:4 è stato descritto a livello dataflow. L'architettura, partendo da un vettore con solo un bit alto, restituisce in uscita il valore su 4 bit della posizione in cui il bit è alto.

```
entity encoder10_4 is
    Port (
        x: in std_logic_vector(9 downto 0);
        y: out std_logic_vector(3 downto 0)
    );
end encoder10_4;

architecture dataflow of encoder10_4 is

begin
       with x select
            y <= "0000" when "0000000001",
                 "0001" when "0000000010",
                 "0010" when "0000000100",
                 "0011" when "0000001000",
                 "0100" when "0000010000",
                 "0101" when "0000100000",
                 "0110" when "0001000000",
                 "0111" when "0010000000",
                 "1000" when "0100000000",
                 "1001" when "1000000000",
                 "----" when others;

end dataflow;
```

Figure 15: Codice Encoder 10:4

Encoder complessivo

L'encoder complessivo è descritto a livello strutturale utilizzando i componenti "arbitro" e "encoder10_4". All'interno del sistema è definito un segnale t di tipo std_logic_vector(9 downto 0), che fa da interconnessione e viene utilizzato come uscita dell'arbitro e come ingresso dell'encoder 10:4.

```
entity encoder is
    Port (
        x: in std_logic_vector(9 downto 0);
        y: out std_logic_vector(3 downto 0)
    );
end encoder;

architecture structural of encoder is

    component arbitro port(
        x: in std_logic_vector(9 downto 0);
        y: out std_logic_vector(9 downto 0)
    );
    end component;

    component encoder10_4 port (
         x: in std_logic_vector(9 downto 0);
         y: out std_logic_vector(3 downto 0)
    );
    end component;

    signal t: std_logic_vector(9 downto 0);



    begin
       arbitro_1: arbitro port map(
            x => x,
            y => t
       );
       encoder10_4_1:  encoder10_4 port map(
            x => t,
            y => y
       );

    end structural;
```

Figure 16: Codice Encoder Complessivo

Display Manager

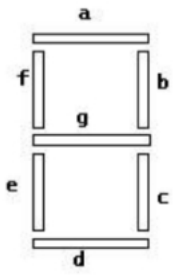

Figure 17: Display Segment

Questo componente viene descritto a livello comportamentale. Viene definita l'entity con una porta in ingresso, value, e due porte in uscita, anode e cathode, tutte di tipo std_logic_vector. Nel costrutto architecture sono definite delle costanti di 7 bit, dove la posizione rappresenta un segmento di una cifra del display: quando il bit è 0 il segmento è acceso, altrimenti è spento. Poiché il valore è in notazione esadecimale, sono state definite le costanti che rappresentano i valori da 0 a f. Per tenere accesa solo la prima cifra del display, l'uscita anode viene settata con tutti i bit alti, ad eccezione di quello meno significativo; in questo modo le cifre successive alla prima sono spente. Si usa l'altra uscita, cathode, per rappresentare il valore tramite una AND tra '1' e la costante che rappresenta il valore da mostrare a video.

```
entity display_manager is
        Port (
              --clk: in std_logic;
              --rst: in std_logic;
              value: in std_logic_vector(3 downto 0);
              anode : out std_logic_vector(7 downto 0);
              cathode : out std_logic_vector(7 downto 0)
        );
end display_manager;
```

```
architecture behavioral of display_manager is

constant zero   : std_logic_vector(6 downto 0) := "1000000";
constant one    : std_logic_vector(6 downto 0) := "1111001";
constant two    : std_logic_vector(6 downto 0) := "0100100";
constant three  : std_logic_vector(6 downto 0) := "0110000";
constant four   : std_logic_vector(6 downto 0) := "0011001";
constant five   : std_logic_vector(6 downto 0) := "0010010";
constant six    : std_logic_vector(6 downto 0) := "0000010";
constant seven  : std_logic_vector(6 downto 0) := "1111000";
constant eight  : std_logic_vector(6 downto 0) := "0000000";
constant nine   : std_logic_vector(6 downto 0) := "0010000";
constant a      : std_logic_vector(6 downto 0) := "0001000";
constant b      : std_logic_vector(6 downto 0) := "0000011";
constant c      : std_logic_vector(6 downto 0) := "1000110";
constant d      : std_logic_vector(6 downto 0) := "0100001";
constant e      : std_logic_vector(6 downto 0) := "0000110";
constant f      : std_logic_vector(6 downto 0) := "0001110";

begin
    anode <= not("00000001");
    with value select
            cathode <= '1' & zero  when "0000",
                       '1' & one   when "0001",
                       '1' & two   when "0010",
                       '1' & three when "0011",
                       '1' & four  when "0100",
                       '1' & five  when "0101",
                       '1' & six   when "0110",
                       '1' & seven when "0111",
                       '1' & eight when "1000",
                       '1' & nine  when "1001",
                       "--------" when others;

end behavioral;
```

