Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

Progetto di Architettura dei Sistemi di Elaborazione

Gruppo 56

Simone D'Orta (M63001283)

Antonio Savino (M63001349)

Sommario

Capitolo 1

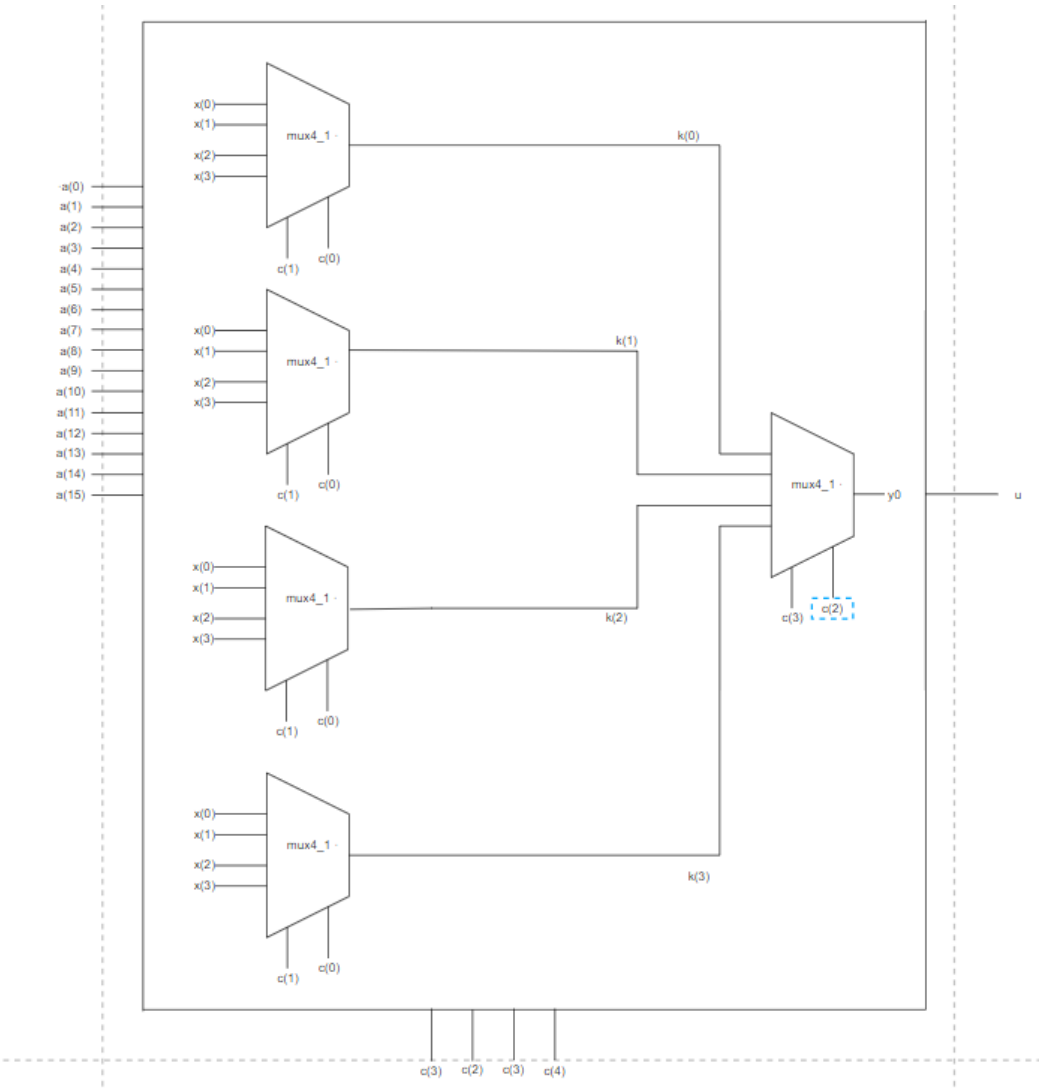
Rete 16:4

1.1 Traccia 1

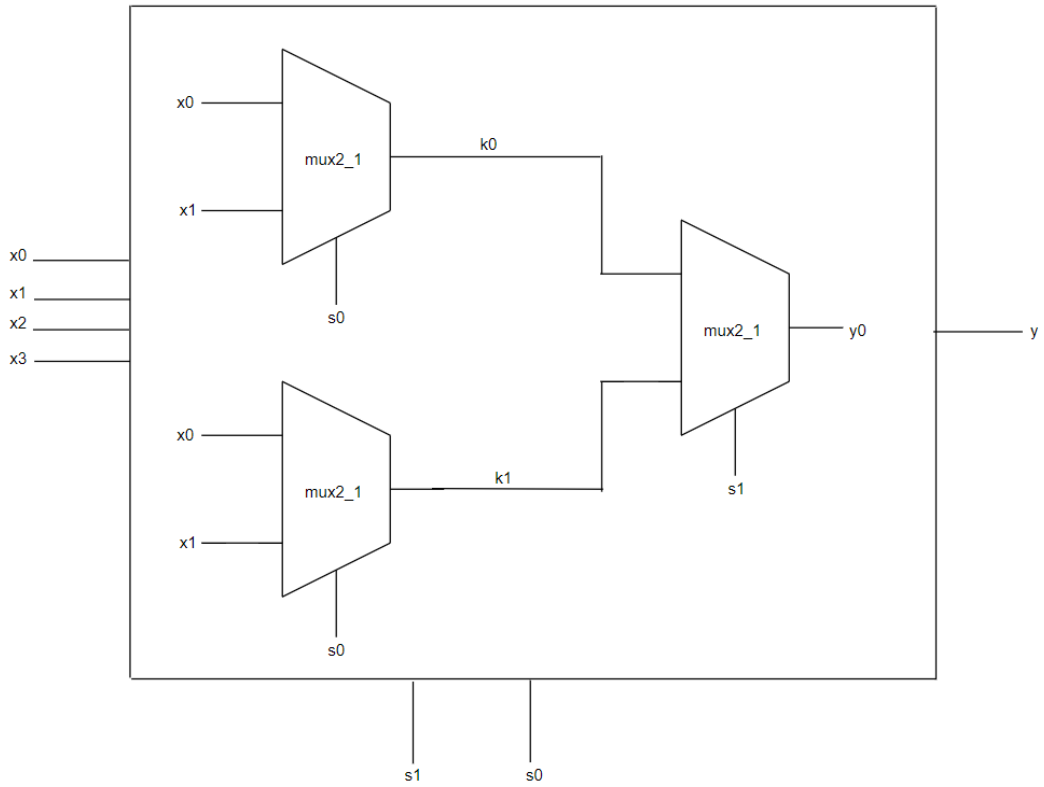
Progettare, implementare in VHDL e testare mediante simulazione un multiplexer indirizzabile 16:1, utilizzando un approccio di progettazione per composizione a partire da multiplexer 4:1.

1.1.1 Soluzione

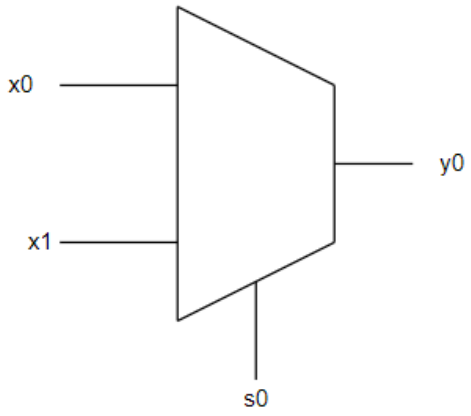
Utilizzando un approccio modulare abbiamo progettato il multiplexer 16:1 tramite composizione di multiplexer 4:1 e 2:1, ai quali arriva un segnale di selezione diverso in base al loro stadio.



A sua volta il multiplexer 4:1 è progettato anch'esso attraverso composizione di multiplexer 2:1.



Quest'ultimo è l'unico descritto ad un livello di astrazione più basso, rtl, in modo tale da garantirne la sintesi.



1.1.2 Codice

Inizialmente siamo partiti da una descrizione rtl del multiplexer 2:1, il componente fondamentale alla base di tutti gli altri multiplexer.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity mux2_1 is
    Port ( x0 : in STD_LOGIC;
           x1 : in STD_LOGIC;
           s0 : in STD_LOGIC;
           y0 : out STD_LOGIC);
end mux2_1;

architecture rtl of mux2_1 is

begin
    y0 <= x0 when s0 = '0' else
        x1 when s0 = '1' else
        '-';

end rtl;
```

Definiamo in primo luogo l'entity, composta da tre ingressi e un'uscita STD_LOGIC. Descriviamo nell'architettura le trasformazioni che subiscono i segnali. L'uscita y0 assume il valore differenti in base al valore dell'ingresso di selezione s0, considerando anche i casi eccezionali nei quali s0 assume un valore diverso da 0 o 1.

Tramite la composizione di tre multiplexer 2:1, è stato possibile descrivere strutturalmente il multiplexer 4:1.

```
entity mux4_1 is
    Port ( x : in STD_LOGIC_VECTOR (0 to 3);
           s : in STD_LOGIC_VECTOR (1 downto 0);
           y : out STD_LOGIC);
end mux4_1;

architecture structural of mux4_1 is

component mux2_1 is
    port ( x0 : in STD_LOGIC;
           x1 : in STD_LOGIC;
           s0 : in STD_LOGIC;
           y0 : out STD_LOGIC);
end component;

signal k : STD_LOGIC_VECTOR(0 to 1);

begin

mux01: for i in 0 to 1 GENERATE m: mux2_1
    port map(
    x0=>x(i*2),
    x1=>x(i*2+1),
    s0=>s(0),
    y0=>k(i)
    );
end GENERATE;

mux2: mux2_1
    port map(
    x0=>k(0),
    x1=>k(1),
    s0=>s(1),
    y0=>y
    );

end structural;
```

Tramite un for generate istanziamo i due multiplexer del primo stadio ai quali arriva in ingresso di selezione il bit meno significativo s(0). L'uscita dei due multiplexer entra in ingresso al multiplexer del secondo stadio, la cui uscita coincide poi con l'uscita effettiva del multiplexer 4:1.

Dalla composizione di cinque multiplexer 4:1, riusciamo ad arrivare alla descrizione strutturale del componente finale, ovvero il multiplexer 16:1.

```
entity mux16_1 is
    Port ( a : in STD_LOGIC_VECTOR (0 to 15);
           c : in STD_LOGIC_VECTOR (3 downto 0);
           u : out STD_LOGIC);
end mux16_1;

architecture structural of mux16_1 is

component mux4_1 is
    port ( x : in STD_LOGIC_VECTOR (0 to 3);
           s : in STD_LOGIC_VECTOR (1 downto 0);
           y : out STD_LOGIC
           );
end component;

signal k : STD_LOGIC_VECTOR(0 to 3);

begin

mux03: for i in 0 to 3 GENERATE m: mux4_1
    port map(
    x(0 to 3) => a(i*4 to i*4+3),
    s(1 downto 0) => c(1 downto 0),
    y => k(i)
    );
end GENERATE;

mux4: mux4_1
    port map(
    x(0 to 3) => k(0 to 3),
    s(1 downto 0) => c(3 downto 2),
    y => u
    );

end structural;
```

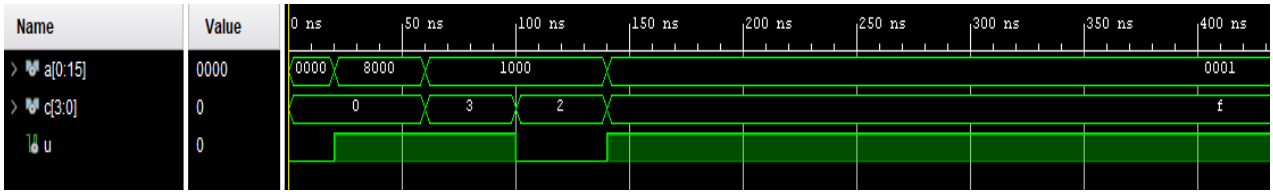
Nuovamente tramite un for generate istanziamo i multiplexer 4:1 del primo stadio, i quali produrranno 4 segnali che entreranno in ingresso al multiplexer del secondo stadio.

1.1.3 Simulazione

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica. Sono stati considerati tre casi di test, il cui risultato è consultabile nell'immagine sottostante.