Figure 18: Codice Display Manager

Sistema Completo

Nella descrizione strutturale dell'architettura si definiscono i componenti encoder, quello complessivo, e display manager e si definisce un segnale interno temp, di tipo std_logic_vector, che fa da interconnessione tra l'uscita dell'encoder e l'ingresso del display manager, al fine di rappresentare a video l'output dell'encoder.

```
entity sistema_completo is
    Port (
        x: in std_logic_vector(9 downto 0);
        anode : out  std_logic_vector(7 downto 0);
        cathode : out std_logic_vector(7 downto 0)
    );
end sistema_completo;
```

```
architecture structural of sistema_completo is

    component encoder is
    Port (
        x: in std_logic_vector(9 downto 0);
        y: out std_logic_vector(3 downto 0)
    );
    end component;

    component display_manager is Port (
            --clk: in std_logic;
            --rst: in std_logic;
            value: in std_logic_vector(3 downto 0);
            anode : out std_logic_vector(7 downto 0);
            cathode : out std_logic_vector(7 downto 0)
       );
    end component;

    signal tmp : std_logic_vector(3 downto 0) := "----";
```

```
    begin
        enc:  encoder Port map (
            x => x,
            y => tmp
        );
        disp:  display_manager Port map (
            value => tmp,
            anode => anode,
            cathode => cathode
        );

    end structural;
```

Figure 19: Codice Sistema Completo

2.4 Simulazione

Per effettuare la simulazione è stato utilizzato il seguente testbench per l'encoder:

```
entity sim_encoderfinale is
--  Port ( );
end sim_encoderfinale;

architecture Behavioral of sim_encoderfinale is

    component encoder is
    Port (
        x: in std_logic_vector(9 downto 0);
        y: out std_logic_vector(3 downto 0)
    );
    end component;

    signal x:  std_logic_vector(9 downto 0);
    signal y:  std_logic_vector(3 downto 0);



begin
    utt: entity work.encoder port map(
        x => x,
        y => y
    );
    sim_proc: process
    begin
        wait for 20ns;
        x <= (0 => '1', others => '0');
        wait for 40ns;
        x <= (0 => '1', 1 => '1', others => '0');
        wait for 40ns;
        x <= (7 => '1', others => '0');
        wait for 20ns;
        x <= (9 => '1', others => '0');
        wait for 20ns;
        x <= (others => '0');
        wait;
    end process;

end Behavioral;
```