```vhdl
entity mux16_1_tb is
end;

architecture bench of mux16_1_tb is

  component mux16_1
      Port ( a : in STD_LOGIC_VECTOR (0 to 15);
             c : in STD_LOGIC_VECTOR (3 downto 0);
             u : out STD_LOGIC);
  end component;

  signal a: STD_LOGIC_VECTOR (0 to 15) := "0000000000000000";
  signal c: STD_LOGIC_VECTOR (3 downto 0) := "0000";
  signal u: STD_LOGIC;

begin

  uut: mux16_1 port map ( a => a,
                          c => c,
                          u => u );

  stimulus: process
  begin

    --linea 0_1
    wait for 20 ns;
    a <= "1000000000000000";
    c <= "0000";
    wait for 20 ns;
    assert u <= '1'
    report "errore"
    severity failure;

    --linea 3_1
    wait for 20 ns;
    a <= "0001000000000000";
    c <= "0011";
    wait for 20 ns;
    assert u <= '1'
    report "errore"
    severity failure;

    --linea 2_0
    wait for 20 ns;
    c <= "0010";
    wait for 20 ns;
    assert u <= '0'
    report "errore"
    severity failure;

    --linea 15_1
    wait for 20 ns;
    a <= "0000000000000001";
    c <= "1111";
    wait for 20 ns;
    assert u <= '1'
    report "errore"
    severity failure;

    wait;
  end process;


end;
```

Possiamo vedere il risultato della simulazione nell'immagine sottostante:

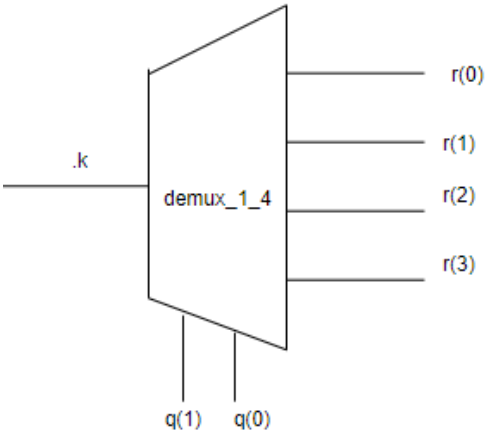


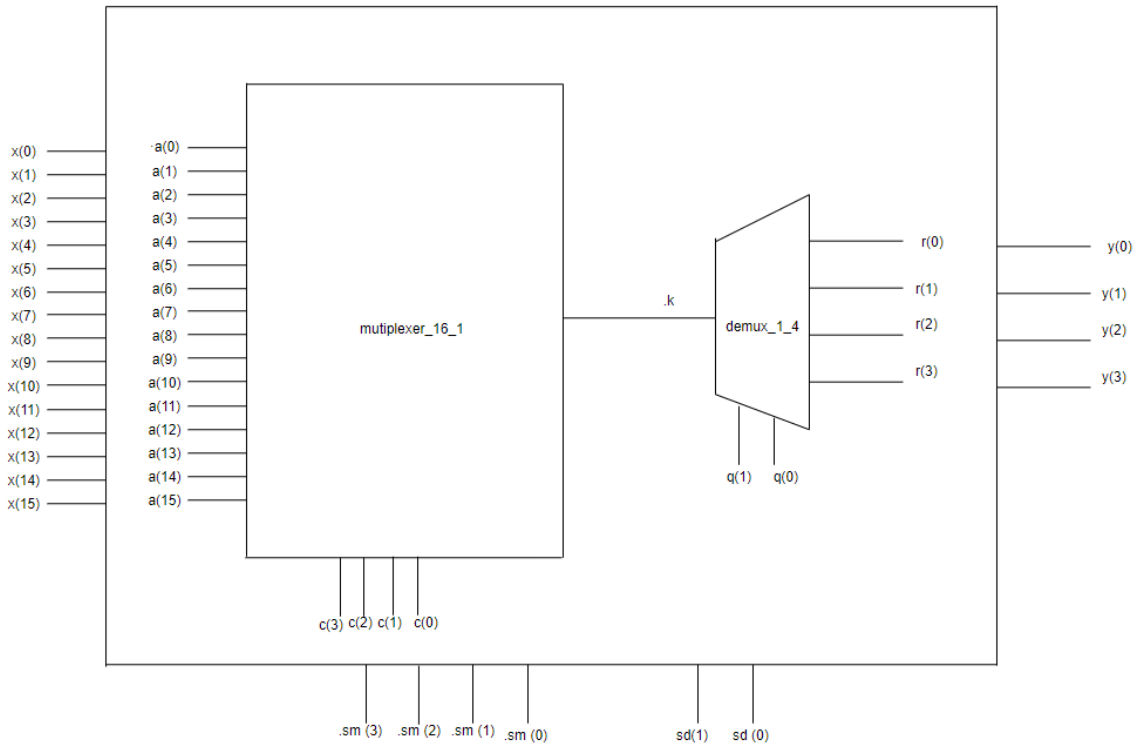
1.2 Traccia 2

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

1.2.1 Soluzione

Partendo dal multiplexer 16:1 sviluppato in precedenza e tramite un ulteriore componente, il demultiplexer 1:4, è stato possibile progettare la rete 16:4.





1.2.2 Codice

È stato necessario prima di tutto definire il demultiplexer a livello rtl, il quale riceve in ingresso un unico segnale, riprodotto in una delle sue quattro uscite in base al valore dell'ingresso di selezione.

```vhdl
entity demux1_4 is
    Port ( p : in STD_LOGIC;
           q : in STD_LOGIC_VECTOR(1 downto 0);
           r : out STD_LOGIC_VECTOR(0 to 3));
end demux1_4;

architecture Behavioral of demux1_4 is

begin

    process(p,q)
    begin
        if q="00" then
            r <= p&"000";
        elsif q="01" then
            r <= '0'&p&"00";
        elsif q="10" then
            r <= "00"&p&'0';
        elsif q="11" then
            r <= "000"&p;
    end if;
    end process;

end Behavioral;
```

Notiamo come l'uscita è prodotta tramite la concatenazione dell'ingresso e tutti 0, posizionati in modo diverso in base all'ingresso di selezione.

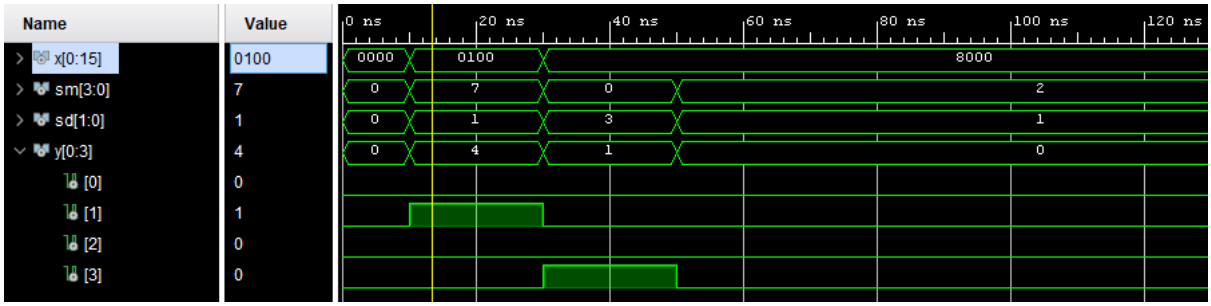
La rete 16:4 è descritta in modo strutturale a partire dal multiplexer 16:1 e dal demultiplexer 1:4.

```
entity rete_16_4 is
  Port (
  x : in STD_LOGIC_VECTOR(0 to 15);
  sm : in STD_LOGIC_VECTOR(3 downto 0);
  sd : in STD_LOGIC_VECTOR(1 downto 0);
  y : out STD_LOGIC_VECTOR(0 to 3)
  );
end rete_16_4;

architecture structural of rete_16_4 is

component mux16_1 is

  Port (
  a: in STD_LOGIC_VECTOR(0 to 15);
  c: in STD_LOGIC_VECTOR(3 downto 0);
  u: out STD_LOGIC
  );

end component;

component demux1_4 is

  Port (
  p : in STD_LOGIC;
  q : in STD_LOGIC_VECTOR (1 downto 0);
  r : out STD_LOGIC_VECTOR (0 to 3)
  );

end component;

signal k : STD_LOGIC;

begin

mux: mux16_1
    port map (
    a(0 to 15) => x(0 to 15),
    c(3 downto 0) => sm(3 downto 0),
    u => k
    );

demux: demux1_4
    port map(
    p => k,
    q(1 downto 0) => sd(1 downto 0),
    r(0 to 3) => y(0 to 3)
    );

end structural;
```

1.2.3 Simulazione

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica. Sono stati considerati vari casi di test, il cui risultato è consultabile nell'immagine sottostante.

```
entity rete_16_4_tb is
end;

architecture bench of rete_16_4_tb is

  component rete_16_4
    Port (
    x : in STD_LOGIC_VECTOR(0 to 15);
    sm : in STD_LOGIC_VECTOR(3 downto 0);
    sd : in STD_LOGIC_VECTOR(1 downto 0);
    y : out STD_LOGIC_VECTOR(0 to 3)
    );
  end component;

  signal x: STD_LOGIC_VECTOR(0 to 15) := "0000000000000000";
  signal sm: STD_LOGIC_VECTOR(3 downto 0) := "0000";
  signal sd: STD_LOGIC_VECTOR(1 downto 0) := "00";
  signal y: STD_LOGIC_VECTOR(0 to 3) ;

begin

  uut: rete_16_4 port map ( x  => x,
                            sm => sm,
                            sd => sd,
                            y  => y );

  stimulus: process
  begin

    --linea 7_1
    wait for 10 ns;
    x <= "0000000100000000";
    sm <= "0111";
    sd <= "01";
    wait for 10 ns;
    assert y <= "0010"
    report "errore"
    severity failure;

    --linea 0_3
    wait for 10 ns;
    x <= "1000000000000000";
    sm <= "0000";
    sd <= "11";
    wait for 10 ns;
    assert y <= "0001"
    report "errore"
    severity failure;

    --linea 0_3
    wait for 10 ns;
    x <= "1000000000000000";
    sm <= "0010";
    sd <= "01";
    wait for 10 ns;
    assert y <= "0000"
    report "errore"
    severity failure;

    wait;
  end process;

end;
```



1.3 Traccia 3

Sintetizzare e 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 gli switch, sviluppando anche in questo caso un'apposita rete di controllo per l'acquisizione.

1.3.1 Sintesi su board

Dopo aver effettuato la simulazione, procediamo con la sintesi su board FPGA, nel nostro caso abbiamo utilizzato la board "Nexys A7-100T". Abbiamo leggermente modificato il codice della rete 16:4, rimuovendo l'ingresso X e inizializzando un segnale di tipo std_logic_vector (0 to 15):="0101110000011010", assegnato poi all'ingresso del multiplexer.

```
##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { sm[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { sm[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { sm[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { sm[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
#set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { sd[0] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { sd[1] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { y[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { y[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { y[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { y[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
```

Capitolo 2

Encoder BCD

2.1 Traccia 1

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

Input: 0000000001 -> Output: 0000 (cifra 0)

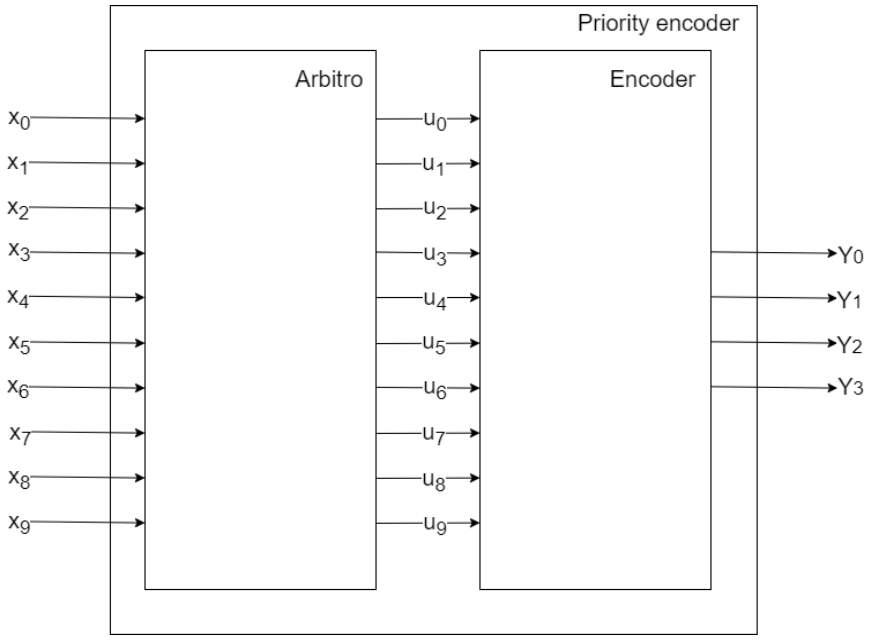
Input: 0000000010 -> Output: 0001 (cifra 1)

Input: 0000000100 -> Output: 0010 (cifra 2)

….

2.1.1 Soluzione 1

La soluzione trovata è di tipo strutturale, composta da un arbitro e dall'encoder vero e proprio. L'arbitro è necessario per stabilire una priorità nel caso più ingressi siano alti nello stesso momento, nel nostro caso la priorità più alta è stata assegnata alla cifra più alta.



2.1.2 Codice

Prima di poter definire l'architettura completa è stato ovviamente necessario prima di tutto definire le due componenti più piccole. Sia l'arbitro che l'encoder sono stati descritti a livello dataflow:

```
entity Arbiter is
--  Port ( );
    Port(
        X : in STD_LOGIC_VECTOR(9 downto 0);
        Y : out STD_LOGIC_VECTOR(9 downto 0)
       );
end Arbiter;

architecture Dataflow of Arbiter is

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


end Dataflow;
```

```
entity Encoder is
    Port(
        X : in STD_LOGIC_VECTOR(9 downto 0);
        Y : out STD_LOGIC_VECTOR(3 downto 0)
       );
end Encoder;

architecture Dataflow of Encoder is

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

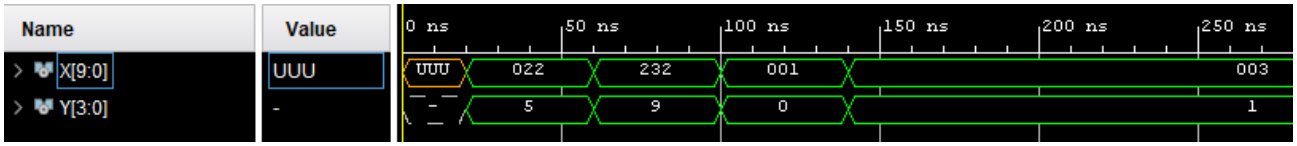
A partire da questi due moduli è stato dunque possibile definire l'Encoder a priorità: gli ingressi arrivano direttamente all'arbitro, il quale secondo la priorità prestabilita alzerà solamente una delle sue dieci linee di uscita. Queste linee entrano nell'encoder il quale produrrà in uscita la codifica binaria della cifra corrispondente alla linea alta.

```
entity PriorityEncoder is
    Port(
        X : in STD_LOGIC_VECTOR(9 downto 0);
        Y : out STD_LOGIC_VECTOR(3 downto 0)
       );
end PriorityEncoder;

architecture Structural of PriorityEncoder is
    COMPONENT Arbiter IS
        Port(
            X : in STD_LOGIC_VECTOR(9 downto 0);
            Y : out STD_LOGIC_VECTOR(9 downto 0)
           );
    END COMPONENT;

    COMPONENT Encoder IS
        Port(
            X : in STD_LOGIC_VECTOR(9 downto 0);
            Y : out STD_LOGIC_VECTOR(3 downto 0)
           );
    END COMPONENT;
    signal u : STD_LOGIC_VECTOR(9 downto 0);
begin

    A : Arbiter
        PORT MAP(
            X => X,
            Y => u
        );

    E : Encoder
        PORT MAP(
            X => u,
            Y => Y
        );

end Structural;
```

2.1.3 Simulazione

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica. I casi di test sono stati pensati sia per testare l'arbitro (quindi alzando più bit d'ingresso alla volta), sia l'encoder.

```
stimulus: process
begin

  wait for 20 ns;
  X <= "0000100010";
  wait for 20 ns;
  assert Y <= "0101"
  report "errore"
  severity failure;

  wait for 20 ns;
  X <= "1000110010";
  wait for 20 ns;
  assert Y <= "1001"
  report "errore"
  severity failure;


  wait for 20 ns;
  X <= "0000000001";
  wait for 20 ns;
  assert Y <= "0000"
  report "errore"
  severity failure;

  wait for 20 ns;
  X <= "0000000011";
  wait for 20 ns;
  assert Y <= "0001"
  report "errore"
  severity failure;

  wait;
end process;
```



2.2 Traccia 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.

2.2.1 Sintesi su board

Per la sintesi su board con uscita su led è stato necessario semplicemente modificare correttamente il file di constraint.

```
##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { X[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { X[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { X[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { X[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { X[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { X[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { X[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { X[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { X[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { X[9] }]; #IO_25_34 Sch=sw[9]

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

2.3 Traccia 3

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 Sintesi su board

In questo caso abbiamo definito un'entità Display che ha come dato in ingresso Y, che è il dato da mostrare sul display. In base al valore assunto da Y, i catodi assumono dei valori costanti definiti nell'architettura, li concateniamo poi con 1 per indicare di non accendere il punto. Per quanto riguarda l'anodo abbiamo spento tutti i display tranne uno, quindi manteniamo un solo anodo a 0 essendo gli anodi 0 attivi (anodo<="01111111").