Figure 20: Codice Testbench

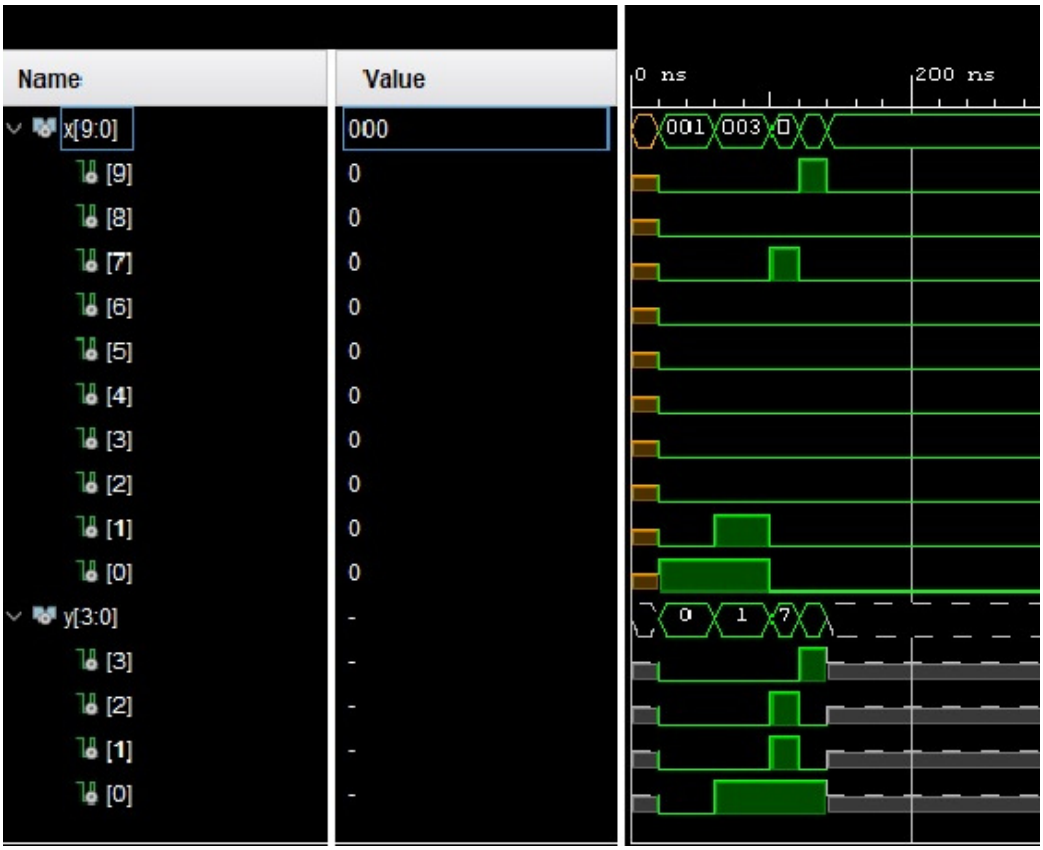

Figure 21: Simulazione Encoder

3 Esercizio 3 - Riconoscitore di Sequenze

3.1 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 tempificazione e un segnale M di modo, che ne disciplina il funzionamento, e fornisce un'uscita Y alta quando la sequenza viene riconosciuta. In particolare,

- se M=0, la macchina valuta i bit seriali in ingresso a gruppi di 4,
- se M=1, la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta.

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 Soluzione

La macchina realizzata è un riconoscitore che, in base al modo stabilito, riconosce la sequenza 1001. Pertanto, il riconoscitore viene descritto a livello strutturale e dispone di 3 componenti:

- Il debouncer (descritto a livello comportamentale);
- Un gestore per il modo (descritto a livello strutturale);
- Il sistema che si occupa del riconoscimento vero e proprio (descritto a livello comportamentale).

Sistema per il riconoscimento

Il sistema che effettua il riconoscimento viene descritto a livello comportamentale.

```
entity sistema is
    Port ( CLK : in STD_LOGIC;
           RESET : in STD_LOGIC := '0';
           I, CBD: in STD_LOGIC;
           M : in STD_LOGIC := '1';
           Y : out STD_LOGIC);
end sistema;

architecture Behavioral of sistema is
type STATUS is (S0, S1, S2, S3, S4, S5, S6, S7);
SIGNAL MODE : STD_LOGIC := '1';
SIGNAL PS : STATUS;
```

Figure 22: Codice Sistema Pt1

Nella descrizione è definito un tipo enumerativo "status" che contiene tutti i possibili stati della macchina: S0, S1, S2, S3, S4, S5, S6, S7. Per quanto riguarda il modo, nello specifico, se M=1 vengono considerati i primi 5 stati, se M=0 vengono considerati tutti. Tra questi, lo stato S4 viene raggiunto se la sequenza 1001 è stata riconosciuta. Il comportamento della macchina è definito da un process sensibile al clock: Se il segnale di RESET è alto, allora resetta lo stato corrente della macchina, riportandolo a S0, e permette di acquisire il nuovo modo M. Se M=1,

- Stato S0: se si riceve in ingresso 0, si permane in S0; se si riceve 1, si va in S1, poiché è stato riconosciuto un 1 (prima cifra della sequenza cercata);
- Stato S1: se si riceve 1, si permane in S1, poiché non è la prossima cifra cercata, ma l'ultima trovata è un 1; se si riceve 0, si va in S2 ed è stata riconosciuta la sequenza 10;
- Stato S2: se si riceve 1, si ritorna in S1; se si riceve 0, si va in S3 ed è stata riconosciuta 100;
- Stato S3: se si riceve 0, si ritorna in S0, poiché sarebbe stata riconosciuta la sequenza 1000, che non è quella che si ricercava; se si riceve 1, si va in S4 e si riconosce proprio la sequenza 1001.

Se M=0, c'è un concetto di conteggio su 4 bit, pertanto anche in caso di valore non ricercato in ingresso, si procede in avanti verso altri stati. Nello specifico,

- Stato S0: se si riceve 1, come prima si va in S1; altrimenti si va in S5;
- Stato S1: se si riceve 0, si va in S2; altrimenti si va in S6;
- Stato S2: se si riceve 0, si va in S3; altrimenti in S7;
- Stato S3: se si riceve 1, si va in S4 e la sequenza 1001 è stata riconosciuta; altrimenti si torna in S0 perché su 4 bit è stata riconosciuta la sequenza 1000, che non è quella cercata.