```
entity Display is
    Port (
        clock: in std_logic;
        Y: in std_logic_vector(3 downto 0);
        cathodes: out std_logic_vector(7 downto 0);
        anodo: out std_logic_vector(7 downto 0)
     );
end Display;

architecture Behavioral of Display 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

anodo<="01111111";
```

```
process (clock)
begin
if(rising_edge(clock)) then
case Y is --per non avere il dot
      when "0000" => cathodes <= '1'&zero;
      when "0001" => cathodes <= '1'&one;
      when "0010" => cathodes <= '1'&two;
      when "0011" => cathodes <= '1'&three;
      when "0100" => cathodes <= '1'&four;
      when "0101" => cathodes <= '1'&five;
      when "0110" => cathodes <= '1'&six;
      when "0111" => cathodes <= '1'&seven;
      when "1000" => cathodes <= '1'&eight;
      when "1001" => cathodes <= '1'&nine;
      when others => cathodes <= (others => '0');
end case;
end if;
end process;
end Behavioral;
```

Successivamente abbiamo definito l'entità "PriorityEncoder" in cui abbiamo inserito il clock filter perché il display deve lavorare ad una frequenza minore.

```
entity PrioEncodDisplay is
    Port (
        clock_in, reset: in std_logic;
        X : in STD_LOGIC_VECTOR(9 downto 0);
        cathodes: out std_logic_vector(7 downto 0);
        anodo: out std_logic_vector(7 downto 0)
     );
end PrioEncodDisplay;

architecture structural of PrioEncodDisplay is

component PriorityEncoder is
    Port(
        X : in STD_LOGIC_VECTOR(9 downto 0);
        Y : out STD_LOGIC_VECTOR(3 downto 0)
       );
end component;

component Display is
    Port (
        clock: in std_logic;
        Y: in std_logic_vector(3 downto 0);
        cathodes: out std_logic_vector(7 downto 0);
        anodo: out std_logic_vector(7 downto 0)
     );
end component;
```

```
component clock_filter is
       generic(
                  CLKIN_freq : integer := 100000000; -
                  CLKOUT_freq : integer := 500       -
                        );
    Port (
           clock_in : in  STD_LOGIC;
               reset : in STD_LOGIC;
           clock_out : out  STD_LOGIC
    );
end component;

signal Y: std_logic_vector(3 downto 0);
signal clock_out: std_logic;

begin
PE: PriorityEncoder port map(X,Y);
D: Display port map(clock_out,Y,cathodes,anodo);
cf: clock_filter port map(clock_in, reset, clock_out);

end structural;
```

Infine, mostriamo come abbiamo modificato il file di constraint.

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


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { X[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { X[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { X[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { X[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { X[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { X[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { X[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { X[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { X[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports { X[9] }]; #IO_25_34 Sch=sw[9]
```

Capitolo 3

Riconoscitore di sequenze

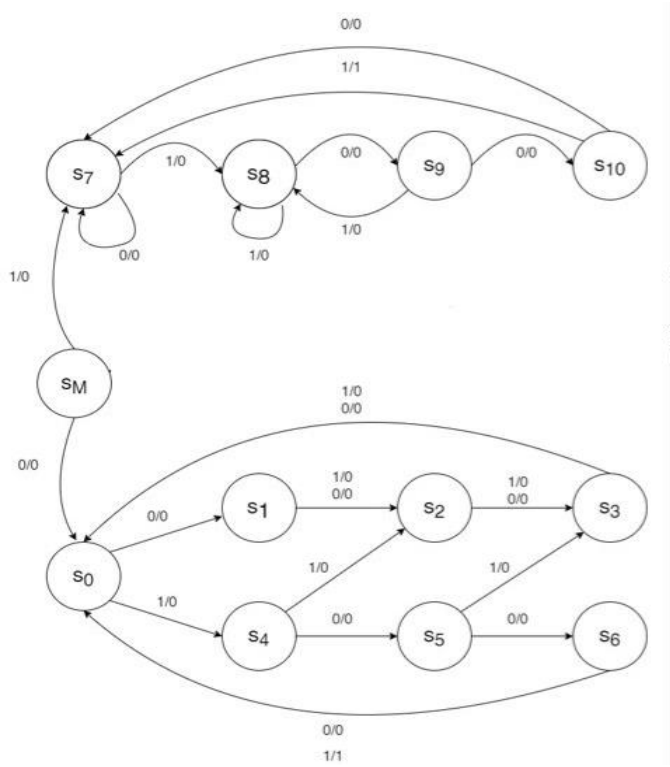
3.1 Traccia 1

Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza **1001**. La macchina prende in ingresso un segnale binario i che rappresenta il dato, un segnale A di tempificazione e un segnale M di modo, che ne disciplina il funzionamento, e fornisce un'uscita Y alta quando la sequenza viene riconosciuta. In particolare,

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

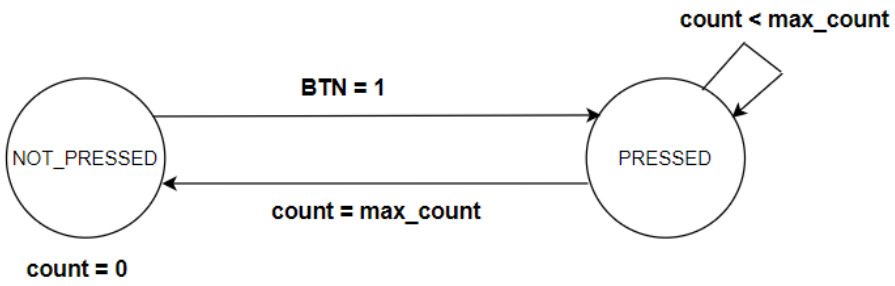
3.1.1 Soluzione 1

Per descrivere il riconoscitore è stato realizzato un automa a stati finiti composto da due macchine, una per ogni modo. Abbiamo inserito infatti lo stato Sm che determina proprio il modo dell'automa complessivo. Nel caso in cui M cambia improvvisamente non accade nulla se il segnale di reset è 0. Per passare da un automa all'altro deve variare prima il segnale di reset e poi M. Abbiamo quindi diviso la macchina con un processo combinatorio, che ci dice cosa fa la macchina, e un processo sequenziale, ovvero la memoria, per la gestione sincrona, tramite reset ed M. Rispetto all'automa descritto nella figura successiva, abbiamo inserito due nuovi stati, che sono S11 ed S12.



3.1.2 Codice

Inizialmente abbiamo definito l'entity del riconoscitore, con le relative porte di ingresso e uscita. Andando a descrivere l'architettura, abbiamo inserito dapprima il componente "ButtonDebouncer". L'obiettivo di tale componente è di generare un unico impulso di durata pari a quella del clock in corrispondenza della pressione del bottone, in quanto in assenza di tale componente potrebbe esserci del rumore sia prima che dopo il colpo di clock. Per garantire che sia generato un unico impulso per ogni pressione del bottone, è stato implementato un automa costituito da due stati, si parte da NOT_PRESSED e, appena si rileva BTN=1, si va in PRESSED dove si aspettano circa 600 millisecondi in modo da "superare" l'oscillazione:



```
type stato is (NOT_PRESSED, PRESSED);
signal BTN_state : stato := NOT_PRESSED;

constant max_count : integer := btn_noise_time/CLK_period; -- 6500000/10= conto 650000 colpi di clock

begin

deb: process (CLK)
variable count: integer := 0;

begin
   if rising_edge(CLK) then

         if( RST = '1') then
             BTN_state <= NOT_PRESSED;
             CLEARED_BTN <= '0';
         else
              case BTN_state is
                  when NOT_PRESSED =>
                      CLEARED_BTN<= '0';
                        if( BTN = '1' ) then
                              BTN_state <= PRESSED;
                        else
                              BTN_state <= NOT_PRESSED;
                        end if;
            when PRESSED =>
               if(count = max_count -1) then
                        count:=0;
                        CLEARED_BTN <= '1';
                        BTN_state <= NOT_PRESSED;
                 else
                      count:= count+1;
                      BTN_state <= PRESSED;
                 end if;
            when others =>
                BTN_state <= NOT_PRESSED;
              end case;
    end if;
```

Successivamente, nell'entity del riconoscitore, abbiamo definito i possibili stati degli automi, per poi definire i segnali di modo Sm come stato iniziale. Nel primo process si usa il costrutto CASE-WHEN per definire cosa fa la macchina sequenziale in ogni stato. Ogni valore dell'ingresso button_i fa evolvere la macchina verso un nuovo stato, specificando una determinata uscita Y.

```
-- stato iniziale SM
signal stato_corrente : stato := SM;
signal stato_prossimo : stato;

begin
bd1: ButtonDebouncer port map(RST, CLK, en_i, button_i);
bd2: ButtonDebouncer port map(RST,CLK,en_m,button_m);

stato_uscita: process(stato_corrente, i, button_i, button_m)
    begin

        -- se M = 0 (S0) : Riconoscitore di Sequenza con valutazione di sequenza per gruppi di 4 bit
        -- se M = 1 (S7): Riconoscitore di Sequenza con valutazione di sequenza ad ogni singolo bit
        case stato_corrente is
            when SM =>
               if(button_m='1') then
               if(M = '0') then
                     stato_prossimo <= S0;
                else
                     stato_prossimo <= S7;
                end if;
                end if;
            when S0 =>
                if(button_i = '1') then
                if( i = '0') then
                   stato_prossimo <= S1;
                   Y <= '0';
                else
                   stato_prossimo <= S4;
                   Y <= '0';
                end if;
                end if;
            when S1 =>
                if(button_i = '1') then
                if( i = '0' or i = '1' ) then
                   stato_prossimo <= S2;
                   Y <= '0';
                end if;
                end if;
            when S2 =>
```

```
            when S9 =>
                if(button_i = '1') then
                if( i = '0' ) then
                   stato_prossimo <= S10;
                   Y <= '0';
                else
                   stato_prossimo <= S8;
                   Y <= '0';
                end if;
                end if;
            when S10 =>
                if(button_i = '1') then
                if( i = '0' ) then
                   stato_prossimo <= S7;
                   Y <= '0';
                else
                   stato_prossimo <= S11;
                end if;
                end if;
            when S11 =>
                Y <= '1';
                if(button_i = '1') then
                if( i = '0' ) then
                   stato_prossimo <= S9;
                else
                   stato_prossimo <= S8;
                end if;
                end if;
            when S12 =>
                Y <= '1';
                if(button_i = '1') then
                if( i = '0' ) then
                   stato_prossimo <= S1;
                else
                   stato_prossimo <= S4;
                end if;
                end if;
        end case;
```

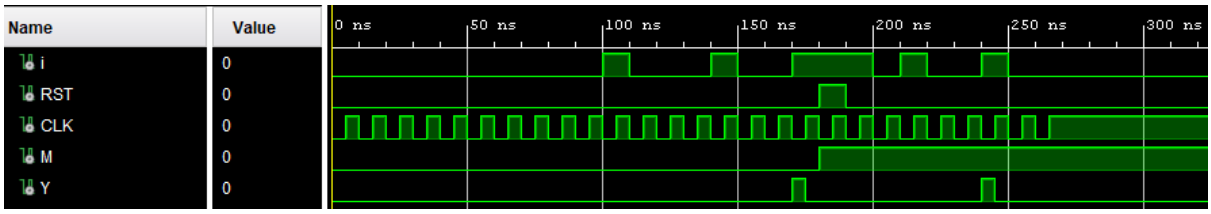
Nel secondo process, invece, andiamo a valutare quando cambia la modalità del riconoscitore. è stata realizzata infine la memoria di stato che consente di retroazionare lo stato prossimo nello stato corrente:

```
mem: process (CLK)
begin
      if(CLK'event and CLK = '1') then
            if(RST = '1') then
              stato_corrente <= SM;
           else
              stato_corrente <= stato_prossimo;
           end if;
   end if;
end process;
```

3.1.3 Simulazione

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica. Sono stati considerati diversi casi di test, il cui risultato è consultabile nell'immagine sottostante.

```
--sequenza (M=0) 10001001  (M=1)1010010
    M<='0';
    button_m<='1';
    wait for 10ns;
    button_m<='0';
    wait for 10ns;
    button_i<='1';
    i<='1';
        wait for 10 ns;
        button_i<='1';
        i<='0';
        wait for 10 ns;
        button_i<='1';
        i<='0';
        wait for 10 ns;
        button_i<='1';
        i<='0';
        wait for 10 ns;
        button_i<='1';
        i<='1';
        wait for 10 ns;
        button_i<='1';
        i<='0';
        wait for 10 ns;
        --Cambio M=1 e resetto
        rst<='1';
        wait for 10ns;
        rst<='0';
        button_m<='1';
        M<='1';
        wait for 10 ns;
        button_m<='0';
        button_i<='0';
        i<='1';
        wait for 10 ns;
        i<='0';
        wait for 10 ns;
        i<='1';
        wait for 10 ns;
        i<='0';
        wait for 10 ns;
        i<='0';
        wait for 10 ns;
        i<='1';
        wait for 10 ns;
        i<='0';
        wait for 10 ns;
stop_the_clock <= true;
wait;
```



3.2 Traccia 2

Sintetizzare e implementare su board la rete sviluppata al punto precedente, utilizzando uno switch S1 per codificare l'input i e uno switch S2 per codificare il modo M, in

combinazione con due bottoni B1 e B2 utilizzati rispettivamente per acquisire l'input da S1 a 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 Sintesi su board

Per la sintesi su board è stato necessario semplicemente modificare correttamente il file di constraint.