Per gli stati successivi non importa il bit ricevuto in ingresso, in quanto a gruppi di 4 bit alla volta non sarebbe riconosciuta la sequenza ricercata, pertanto

- Stato S5: va in S6;
- Stato S6: va in S7;
- Stato S7: va in S0.

```
begin

delta : process (clk)
begin
if ( CLK = '1' and CLK'event) then
    if ( RESET = '1') then
         PS <= S0;
        if(M = '1') then
            MODE <= '1';
        else
            MODE <= '0';
        end if;
    elsif(CBD = '1') then
        if (MODE = '1') then
            case PS is
                when S0 | S4 =>
                    if ( I = '0') then
                        PS <= S0;
                    else
                        PS <= S1;
                    end if;
                 when S1 =>
                    if ( I = '0') then
                        PS <= S2;
                    else
                        PS <= S1;
                    end if;
                when S2 =>
                    if ( I = '0') then
                        PS <= S3;
                    else
                        PS <= S1;
                    end if;
```

```
            when S3 =>
               if ( I = '0') then
                   PS <= S0;
               else
                   PS <= S4;
               end if;
            when others =>
               PS <= S0; -- Error

         end case;
      else
         case PS is
            when S0 | S4 =>
               if ( I = '0') then
                   PS <= S5;
               else
                   PS <= S1;
               end if;
            when S1 =>
               if ( I = '0') then
                   PS <= S2;
               else
                   PS <= S6;
               end if;
            when S2 =>
               if ( I = '0') then
                   PS <= S3;
               else
                   PS <= S7;
               end if;
             when S3 =>
               if ( I = '0') then
                   PS <= S0;
               else
                   PS <= S4:



                 end if;
                when S5 =>
                      PS <= S6;
                when S6 =>
                      PS <= S7;
               when S7=>
                      PS <= S0;
               when others =>
                      PS <= S0; -- Error
            end case;
         end if;
      end if;
   end if;
end process;

with PS select
   Y <= '1' when S4,
        '0' when others;

end Behavioral;
```

Figure 23: Codice Sistema Pt2

Debouncer

Tale componente ha il compito di trasformare un segnale rumoroso in un segnale pulito. Quando un segnale arriva da un bottone sarà sicuramente effetto da rumore, come riportato in figura. Ad un occhio umano, tali oscillazioni non vengono percepite ma, quando tale segnale viene analizzato ad un microcontrollore, queste vengono rilevate a pieno e potrebbero creare problemi (se tale segnale va in ingresso ad un contatore si traduce in conteggi spuri). Il debouncer viene progettato come una macchina a stati:

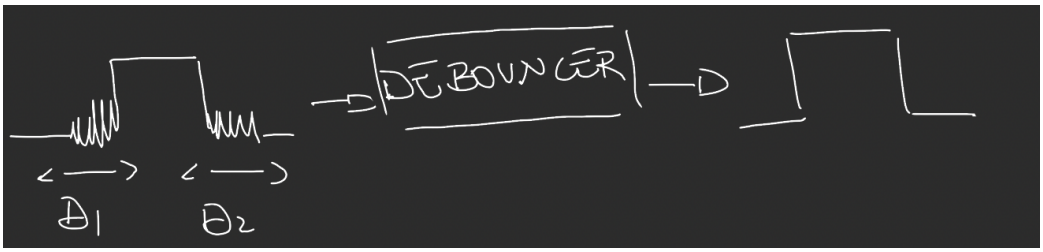

Figure 24: Debouncer

- Stato not pressed: il sistema permane in questo stato finchè non vede il segnale in ingresso alzarsi. Passa così nello stato pressed.
- Stato pressed: il sistema permane in questo stato finchè il segnale in ingresso non si abbassa. Quando ciò accade, si utilizza una variabile di conteggio, la quale fa si che passi un periodo di tempo pari a D2 prima di riportare il segnale come alto in uscita e tornare nello stato not pressed.

Il segnale in uscita rimane alto per un periodo di clock dato che, quando il sistema ritorna nello stato not pressed, l'uscita viene abbassata nuovamente. A differenza di un debouncer classico, l'uscita viene riportata alta solo dopo che il segnale si è abbassato e non dopo un tempo D1 dal fronte di salita; la scelta è stata fatta poiché, altrimenti, se il bottone venisse premuto troppo a lungo potrebbe essere rilevato nuovamente ed in uscita si produrrebbero 2 segnali puliti invece di 1.