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


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { i }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { M }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
#set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]

## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { Y }]; #IO_L18P_T2_A24_15 Sch=led[0]
#set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]

##Buttons
#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15 Sc
#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { BTNC }]; #IO_L9P_T1_DQS_14 Sch=btnc
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { BTNU }]; #IO_L4N_T0_D05_14 Sch=btnu
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { en_m }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { RST }]; #IO_L10N_T1_D15_14 Sch=btnr
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { en_i }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

Capitolo 4

Shift Register

4.1 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}).

4.1.1 Soluzione A

Abbiamo optato per una soluzione con X=5 ed Y={1,2}. La prima soluzione adottata è stata quella di tipo Behavioral. Adottando questo tipo di soluzione è necessario, mediante un process, effettuare le dovute assegnazioni dei segnali dei flip flop.

Il registro a scorrimento è di tipo circolare, ciò vuol dire che se ad esempio stiamo considerando il FF0, il suo predecessore è il FF4.

4.1.2 Codice A

Abbiamo dapprima definito la entity del registro in cui sono stati dichiarati i segnali di ingresso e uscita. Successivamente, nell'architettura, è stato dichiarato il segnale "temp", un vettore di tipo std logic.

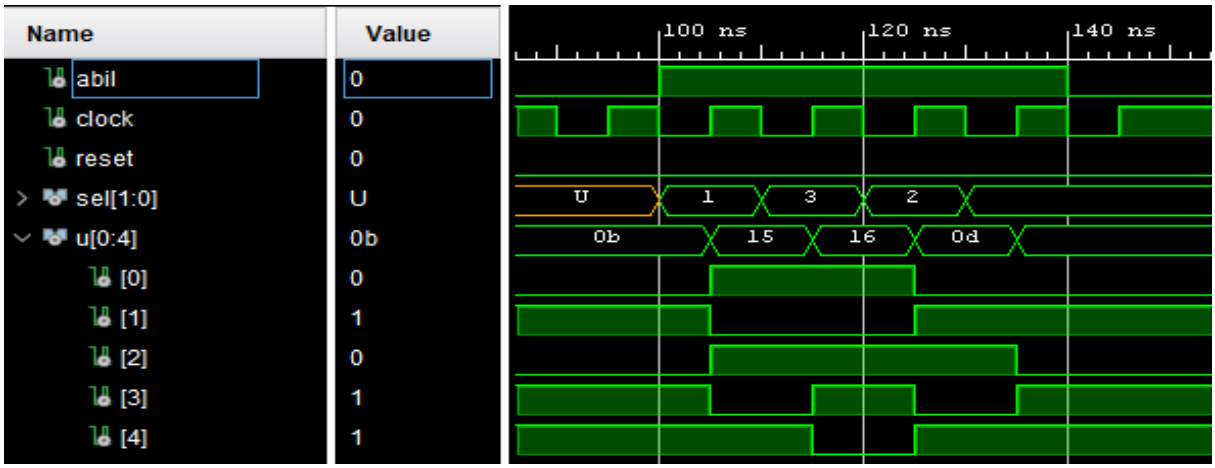
Nel process dell'architettura sono state fatte le assegnazioni dei segnali dei singoli flop flop che compongono il registro. In particolare abbiamo fatto attenzione al segnale di selezione, che può assumere diversi valori, in base ai quali il registro potrà effettuare uno o più shift verso destra o sinistra.

```
entity reg_behavioral is
    Port ( abil: in std_logic;
           clock, reset : in STD_LOGIC;
           sel : in std_logic_vector ( 1 downto 0 );
           u : out std_logic_vector ( 0 to 4 )
           );
end reg_behavioral;

architecture Behavioral of reg_behavioral is

signal temp: std_logic_vector ( 0 to 4 ) := "01011";

begin

SR: process ( clock, reset )
begin
  if ( clock'event and clock = '1' and reset = '1') then
  temp ( 0 to 4 ) <= "00000";
 elsif ( clock'event and clock = '1' and abil = '1') then
  if sel <= "00" then
  --è possibile un'assegnazione di questo tipo poichè viene
    temp(0) <= temp(3);
    temp(1) <= temp(4);
    temp(2) <= temp(0);
    temp(3) <= temp(1);
    temp(4) <= temp(2);

  elsif sel <= "01" then
    temp(0) <= temp(4);
    temp(1) <= temp(0);
    temp(2) <= temp(1);
    temp(3) <= temp(2);
    temp(4) <= temp(3);
  elsif sel <= "10" then
    temp(0) <= temp(1);
    temp(1) <= temp(2);
    temp(2) <= temp(3);
    temp(3) <= temp(4);
    temp(4) <= temp(0);

  elsif sel <= "11" then
    temp(0) <= temp(2);
    temp(1) <= temp(3);
    temp(2) <= temp(4);
    temp(3) <= temp(0);
    temp(4) <= temp(1);

    end if;
    end if;
    end process;

    u <= temp;

end Behavioral;
```

4.1.3 Simulazione A

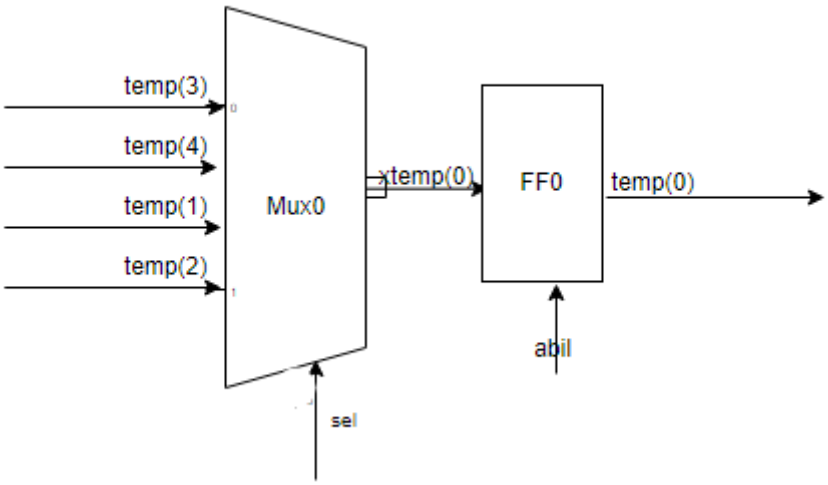
Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica. Sono stati considerati vari casi di test, il cui risultato è consultabile nell'immagine sottostante. In particolare, la selezione avviene a fronte di un segnale di abilitazione alto.

```
  stimulus: process
  begin

wait for clock_period*10;

    --shift 1 d
    sel<="01";
    abil<='1';
    wait for 10ns;
    --shift 2 s
    sel<="11";
    wait for 10ns;
    --shift 1 s
    sel<="10";
    wait for 10ns;
    --shift 2 d
    sel<="00";
    wait for 10ns;
    abil<='0';

    stop_the_clock <= true;
    wait;
  end process;

  clocking: process
  begin
    while not stop_the_clock loop
      clock <= '0', '1' after clock_period / 2;
      wait for clock_period;
    end loop;
    wait;
  end process;
```



4.1.4 Soluzione B

Abbiamo optato per una soluzione con X=5 ed Y={1,2}. Per arrivare ad una soluzione di tipo strutturale è stato necessario utilizzare dei multiplexer 4:1. Ogni flip flop del registro a scorrimento è preceduto da un multiplexer, il quale riceve in ingresso i valori degli altri quattro flip flop. In base alla selezione, l'uscita sarà pari ad uno di questi quattro valori, che salvato nel flip flop, produrrà lo shift desiderato. Il registro a scorrimento è di tipo circolare, ciò vuol dire che se ad esempio stiamo considerando il FF0, il suo predecessore è il FF4.



Come è possibile notare è stato aggiunto anche un segnale di abilitazione, solo quando quest'ultimo è alto sarà possibile effettuare lo shift.

4.1.5 Codice B

In seguito è riportato il codice della soluzione.

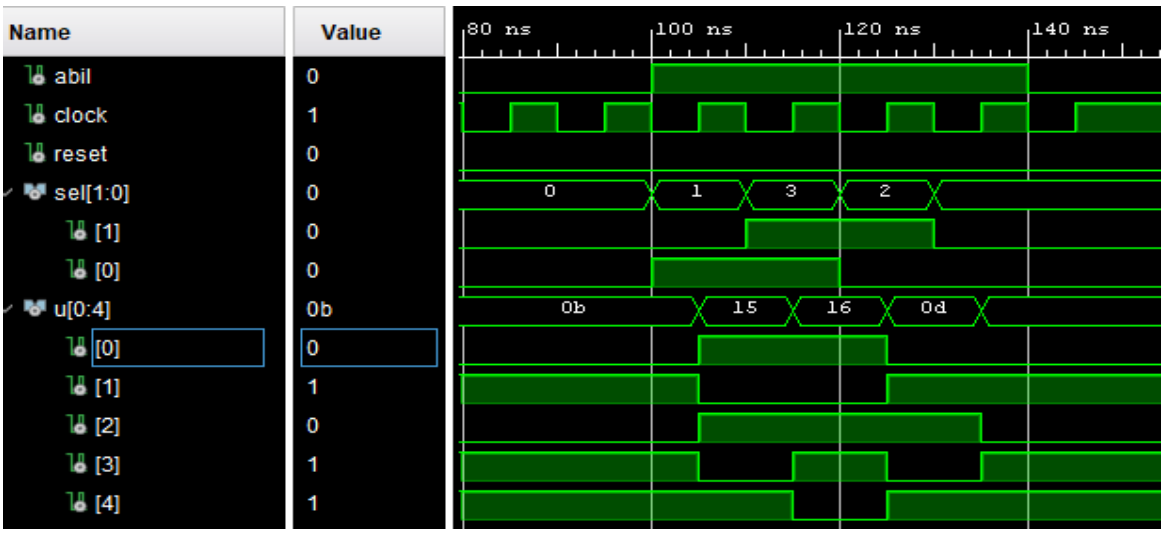
```
entity shift_reg is
    Port (
        abil: in std_logic;
        clock, reset: in std_logic;
        sel: in std_logic_vector(1 downto 0); --00 shi
        u: out std_logic_vector(0 to 4)
    );
end shift_reg;

architecture structural of shift_reg is

    component mux4_1 is
    port(
        x : in STD_LOGIC_VECTOR (0 to 3);
        s : in STD_LOGIC_VECTOR (1 downto 0);
        y : out STD_LOGIC
    );
    end component;

    signal temp: std_logic_vector(0 to 4):="01011";
    signal xtemp: std_logic_vector(0 to 4);


begin

    mux0: mux4_1
    port map(
        x(0)=>temp(3),
        x(1)=>temp(4),
        x(2)=>temp(1),
        x(3)=>temp(2),
        s=>sel,
        y=>xtemp(0)
    );

    mux1: mux4_1
    port map(
        x(0)=>temp(4),
        x(1)=>temp(0),
        x(2)=>temp(2),
        x(3)=>temp(3),
        s=>sel,
        y=>xtemp(1)
    );

    mux3: mux4_1
    port map(
        x(0)=>temp(1),
        x(1)=>temp(2),
        x(2)=>temp(4),
        x(3)=>temp(0),
        s=>sel,
        y=>xtemp(3)
    );

    mux4: mux4_1
    port map(
        x(0)=>temp(2),
        x(1)=>temp(3),
        x(2)=>temp(0),
        x(3)=>temp(1),
        s=>sel,
        y=>xtemp(4)
    );


    SR: process(clock, reset)
    begin
    if(clock'event and clock='1' and reset='1') then
        temp(0 to 4) <= "00000";
    elsif(clock'event and clock='1' and abil='1') then
        temp <= xtemp;

    end if;

    end process;

    u <= temp;

end structural;
```

Il registro a scorrimento è stato inizializzato al valore “01011”.

4.1.6 Simulazione B

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica. Sono stati considerati vari casi di test, il cui risultato è consultabile nell'immagine sottostante.

```
stimulus: process
begin

  -- Put initialisation code here
  wait for clock_period*10;

  --shift 1 d
  sel<="01";
  abil<='1';
  wait for 10ns;
  --shift 2 s
  sel<="11";
  wait for 10ns;
  --shift 1 s
  sel<="10";
  wait for 10ns;
  --shift 2 d
  sel<="00";
  wait for 10ns;
  abil<='0';


  stop_the_clock <= true;
  wait;
end process;
```



Capitolo 5

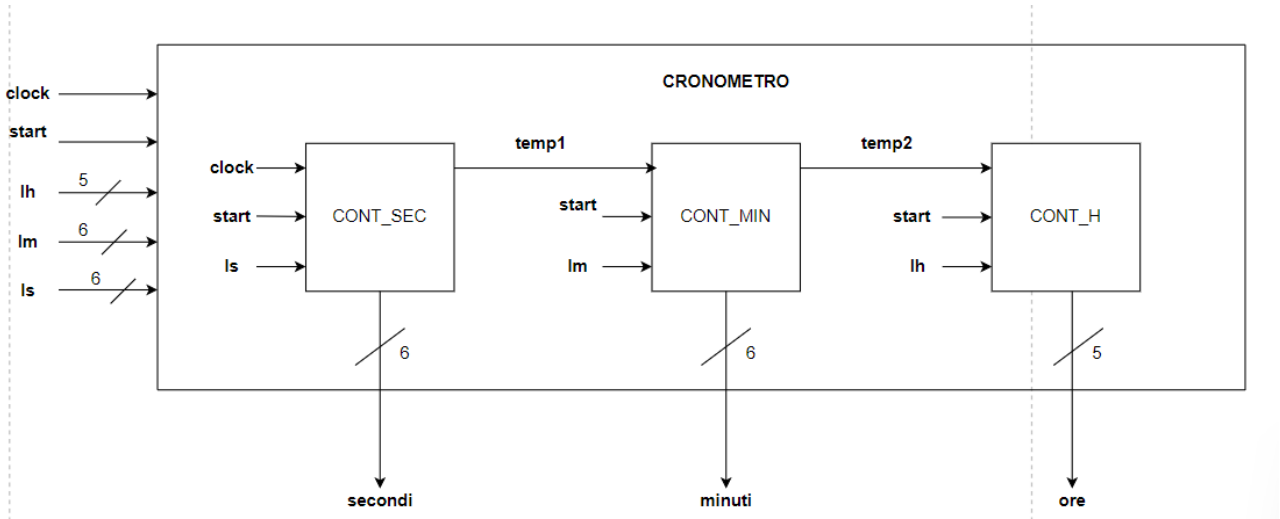
Cronometro

5.1 Traccia 1

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 Soluzione 1

Per progettare il cronometro abbiamo realizzato 3 contatori, il primo conta i secondi, da 0 a 59, il secondo i minuti, da 0 a 59, e il terzo le ore, da 0 a 23. Si è deciso quindi di utilizzare un contatore modulo 60, implementato in maniera Behavioral, a partire da un modulo 64, per realizzare il conteggio dei secondi e dei minuti. Invece, per realizzare il conteggio delle ore è stato realizzato un contatore modulo 24, a partire da un modulo 32. Il componente cronometro viene realizzato usando un approccio strutturale, collegando opportunamente i contatori. In questa prima soluzione abbiamo omesso il clock filter necessario per far lavorare il contatore dei secondi alla frequenza di 1Hz.



5.1.2 Codice 1

Per quanto riguarda il contatore dei minuti e dei secondi è stata realizzata un'architettura di tipo behavioral. Si usa il Generic inizializzando la stringa a "111011", ovvero a 59, in ingresso il blocco riceve il clock, che funge da ingresso di conteggio, il segnale di start, che abilita il contatore al suo funzionamento, il segnale di reset, di load, che permette di caricare il dato e una stringa binaria di 6 bit che contiene il dato stesso. In uscita, oltre alla stringa di 6 bit, viene prodotto un altro segnale che decreta la fine del conteggio. Nell'architettura viene definito un segnale TY, di 6 bit, che va in ingresso al contatore successivo. Viene poi definito un process in cui si effettuano i vari controlli. In particolare, quando il reset è alto, l'uscita TY è pari a zero e il segnale CLK_out resta alto, questo vuol dire che il conteggio non è finito. Nel caso in cui il dato caricato è maggiore del valore massimo, al dato viene assegnato proprio il valore più alto. Infine nell'ultima if viene fatto l'incremento del conteggio. Di seguito è rappresentato il codice.

```
entity contatore_beha is
    generic(M: std_logic_vector(0 to 5) := "111011");
    port(
        CLK_in: in std_logic;
        RESET: in std_logic;
        LOAD, START: in std_logic;
        DATA: in std_logic_vector(0 to 5);
        Y: out std_logic_vector (0 to 5);
        CLK_out: out std_logic
    );

end contatore_beha;

architecture behavioural of contatore_beha is
    signal TY: std_logic_vector(0 to 5);

begin

count: process(CLK_in, RESET, LOAD)
    begin
    if(RESET='1') then
        TY<=(others=>'0');
        CLK_out<='1';
    elsif(LOAD='1') then
        CLK_out<='1';
        if(unsigned(DATA) > unsigned(M)) then
            TY<=M;
        else
            TY<=DATA;
        end if;
    elsif(CLK_in'event and CLK_in = '0' and START='1') then
        CLK_out<='1';
        if(TY=M) then
            CLK_out<='0';
            TY<=(others=>'0');
         else
            TY<= std_logic_vector(unsigned(TY)+1);
         end if;
    end if;

    end process;

    Y<=TY;

end behavioural;
```

Per quanto riguarda il contatore delle ore il ragionamento è lo stesso, l'unica differenza, in questo caso, è che la stringa in ingresso e uscita è su 5 bit.

```
entity cont_h is
    generic(M: std_logic_vector(0 to 4) := "10111");
    port(
        CLK_in: in std_logic;
        RESET: in std_logic;
        LOAD,START: in std_logic;
        DATA: in std_logic_vector(0 to 4);
        Y: out std_logic_vector (0 to 4);
        CLK_out: out std_logic
    );

end cont_h;

architecture behavioural of cont_h is
    signal TY: std_logic_vector(0 to 4);

begin

count: process(CLK_in, RESET, LOAD)
    begin
    if(RESET='1') then
        TY<=(others=>'0');
        CLK_out<='1';
    elsif(LOAD='1') then
        CLK_out<='1';
        if(unsigned(DATA) > unsigned(M)) then
            TY<=M;
        else
            TY<=DATA;
        end if;
    elsif(CLK_in'event and CLK_in = '0' and START='1') then
        CLK_out<='1';
        if(TY=M) then
            CLK_out<='0';
            TY<=(others=>'0');
         else
            TY<= std_logic_vector(unsigned(TY)+1);
         end if;
    end if;

    end process;

    Y<=TY;

end behavioural;
```

Infine, realizziamo il componente cronometro utilizzando un approccio strutturale, in cui si collegano i vari contatori.

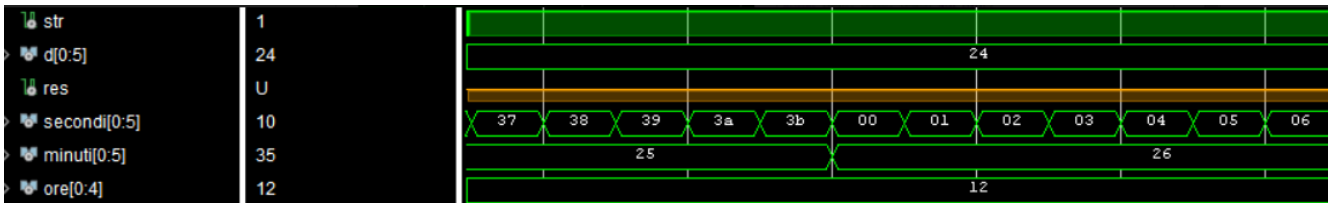
```
entity Cronometro is
    Port (
        clk: in std_logic;
        ls, lm, lh: in std_logic; --load
        str: in std_logic; --start
        d: in std_logic_vector(0 to 5); --data
        res: in std_logic;
        secondi, minuti: inout std_logic_vector(0 to 5);
        ore: inout std_logic_vector(0 to 4)
     );
end Cronometro;

architecture structural of Cronometro is

component contatore_beha is
    generic(M: std_logic_vector(0 to 5) := "111011");
    port(
        CLK_in: in std_logic;
        RESET: in std_logic;
        LOAD, START: in std_logic;
        DATA: in std_logic_vector(0 to 5);
        Y: out std_logic_vector (0 to 5);
        CLK_out: out std_logic
    );
end component;

component cont_h is
    generic(M: std_logic_vector(0 to 4) := "10111");
    port(
        CLK_in: in std_logic;
        RESET: in std_logic;
        LOAD,START: in std_logic;
        DATA: in std_logic_vector(0 to 4);
        Y: out std_logic_vector (0 to 4);
        CLK_out: out std_logic
    );
end component;

component clock_filter is
     generic(
            CLKIN_freq : integer := 100000000; --clock board 100MHz
            CLKOUT_freq : integer := 500       --frequenza desiderata 500Hz
                );
    Port (
           clock_in : in  STD_LOGIC;
           reset : in STD_LOGIC;
           clock_out : out  STD_LOGIC -- attenzione: non è un vero clock ma un
    );
end component;


signal temp1, temp2, temp3: std_logic;
signal dati: std_logic_vector(16 downto 0);
signal useless: std_logic_vector(16 downto 0);
signal clock_out: std_logic;

begin
--filtro ad 1Hz
cf: clock_filter generic map(100000000, 100000) port map(clk, res, clock_out);
s: contatore_beha port map(clock_out, res, ls, str, d, secondi, temp1);
m: contatore_beha port map(temp1, res, lm, str, d, minuti, temp2);
h: cont_h port map(temp2, res, lh, str, d(0 to 4), ore, temp3);

dati<=secondi&minuti&ore;

end structural;
```

Nota: il secondo generic del clock_filter viene impostato ad 1 (uscita ad 1Hz) quando bisogna sintetizzare il cronometro su board.

5.1.3 Simulazione 1

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica. Sono stati considerati vari casi di test. Nell'immagine sottostante viene mostrato il TestBench del cronometro con il rispettivo risultato della simulazione.

```
stimulus: process
begin

  wait for 20ns;
  d<="100100";
  ls<='1';
  lm<='1';
  lh<='1';
  wait for 10 ns;
  ls<='0';
  lm<='0';
  lh<='0';
  wait for 10 ns;
  str<='1';

  wait;
end process;
```

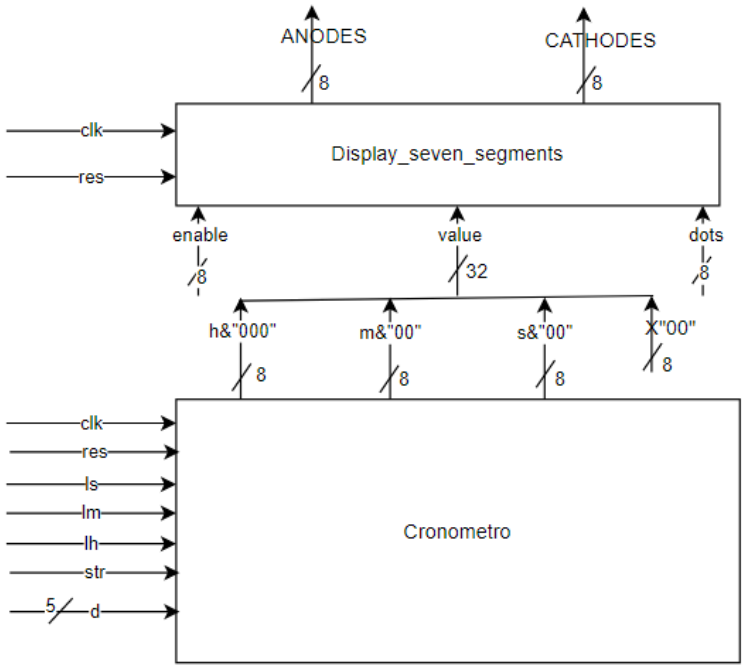


5.2 Traccia 2

Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando i display a 7 segmenti per la visualizzazione dell'orario (o una combinazione di display e led nel caso in cui i display a disposizione siano in numero inferiore a quello necessario), gli switch per l'immissione dell'orario iniziale e due bottoni, uno per il set dell'orario e uno per il reset. Si utilizzi una codifica a scelta dello studente per la visualizzazione dell'orario sul display (esadecimale o decimale).

5.2.1 Sintesi su board

Per poter visualizzare il cronometro sul display della board è stato necessario utilizzare un componente chiamato display_seven_segments, il cui funzionamento è stato dettagliatamente spiegato a lezione. Le due uscite di questo componente, ANODES e CATHODES, sono quelle che andremo a mappare nel file di constraint nella sezione del display.



Riportiamo l'architettura strutturale.

```vhdl
entity Cronometro is
    Port (
        clk: in std_logic;
        ls, lm, lh: in std_logic; --load
        str: in std_logic; --start
        d: in std_logic_vector(0 to 5); --data
        res: in std_logic;
        ANODES : out  STD_LOGIC_VECTOR (7 downto 0);
        CATHODES : out  STD_LOGIC_VECTOR (7 downto 0)
     );
end Cronometro;

architecture structural of Cronometro is
```


```vhdl
entity Cronometro is
    Port (
        clk: in std_logic;
        ls, lm, lh: in std_logic; --load
        str: in std_logic; --start
        d: in std_logic_vector(0 to 5); --data
        res: in std_logic;
        ANODES : out  STD_LOGIC_VECTOR (7 downto 0);
        CATHODES : out  STD_LOGIC_VECTOR (7 downto 0)
     );
end Cronometro;

architecture structural of Cronometro is
```

```
component display_seven_segments is
    Generic(
                CLKIN_freq : integer := 100000000;
                CLKOUT_freq : integer := 500
                );
    Port ( CLK : in  STD_LOGIC;
           RST : in  STD_LOGIC;
           VALUE : in  STD_LOGIC_VECTOR (31 downto 0);
           ENABLE : in  STD_LOGIC_VECTOR (7 downto 0); -- decide quali cifre abilitare
           DOTS : in  STD_LOGIC_VECTOR (7 downto 0); -- decide quali punti visualizzare
           ANODES : out  STD_LOGIC_VECTOR (7 downto 0);
           CATHODES : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

signal templ, temp2, temp3: std_logic;
signal useless: std_logic_vector(16 downto 0);
signal clock_out: std_logic;
signal value: std_logic_vector(31 downto 0);
signal secondi,minuti: std_logic_vector(0 to 5);
signal ore: std_logic_vector(0 to 4);

begin
dss: display_seven_segments port map(clk, res,value,"11111100","00000000",ANODES, CATHODES);
--filtro ad 1Hz
cf: clock_filter generic map(100000000, 1) port map(clk, res, clock_out);
s: contatore_beha port map(clock_out, res, ls, str, d, secondi, templ);
m: contatore_beha port map(templ, res, lm, str, d, minuti, temp2);
h: cont_h port map(temp2, res, lh, str, d(0 to 4), ore, temp3);
--valore da mandare al display seven segments
value<="000"&ore&"00"&minuti&"00"&secondi&X"00";

end structural;
```

Nel display_seven_segments abbiamo che value<=”000”&ore&”00”&minuti&”00”&secondi&X”00”, enable<=”11111100” e dots<=”00000000”. Questo perché sono necessari solo sei display per visualizzare il cronometro, mentre non è necessario alcun punto.

Riportiamo il file di constraint utilizzato per mappare gli ingressi e le uscite.