Filtro per il modo

Questo componente prende in ingresso il segnale filtrato dal debouncer, chiamato "cleared_button", il clock e l'input dallo switch S2, che rappresenta il modo, e restituisce il modo. Viene descritto a livello comportamentale: se il bottone B2 è premuto, il modo viene impostato tramite S2.

```
entity filtro_modo is
    Port ( cleared_button : in STD_LOGIC;
           s2,clk : in STD_LOGIC;
           modo : out STD_LOGIC);
end filtro_modo;

architecture Behavioral of filtro_modo is

signal temp: std_logic;
begin

filtro: process(clk)
begin
if(clk='1' and clk'event) then
    if( cleared_button = '1') then
        modo <= s2;
   end if;
end if;
end process;
end Behavioral;
```

Figure 25: Codice filtro modo

Gestore per il modo

Il componente relativo alla gestione del modo viene descritto a livello strutturale e si compone di un debouncer e di un filtro per il modo. Prende in ingresso il clock, il segnale del bottone B2, l'input dallo switch S2. B2 va in ingresso al debouncer per la pulizia del segnale e l'uscita del debouncer, attraverso il segnale interno cb_temp va in ingresso al filtro insieme a S2. L'uscita sarà data dal filtro, pertanto sarà il modo, che successivamente sarà dato in ingresso al sistema di riconoscimento.

```
entity gestore_modo is
    Port ( b2 : in STD_LOGIC;
           s2 : in STD_LOGIC;
           clk : in STD_LOGIC;
           modo : out STD_LOGIC);
end gestore_modo;

architecture Structural of gestore_modo is
component db PORT(button : in STD_LOGIC;
          clk : in STD_LOGIC;
          cleared_button : out STD_LOGIC);
end component;

for all: db use entity work.debouncer(Behavioral);

component fm PORT(cleared_button : in STD_LOGIC;
          s2,clk : in STD_LOGIC;
          modo : out STD_LOGIC);
end component;

for all: fm use entity work.filtro_modo(Behavioral);


signal cb_temp: std_logic;

begin

db2: db PORT MAP(button => b2, clk => clk, cleared_button => cb_temp);

fm1: fm PORT MAP(cleared_button => cb_temp, clk => clk, s2 => s2, modo => modo);

end Structural;
```

Figure 26: Codice gestore modo

Riconoscitore

Il riconoscitore complessivo viene descritto in modo Strutturale tramite i componenti presentati finora.

```
entity riconoscitore is
    Port ( b1,b2,s1,s2,clk,reset : in STD_LOGIC;
           u : out STD_LOGIC);
end riconoscitore;

architecture Structural of riconoscitore is

component sistem PORT(clk : in STD_LOGIC;
                      reset : in STD_LOGIC := '0';
                      i,CBD : in STD_LOGIC;
                      m : in STD_LOGIC := '1';
                      y : out STD_LOGIC);
end component;

for all: sistem use entity work.sistema(Behavioral);



component gmodo PORT(b2 : in STD_LOGIC;
                     s2 : in STD_LOGIC;
                     clk : in STD_LOGIC;
                     modo : out STD_LOGIC);
end component;

for all: gmodo use entity work.gestore_modo(Structural);



component deb PORT(button : in STD_LOGIC;
                   clk : in STD_LOGIC;
                   cleared_button : out STD_LOGIC);
end component;

for all: deb  use entity work.debouncer(Behavioral);
```

```
signal CBD : std_logic;
signal cleared_reset : std_logic;
signal modo: std_logic;

begin

sistem1: sistem PORT MAP ( clk => clk, i => s1, m => modo, reset=> cleared_reset, CBD => CBD, y => u);

deb_reset: deb PORT MAP( clk => clk, button => reset, cleared_button => cleared_reset);
deb1: deb PORT MAP( clk => clk, button => b1, cleared_button => CBD);

gmodo1: gmodo PORT MAP( clk => clk, b2 => b2, s2 => s2, modo => modo);

end Structural;
```

Figure 27: Codice riconoscitore

Simulazione

Per la simulazione si è usato il seguente testbench sul riconoscitore completo, compreso di sistema di riconoscimento, debouncer e gestore del modo:

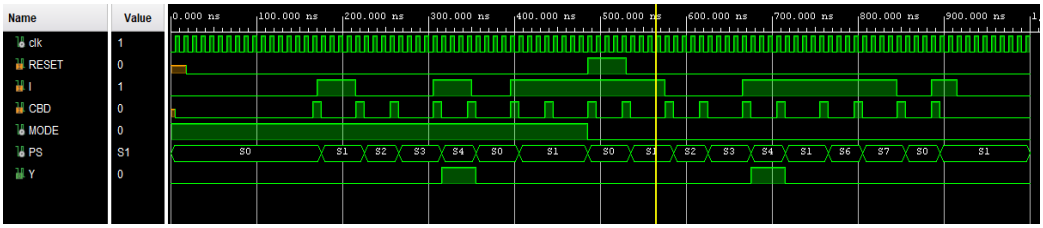

Figure 28: Simulazione riconoscitore

4 Esercizio 4 - Shift Register

4.1 Traccia

Shift register

L'esercizio richiede la progettazione di uno shift register, mediante un approccio sia comportamentale che strutturale, con le seguenti caratteristiche:

- Modo variabile: lo shift register, sulla base di un segnale in ingresso, deve effettuare shift verso destra o verso sinistra.
- Shift variabile: sulla base di un segnale in ingresso, il registro deve poter variare il numero di posizioni di shift.

```
entity shift_register_bidirezionale is
    Generic( N: positive :=6);
    Port ( input : in STD_LOGIC;
           output : inout STD_LOGIC_VECTOR (1 to N);
           clk : in STD_LOGIC;
           mode,a : in bit;--0 sinsitra, 1 destra;
           reset : in STD_LOGIC;
           shift: in bit:='0');-- 0 shifta di 1, 1 shifta di 2
end shift_register_bidirezionale;
```

Figure 29: Entity Shift Bidirezionale

4.2 Approccio comportamentale

Per la progettazione di tipo comportamentale è stato utilizzato il costrutto *if*, dato che sia le opzioni di modo che di shift comprendevano solo due possibilità; in un caso più generale, il costrutto case sarebbe più appropriato. Lo shift register realizzato prevede un ingresso seriale ed un output parallelo, gestiti nel seguente modo:

- Ingresso mode: tale segnale varia la modalità di shift: se pari ad 1, il valore viene inserito da destra, altrimenti da sinistra.

- Ingresso a: tale segnale funge da abilitazione, ovvero il registro effettua uno shift solo se, sul fronte di salita del clock, rileva un'abilitazione pari ad 1.
- Ingresso shift: è il segnale che gestisce il numero di posizioni dello shift: quando è pari a 0, avviene lo shift di una posizione, altrimenti di due.

```
architecture Behavioral of shift_register_bidirezionale is
signal reg: std_logic_vector (1 to N);
begin
main: process (clk)
begin
    if (clk = '1' and clk'event) then
        if ( reset = '1') then
           reg <= (others => '0');
        else
             if ( mode = '1' ) then
                  if(a='1') then
                      if(shift = '0') then
                          reg <= input & reg(1 to N-1);
                       elsif(shift = '1') then
                           reg <= input & '0' & reg(1 to N-2);
                       end if;
                  end if;
              else
                   if(a='1') then
                      if(shift = '0') then
                          reg <= reg(2 to N) & input;
                       elsif(shift = '1') then
                           reg <= reg(3 to N)& '0' & input;
                       end if;
                   end if;
              end if;
          end if;
        end if;


                  end process;

                   output <= reg;

                  end Behavioral;
```

Figure 30: Architecture Shift Register Bidirezionale

4.3 Approccio Strutturale

Per la realizzazione tramite un approccio strutturale, sono stati realizzati dei flip flop D bidirezionali.

Component Ffd_bidirezionali:

```
entity ffd_bidirezionale is
    Port ( leftInput, rightInput, clk, reset : in STD_LOGIC;
           mode, a: in bit:='0';
           output : out STD_LOGIC);
end ffd_bidirezionale;
```

Figure 31: Entity Flip Flop D

La logica di tale componente è molto semplice:
prevede 2 ingressi (leftInput e RightInput) e, in base al segnale di mode in ingresso, decide quale riportare in uscita. Tale componente è sincrono e lavora solo con abilitazione pari ad 1.

```
architecture Behavioral of ffd_bidirezionale is
signal temp: std_logic;
begin

delta: process(clk)
     begin
        if(clk='1' and clk'event) then
           if(reset='1') then
                temp <= '0';
            elsif(a='1') then
                if(mode ='0') then
                    temp <= rightInput;
                else
                    temp <= leftInput;
                end if;
            end if;
        end if;
     end process;
output <= temp;
end Behavioral;
```

Figure 32: Architecture Flip Flop D

Component mux_2_1:

Tale componente è necessario per selezionare quale valore deve essere riportato in ingresso ai flipflop, in base allo shift richiesto. Dato che lo shift può variare al massimo di una posizione, un multiplexer 2:1 è sufficiente; in un caso più generale, si potrebbe ricorrere ad un multiplexer con un numero maggiore di ingressi.

```
entity mux_2_1 is
    Port ( x1 : in STD_LOGIC;
           x2 : in STD_LOGIC;
           s : in bit;
           y : out STD_LOGIC);
end mux_2_1;

architecture Behavioral of mux_2_1 is
begin

y<= x1 when (s='0') else x2;

end Behavioral;
```