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


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { str }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
#set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
#set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { SW[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
#set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
#set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { SW[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { SW[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
#set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { SW[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 { d[5] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports { d[4] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { d[3] }]; #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports { d[2] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { d[1] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { d[0] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]
```

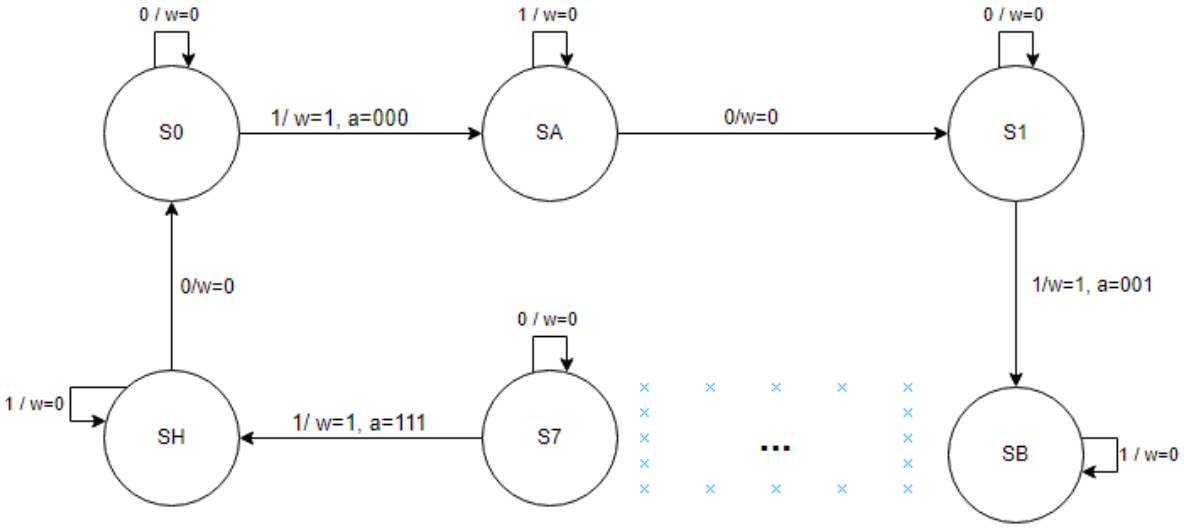
```
##7 segment display
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { CATHODES[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { CATHODES[1] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { CATHODES[2] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { CATHODES[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { CATHODES[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { CATHODES[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { CATHODES[6] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { CATHODES[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { ANODES[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { ANODES[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { ANODES[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { ANODES[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { ANODES[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { ANODES[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { ANODES[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { ANODES[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons
#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { str }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { ls }]; #IO_L4N_T0_D05_14 Sch=btnu
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { lm }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { res }]; #IO_L10N_T1_D15_14 Sch=btnr
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { lh }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

5.3 Traccia 3

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*.

5.3.1 Soluzione 3

Per arrivare a tale soluzione abbiamo pensato di implementare un automa in cui agiscono un segnale di ingresso, un segnale di write, che abilita la scrittura, e un segnale address per indicare l'indirizzo. In particolare, quando siamo nello stato iniziale, permaniamo in esso fin quando non si alza il segnale di ingresso. A fronte di tale segnale viene alzato il segnale di write e al segnale address viene assegnato il valore '000'. Si passa così allo stato successivo nel quale il segnale di write viene abbassato e si permane in tale stato finché l'ingresso è pari ad 1, nel caso contrario l'automa passa allo stato successivo e così via.



5.3.2 Codice 3

Abbiamo definito dapprima l'entity con i necessari ingressi e uscite. Successivamente, nell'architettura, sono stati dichiarati gli stati che vanno a comporre l'automa, indicando come stato iniziale lo stato S0. Nel process abbiamo definito il percorso dell'automa, indicando come vengono effettuati i vari passaggi da uno stato all'altro.

```
entity Gestore_intertempo is
    Port (
        clock : in std_logic;
        ingresso: in std_logic;
        w: out std_logic; --write
        a: out std_logic_vector(2 downto 0) --address
     );
end Gestore_intertempo;

architecture Behavioral of Gestore_intertempo is

type stato is (S0, SA, S1, SB, S2, SC, S3, SD, S4, SE, S5, SF, S6, SG, S7, SH);
signal curr_state, next_state: stato := S0;

begin

reg: process(clock)
    begin
    if (clock'event and clock='0') then
        curr_state<=next_state;
    end if;
    end process;

comb: process(ingresso, curr_state)
    begin
    case curr_state is
        when S0 => if(ingresso='1') then
                   w<='1';
                   a<="000";
                   next_state<=SA;
                   end if;
        when SA =>  w<='0';
                   if(ingresso='0') then
                   next_state<=S1;
                   end if;
        when S1 => if(ingresso='1') then
                   w<='1';
                   a<="001";
                   next_state<=SB;
                   end if;
```

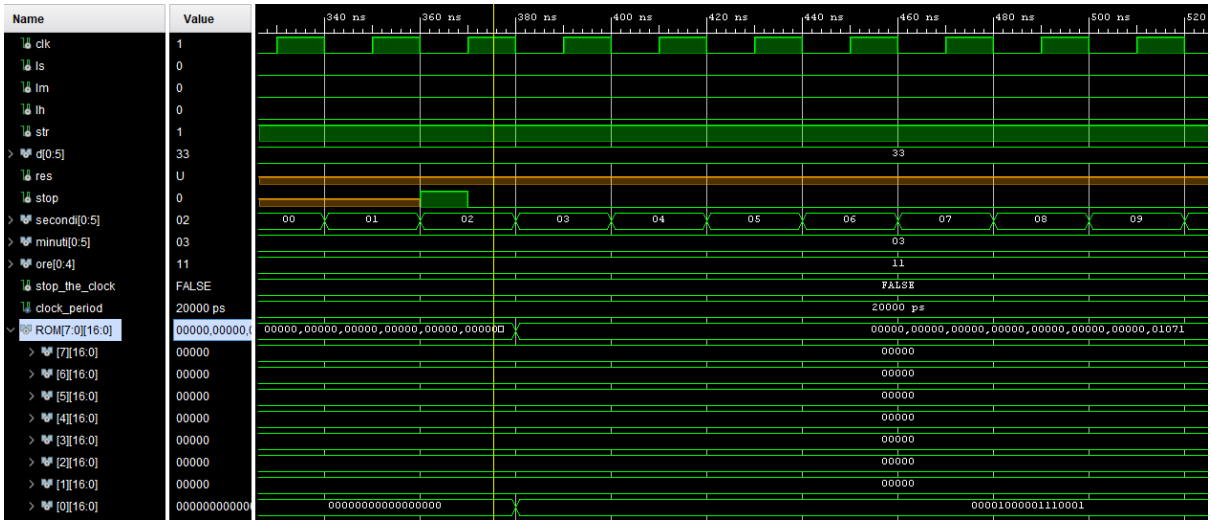
```
        when S6 => if(ingresso='1') then
                    w<='1';
                    a<="110";
                    next_state<=SG;
                    end if;
        when SG =>  w<='0';
                    if(ingresso='0') then
                    next_state<=S7;
                    end if;
        when S7 => if(ingresso='1') then
                    w<='1';
                    a<="111";
                    next_state<=SH;
                    end if;
        when SH =>  w<='0';
                    if(ingresso='0') then
                    next_state<=S0;
                    end if;
    end case;
    end process;

end Behavioral;
```

Nota: sarebbe stato possibile semplificare molto l'automa utilizzando un contatore.

5.3.3 Simulazione 3

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica. Sono stati considerati vari casi di test, mostrati nell'immagine sottostante.

```
stimulus: process
begin

wait for clock_period*5;

d<="000010";
lm<='1';
wait for 20 ns;
d<="100011";
lm<='0';
lh<='1';
wait for 20 ns;
d<="110011";
lh<='0';
ls<='1';
wait for 20 ns;
ls<='0';
str<='1';
wait for 200ns;
stop<='1';
wait for 10 ns;
stop<='0';
wait for 300ns;
str<='0';

  wait;
end process;
```



Capitolo 6

Testing Automatico

6.1 Traccia 1

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 memorizzati in una memoria interna, che deve poter essere svuotata in qualsiasi momento in presenza di un segnale di reset.

6.1.1 Soluzione e codice

Per progettare un sistema in grado di testare automaticamente una macchina combinatoria abbiamo prima di tutto realizzato quattro componenti di base, una ROM, la macchina da testare, una memoria ed un contatore. La ROM è inizializzata con i valori di ingresso con i quali si vuole testare la macchina combinatoria.

```
entity ROM is
port(
    CLK : in std_logic; -- clock della board
    RST : in std_logic;
    READ : in std_logic; -- segnale che abilita la lettura, inserito
    ADDR : in std_logic_vector(2 downto 0); --3 bit di indirizzo per
                                            --sono inseriti tramite g
    DATA : out std_logic_vector(3 downto 0) -- dato su 8 bit letto da
    );
end ROM;
-- creo una ROM di 8 elementi da 8 bit ciascuno
architecture behavioral of ROM is
type rom_type is array (7 downto 0) of std_logic_vector(3 downto 0);
signal ROM : rom_type := (
"0110",
"1101",
"0100",
"0000",
"0001",
"1000",
"0100",
"1010");

attribute rom_style : string;
attribute rom_style of ROM : signal is "block";-- block dice al tool
                                               -- distributed di usar
begin

process(CLK)
  begin
    if rising_edge(CLK) then
        if (RST = '1') then
            DATA <= ROM(conv_integer("000"));
        elsif (READ = '1') then
            DATA <= ROM(conv_integer(ADDR));
        end if;
    end if;
end process;
end behavioral;
```

Nella memoria, ad ogni colpo di clock, se il segnale di write è alto, viene salvato il valore dell'uscita della macchina combinatoria all'interno della locazione indicata da address.

```
entity MEM is
port(
    CLK : in std_logic; -- clock della board
    RST : in std_logic;
    WRITE,READ : in std_logic; -- segnale che abilita la lettura, inseri
    ADDR : in std_logic_vector(2 downto 0); --3 bit di indirizzo per acc
                                            --sono inseriti tramite gli
    DATA : in std_logic_vector(2 downto 0); -- dato su 3 bit letto dalla
    Y: out std_logic_vector(2 downto 0)
    );
end MEM;
-- creo una ROM di 8 elementi da 3 bit ciascuno
architecture behavioral of MEM is
type rom_type is array (7 downto 0) of std_logic_vector(2 downto 0);
signal ROM : rom_type := (
"000",
"000",
"000",
"000",
"000",
"000",
"000",
"000" );

attribute rom_style : string;
attribute rom_style of ROM : signal is "block";-- block dice al tool di

begin

process(CLK)
  begin
    if (CLK'event and CLK='1') then
        if (RST = '1') then
            ROM(0)<="000";
            ROM(1)<="000";
            ROM(2)<="000";
            ROM(3)<="000";
            ROM(4)<="000";
            ROM(5)<="000";
            ROM(6)<="000";
            ROM(7)<="000";
            Y<=ROM(conv_integer("000"));
        elsif(READ='1') then
            Y<=ROM(conv_integer(ADDR));
        elsif (WRITE = '1') then
            ROM(conv_integer(ADDR))<= DATA;
        end if;
    end if;
end process;
end behavioral;
```

La macchina combinatoria è molto semplice, l'uscita dipende dai bit in ingresso secondo questa relazione in ordine di priorità:

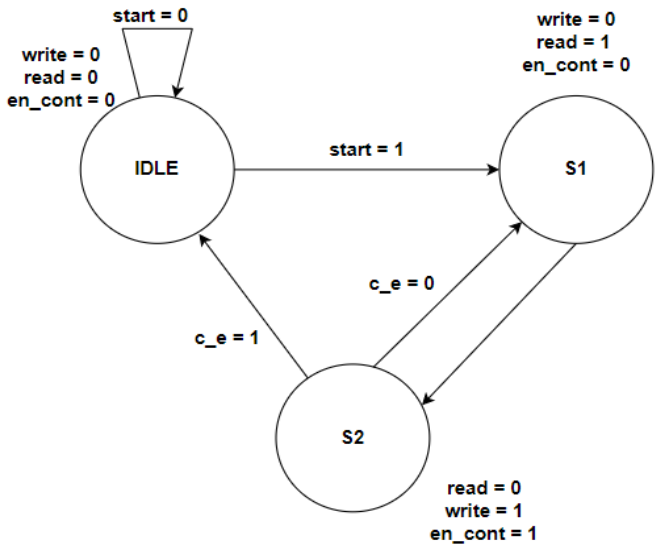
ingressi(3)=1 => uscite=111
ingressi(2)=1 => uscite=100
ingressi(1)=1 => uscite=010
ingressi(0)=1 => uscite=001
ingressi=0000=> uscite=000

```
entity macchina_comb is
    Port (
        ingressi: in std_logic_vector(3 downto 0);
        uscite: out std_logic_vector(2 downto 0)
    );
end macchina_comb;

architecture Behavioral of macchina_comb is

begin

process(ingressi)
begin
    if(ingressi(3)='1') then
        uscite<="111";
    elsif(ingressi(2)='1') then
        uscite<="100";
    elsif(ingressi(1)='1') then
        uscite<="010";
    elsif(ingressi(0)='1') then
        uscite<="001";
    elsif(true)   then
        uscite<="000";
end if;
end process;

end Behavioral;
```

E’ inoltre presente un contatore modulo 8, implementato comportamentalmente, necessario per inserire correttamente l’address nella ROM e nella memoria, oltre che per scandire la fine del test.

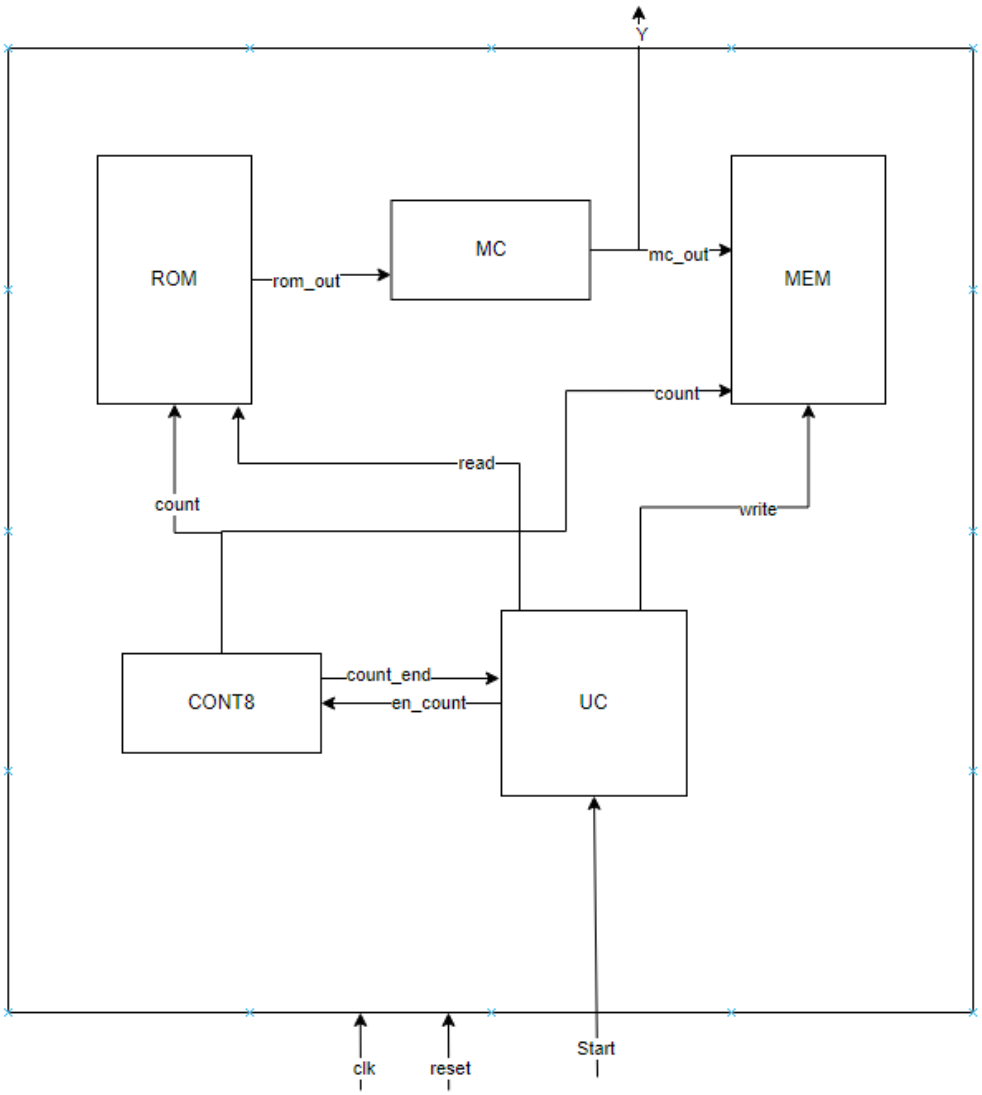
```vhdl
entity Cont8 is
    Port (
        clock, reset: in std_logic;
        count_in: in std_logic;
        count_end: out std_logic;
        count: out std_logic_vector(2 downto 0)
    );
end Cont8;

architecture Behavioral of Cont8 is

signal c: std_logic_vector(2 downto 0):="000";

begin

CM8: process(clock)
begin
    if(clock'event and clock='1') then
        if(reset='1') then
            c<=(others=>'0');
            count_end<='0';
        elsif(count_in='1') then
            c<=std_logic_vector(unsigned(c)+1);
            if(c="111") then    count_end<='1';
            else    count_end<='0';
            end if;
        end if;
    end if;
end process;
count <= c;
end Behavioral;
```

Questi 4 componenti sono gestiti attraverso un'unità di controllo definita dal seguente automa.



```
entity Control_unit is
    Port (
        clk,reset,start: in std_logic;
        count_end: in std_logic;
        write,read,en_count: out std_logic
     );
end Control_unit;

architecture automa of Control_unit is

type state is(idle, S1, S2);
signal current_state: state := idle;
signal next_state: state;

begin

reg:process(clk)
begin
if (clk'event and clk='0') then
    if(reset='1') then
        current_state<=idle;
    else
    current_state<=next_state;
    end if;
end if;
end process;

comb: process(start, count_end, current_state)
begin
    case current_state is
        when idle =>
                    read<='0';
                    write<='0';
                    en_count<='0';
                    if(start='1') then
                       next_state<=S1;
                    end if;
        when S1 =>  read<='1';
                    write<='0';
                    en_count<='0';
                    next_state<=S2;
        when S2 =>  en_count<='1';
                    read<='0';
                    write<='1';
                    if (count_end='0') then
                       next_state<=S1;
                    else
                       next_state<=idle;
                    end if;
    end case;
end process;

end automa;
```

Il top module assume la seguente struttura




```
entity TopModule is
    Port (
        clk_in,reset,start: in std_logic;
        Y: out std_logic_vector(2 downto 0)
     );
end TopModule;

architecture structural of TopModule is
```

```
signal count_end, write, read, en_count: std_logic;
signal count: std_logic_vector(2 downto 0);
signal rom_out: std_logic_vector(3 downto 0);
signal mc_out, useless: std_logic_vector(2 downto 0);
signal clk: std_logic;

begin
cu: Control_unit port map(clk, reset, start, count_end, write, read, en_count);
r: ROM port map(clk, reset, read, count, rom_out);
m: MEM port map(clk, reset, write, '0', count, mc_out, useless);
c: cont8 port map(clk, reset, en_count, count_end, count);
mc: macchina_comb port map(rom_out, mc_out);
cf: clock_filter generic map(100000000, 1) port map(clk_in, reset, clk);
Y<=mc_out;
end structural;
```

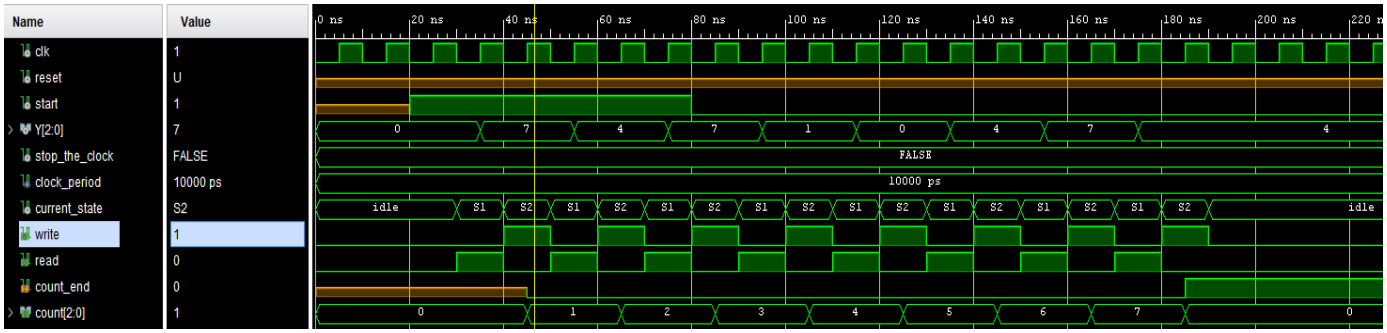
Nota: il clock filter serve solo per la sintesi su board.

6.1.2 Simulazione

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica. Gli otto valori precaricati nella ROM generano le seguenti uscite:

1010=>111, 0100=>100, 1000=>111, 0001=>001, 0000=>000, 0100=>100, 1101=>111, 0110=>100

```
  stimulus: process
  begin

  wait for 20ns;
  start<='1';
  wait for 60 ns;
  start<='0';

    wait;
  end process;
  clocking: process
  begin
    while not stop_the_clock loop
      clk <= '0', '1' after clock_period / 2;
      wait for clock_period;
    end loop;
    wait;
  end process;

end;
```



6.2 Traccia 2

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 Sintesi su board

Per poter visualizzare i led abbiamo dovuto utilizzare un clock_filter ad 1Hz implementato in modo comportamentale.

```
entity clock_filter is
     generic(
            CLKIN_freq : integer := 100000000; --clock board 100MHz
            CLKOUT_freq : integer := 500       --frequenza desiderata 50(
                );
    Port (
           clock_in : in  STD_LOGIC;
           reset : in STD_LOGIC;
           clock_out : out  STD_LOGIC -- attenzione: non è un vero clock
    );
end clock_filter;

architecture Behavioral of clock_filter is

signal clockfx : std_logic := '0';

constant count_max_value : integer := CLKIN_freq/(CLKOUT_freq)-1;

begin

clock_out <= clockfx;

count_for_division: process(clock_in)

variable counter : integer range 0 to count_max_value := 0;
begin

    if rising_edge(clock_in) then
       if( reset = '1') then
        counter := 0;
        clockfx <= '0';
       else
        if counter = count_max_value then
            clockfx <=  '1';
            counter := 0;
        else
            clockfx <=  '0';
            counter := counter + 1;
        end if;
       end if;
    end if;
end process;
```

Abbiamo inoltre utilizzato il seguente file di constraint.

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


## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { Y[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { Y[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { Y[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]

##Buttons
#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { BTNC }]; #IO_L9P_T1_DQS_14 Sch=btnc
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { BTNU }]; #IO_L4N_T0_D05_14 Sch=btnu
#set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { BTNL }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { reset }]; #IO_L10N_T1_D15_14 Sch=btnr
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { start }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

Capitolo 7

Comunicazione con Handshaking

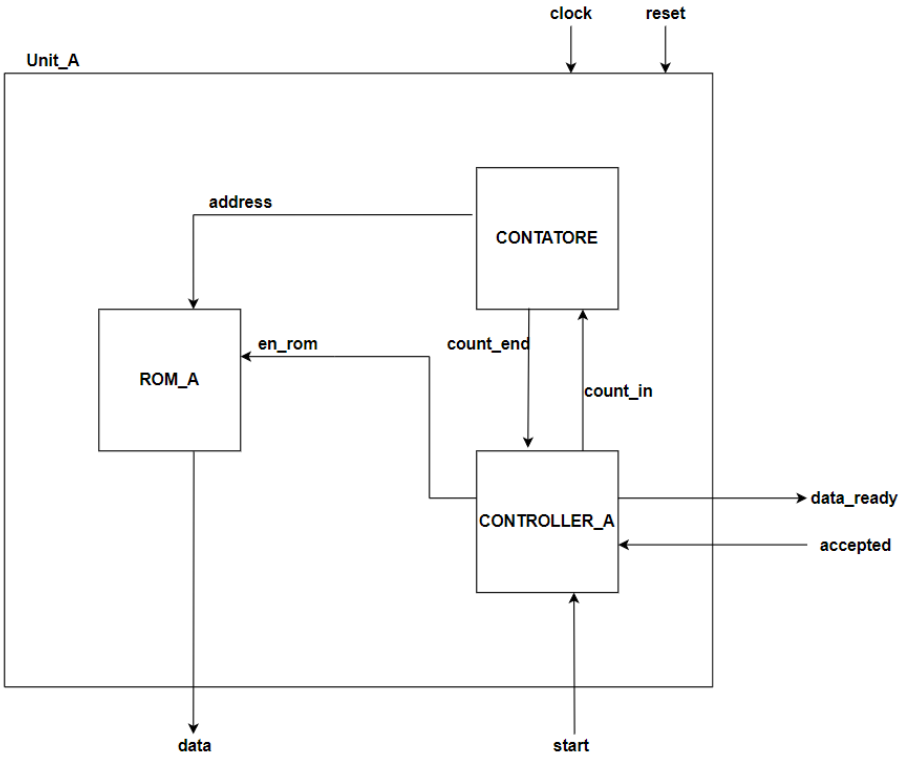
7.1 Traccia

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,..,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.

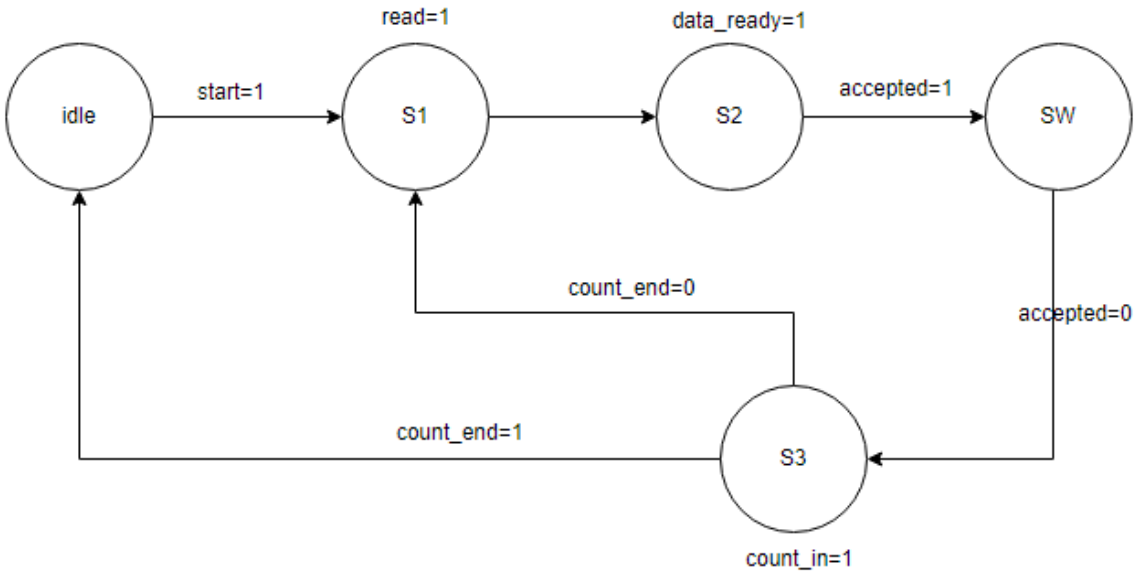
7.1.1 Soluzione

Il sistema A preleva il valore della ROM, il cui indirizzo proviene da un contatore mod8. Il dato di questa ROM arriva all'unità B, la quale lo somma con l'omologo della sua ROM. Il valore della somma viene poi salvato all'interno di un'altra memoria. I valori contenuti all'interno di questa memoria vengono poi mandati in uscita.