Figure 33: Mux 2:1

Vediamo ora come tali componenti sono stati combinati:

```
 ff_with_left0:if i=2 generate
                    mux_right: mux_2_1 port map( x1 => output(i+1), x2 => output(i+2), s => shift, y => muxRight(i));
                    mux_left: mux_2_1 port map( x1 => output(i-1), x2 => '0', s => shift, y => muxleft(i));
                    ff: ffd_bidirezionale PORT MAP(a => a, leftInput => muxleft(i) , clk => clk, reset => reset,mode => mode,
                                                   output => output(i), rightInput => muxRight(i));
 end generate ff_with_left0;

 ff_with_rigth0:if i= N-1 generate
                    mux_right: mux_2_1 port map( x1 => output(i+1), x2 => '0', s => shift, y => muxRight(i));
                    mux_left: mux_2_1 port map( x1 => output(i-1), x2 => output(i-2), s => shift, y => muxleft(i));
                    ff: ffd_bidirezionale PORT MAP(a => a, leftInput => muxleft(i) , clk => clk, reset => reset,mode => mode,
                                                   output => output(i), rightInput => muxRight(i));
 end generate ff_with_rigth0;




           ffintermedi:if i > 2 and i < N-1 generate
                           mux_right: mux_2_1 port map( x1 => output(i+1), x2 => output(i+2), s => shift, y => muxRight(i));
                           mux_left: mux_2_1 port map( x1 => output(i-1), x2 => output(i-2), s => shift, y => muxleft(i));
                           ff: ffd_bidirezionale PORT MAP(a => a, leftInput => muxleft(i) , clk => clk, reset => reset,mode => mode,
                                                          output => output(i), rightInput => muxRight(i));
           end generate ffintermedi;

end generate;

end Structural;
```

Figure 34: Generate FFD

- **PrimoFF:** è il primo flip flop da sinistra, ovvero quello di posizione 1. È stato differenziato dagli altri poiché presenterà il multiplexer solo sull'ingresso destro, dato che il sinistro è legato all'input del registro.
- **UltimoFF:** è il primo flip flop da destra, ovvero quello di posizione N. È stato differenziato dagli altri poiché presenterà il multiplexer solo sull'ingresso sinistro, dato che il destro è legato all'input del registro.
- **FF_with0:** sono i flipflop che, nel multiplexer per la selezione, presentano degli 0, in quanto non hanno abbastanza flipflop che li precedono. Es: nel flipflop di posizione 2, per l'ingresso sinistro, la scelta ricadrà tra l'uscita del flipflop di posizione 1 ed uno 0, dato che non esiste un flipflop di posizione 0.
- **FFintermedi:** sono quei flipflop che presentano multiplexer si per l'ingresso di destra che di sinistra, ed entrambi prendono gli ingressi da flipflop precedenti.

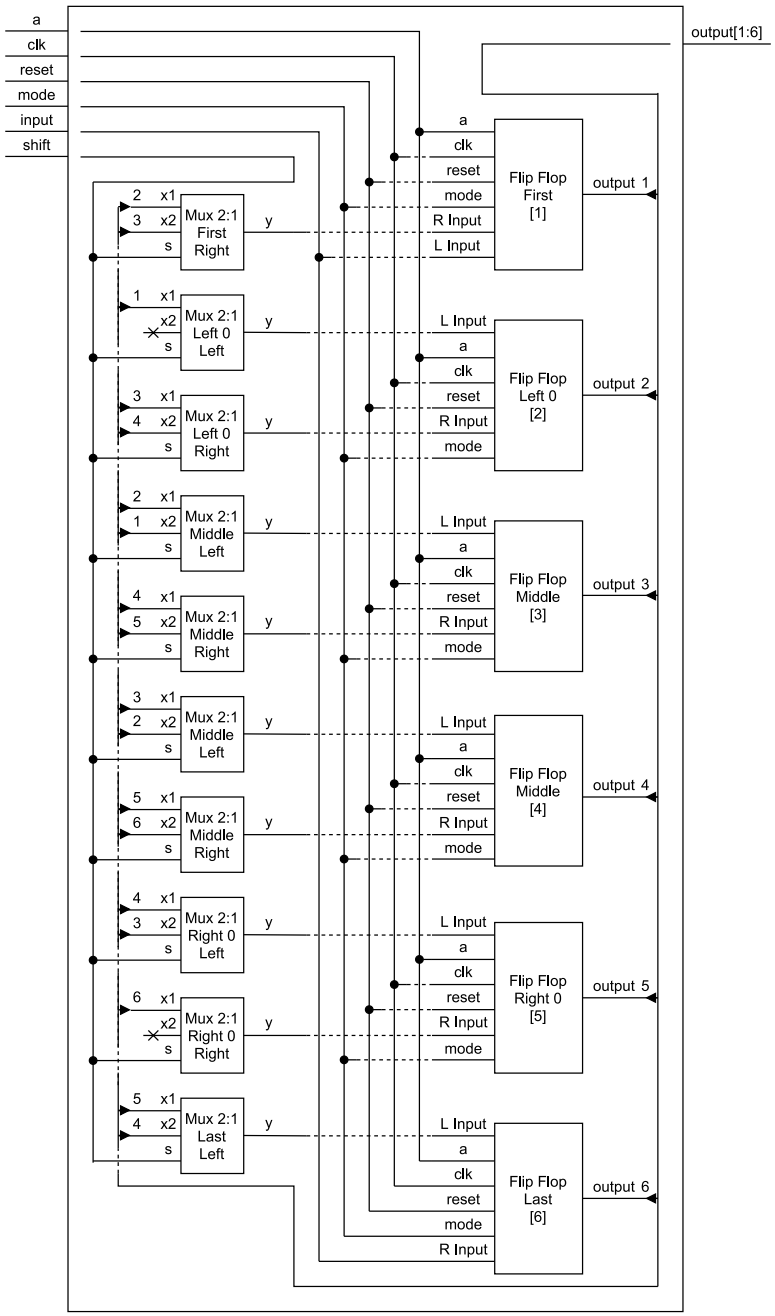

Figure 35: Progetto Completo

4.4 Simulazione

Verrà ora presentata una simulazione del funzionamento della macchina, verificando tutte le combinazioni dei segnali mode e shift. I risultati della simulazione sono analoghi per entrambe le architecture.

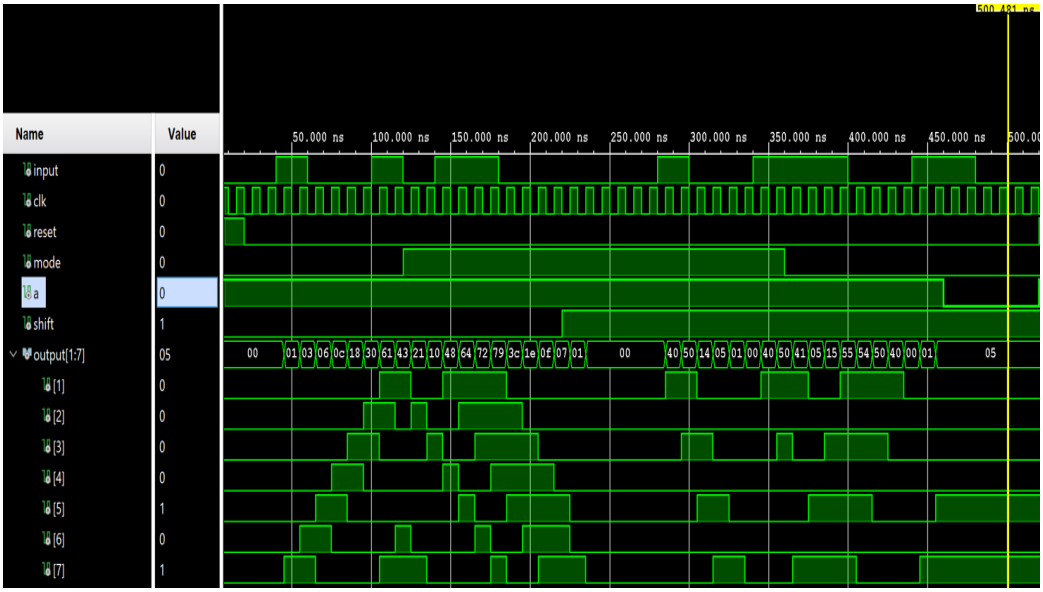

Figure 36: Simulazione Shift Register

5 Esercizio 5 - Cronometro

6 Esercizio 6 - Sistema di Testing

7 Esercizio 7 - Comunicazione con Handshaking

8 Esercizio 8 - Processor

9 Esercizio 9 - Interfaccia UART

10 Esercizio 10 - Switch Multistadio

11 Esercizio 11 - Divisore Restoring

12 Esercizio 12 - Interfaccia VGA

List of Figures