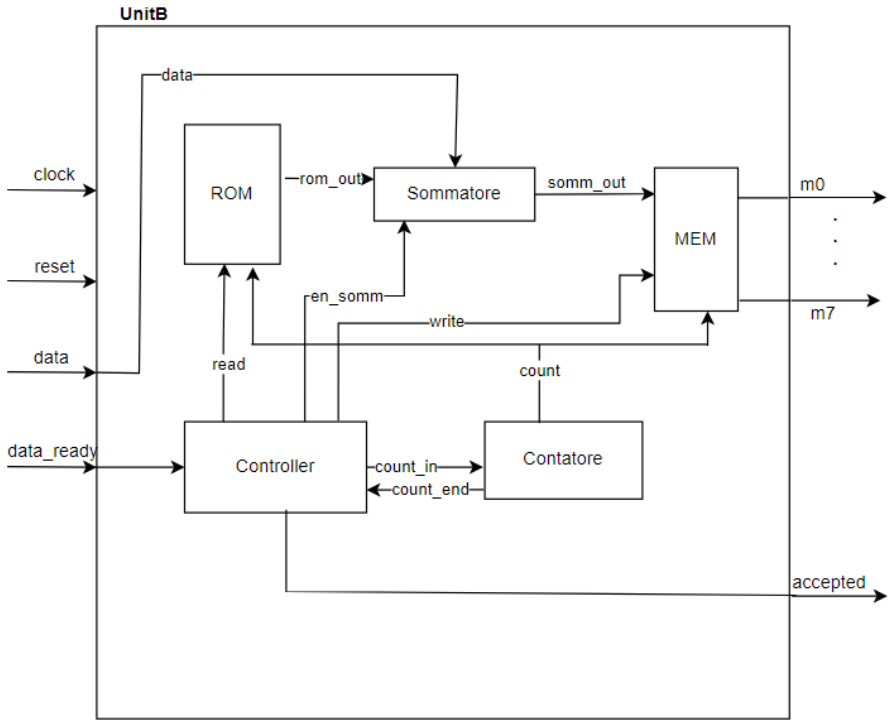
Per entrambi i nodi, l'intera architettura è stata decomposta in parte operativa e parte di controllo.



Il nodo A ha come componenti interni una ROM con valori precaricati, un contatore che utilizziamo per accedere all'indirizzo corretto della ROM e per decidere quando terminare la comunicazione con il nodo B. I due nodi sono gestiti dall'unità "CONTROLLER_A", realizzata mediante un automa a stati finiti mostrato in figura.

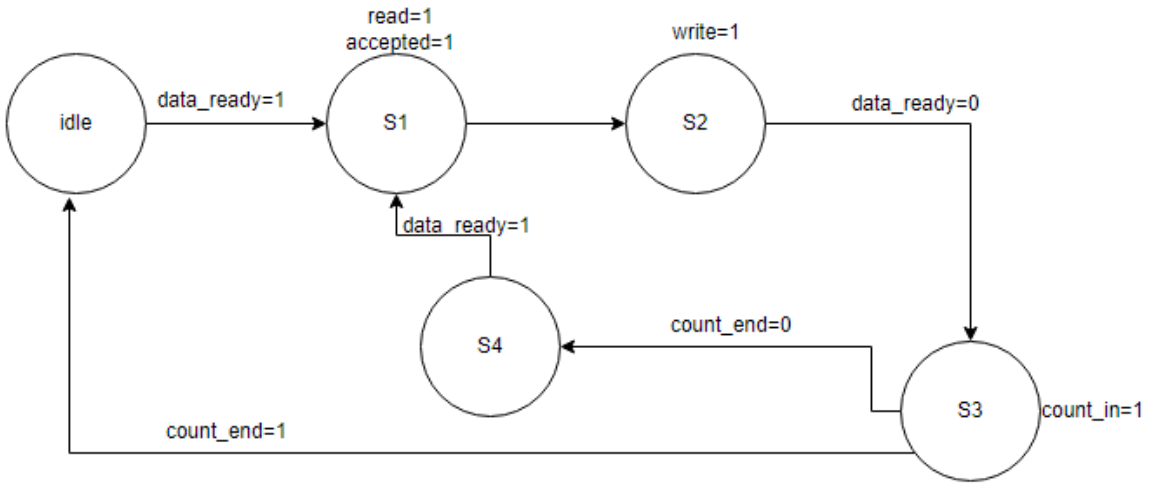


L'unità B, rispetto all'unità A, aggiunge un sommatore ed una memoria.

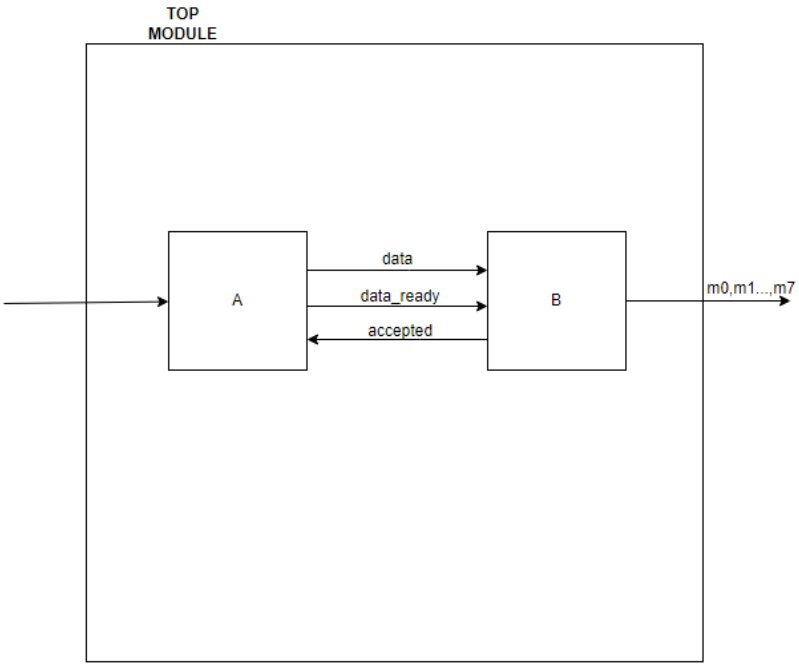


Nota: abbiamo successivamente rimosso en_somm.

Il controller dell'unità B, così come per l'unità A, è stato implementato mediante logica cablata, a partire dal seguente automa:



Le due unità, opportunamente collegate, vanno a formare il seguente Top Module:



I due sistemi comunicano mediante un protocollo di handshaking. L'unità A inserisce il valore della ROM nella linea "data", per poi alzare "data_ready". L'unità B può quindi alzare la linea "accepted" per comunicare di aver ricevuto correttamente i segnali. In corrispondenza a questa variazione di segnale, l'unità A abbassa "data_ready". Il segnale "accepted" verrà poi abbassato una volta completata l'elaborazione dei dati. Successivamente l'unità A potrà nuovamente inviare dati.

7.1.2 Codice

Abbiamo implementato il contatore del nodo A con un approccio Behavioral.

```vhdl
entity Cont8 is
    Port (
        clock, reset: in std_logic;
        count_in: in std_logic;
        count_end: out std_logic;
        count: out std_logic_vector(2 downto 0)
    );
end Cont8;

architecture Behavioral of Cont8 is

signal c: std_logic_vector(2 downto 0):="000";

begin

CM8: process(clock)
begin
    if(clock'event and clock='1') then
        if(reset='1') then
            c<=(others=>'0');
            count_end<='0';
        elsif(count_in='1') then
            c<=std_logic_vector(unsigned(c)+1);
            if(c="111") then    count_end<='1';
            else    count_end<='0';
            end if;
        end if;
    end if;
end process;
count <= c;
end Behavioral;
```

Per quanto riguarda la ROM, è stata utilizzata l'implementazione standard vista a lezione.

Modificando leggermente l'implementazione della ROM, abbiamo definito la memoria.

```
entity MEM is
port(
    CLK : in std_logic; -- clock della board
    RST : in std_logic;
    WRITE,READ : in std_logic; -- segnale che abilita la
    ADDR : in std_logic_vector(2 downto 0); --3 bit di i
                                            --sono inser
    DATA : in std_logic_vector(11 downto 0); -- dato su
    Y: out std_logic_vector(11 downto 0)
    );
end MEM;
-- creo una ROM di 8 elementi da 6 bit ciascuno
architecture behavioral of MEM is
type rom_type is array (7 downto 0) of std_logic_vector(
signal ROM : rom_type := (
X"000",
X"000",
X"000",
X"000",
X"000",
X"000",
X"000",
X"000" );

attribute rom_style : string;
attribute rom_style of ROM : signal is "block";-- block
                                               -- distri
begin


   process(CLK)
     begin
       if (CLK'event and CLK='1') then
           if (RST = '1') then
               Y<=ROM(conv_integer("000"));
               ROM(0)<=X"000";
               ROM(1)<=X"000";
               ROM(2)<=X"000";
               ROM(3)<=X"000";
               ROM(4)<=X"000";
               ROM(5)<=X"000";
               ROM(6)<=X"000";
               ROM(7)<=X"000";
           elsif(READ='1') then
               Y<=ROM(conv_integer(ADDR));
           elsif (WRITE = '1') then
               ROM(conv_integer(ADDR))<= DATA;
           end if;
       end if;
   end process;
   end behavioral;
```

Il sommatore è stato implementato in modo comportamentale.

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Sommatore is
    Port (
        a,b: in std_logic_vector(7 downto 0);
        c: out std_logic_vector(11 downto 0)
     );
end Sommatore;

architecture Behavioral of Sommatore is

begin

c<=std_logic_vector(unsigned("0000"&a)+(unsigned("0000"&b)));

end Behavioral;
```

Per quanto riguarda i due controller, abbiamo implementato, tramite due process, gli automi visti in precedenza.

```
entity ControllerA is
    Port (
        clock, reset: in std_logic;
        start: in std_logic;
        accepted, count_end: in std_logic;
        count_in, read, data_ready: out std_logic
     );
end ControllerA;

architecture automa of ControllerA is

type state is (idle, S1, S2, SW, S3);
signal current_state: state:= idle;
signal next_state: state;

begin

reg:process(clock)
begin
if (clock'event and clock='0') then
    if(reset='1') then
        current_state<=idle;
    else
        current_state<=next_state;
    end if;
end if;
end process;
```

```
comb: process(start, accepted, count_end, current_state)
begin
    case current_state is
        when idle => count_in<='0';
                     read<='0';
                     data_ready<='0';
                     if(start='1') then
                        next_state<=S1;
                     end if;
        when S1 =>   read<='1';
                     count_in<='0';
                     next_state<=S2;
        when S2 =>   data_ready<='1';
                     if(accepted='1') then
                        next_state<=SW;
                     end if;
        when SW =>   data_ready<='0';
                     if(accepted='0') then
                        next_state<=S3;
                     end if;
        when S3 =>   count_in<='1';
                     if(count_end='1') then
                        next_state<=idle;
                     else
                        next_state<=S1;
                     end if;
    end case;
end process;

end automa;
```

```
entity ControllerB is
    Port (
        clock, reset: in std_logic;
        data_ready, count_end: in std_logic;
        accepted, count_in, read, write: out std_logic
     );
end ControllerB;

architecture automa of ControllerB is

type state is(idle, S1, S2, S3, S4);
signal current_state: state:=idle;
signal next_state: state;

begin

reg:process(clock)
begin
if (clock'event and clock='0') then
    if(reset='1') then
        current_state<=idle;
    else
    current_state<=next_state;
    end if;
end if;
end process;

comb: process(data_ready, count_end, current_state)
begin
    accepted<='0';
    read<='0';
    write<='0';
    count_in<='0';
    case current_state is
        when idle =>
                    if(data_ready='1') then
                       next_state<=S1;
                    end if;
        when S1 =>  read<='1';
                    accepted<='1';
                    next_state<=S2;
        when S2 =>  write<='1';
                    if(data_ready='0') then
                       next_state<=S3;
                    end if;
        when S3 =>  count_in<='1';
                    if(count_end='1') then
                       next_state<=idle;
                    else
                       next_state<=S4;
                    end if;
        when S4 =>  if(data_ready='1') then
                       next_state<=S1;
                    end if;
    end case;
end process;

end automa;
```

Utilizzando un approccio strutturale, abbiamo opportunamente definito le due unità.

```
entity UnitA is
    Port (
        clock, reset: in std_logic;
        start, accepted: in std_logic;
        data: out std_logic_vector(7 downto 0);
        data_ready: out std_logic
         );
end UnitA;

signal en_count, en_rom: std_logic;
signal address: std_logic_vector(2 downto 0);
signal count_end: std_logic;

begin
r: ROM port map(clock, reset, en_rom, address, data);
c: cont8 port map(clock, reset, en_count, count_end , address );
contr: controllerA port map(clock, reset, start, accepted, count_end, en_count, en_rom, data_ready);
end structural;
```

```
entity UnitB is
    Port (
        data: in std_logic_vector(7 downto 0);
        data_ready: in std_logic;
        clock, reset: in std_logic;
        accepted: out std_logic
     );
end UnitB;

signal count_end: std_logic;
signal count_in,read,write: std_logic;
signal count: std_logic_vector(2 downto 0);
signal useless: std_logic_vector(11 downto 0);
signal rom_out: std_logic_vector(7 downto 0);
signal somm_out: std_logic_vector(11 downto 0);

begin

cont: cont8 port map(clock, reset, count_in, count_end, count);
r: ROM_B port map(clock, reset, read, count, rom_out);
contr: ControllerB port map(clock, reset, data_ready, count_end, accepted, count_in, read, write);
s: Sommatore port map(data, rom_out, somm_out );
m: MEM port map(clock, reset, write, '0', count, somm_out, useless);

end structural;
```

Infine, abbiamo costruito il TopModule per collegare le due unità.

```
entity TopModule is
    Port (
        clock,reset,start: in std_logic
     );
end TopModule;

architecture structural of TopModule is

component unitA is
    Port (
        clock, reset: in std_logic;
        start, accepted: in std_logic;
        data: out std_logic_vector(7 downto 0);
        data_ready: out std_logic
         );
end component;

component UnitB is
    Port (
        data: in std_logic_vector(7 downto 0);
        data_ready: in std_logic;
        clock, reset: in std_logic;
        accepted: out std_logic
     );
end component;

signal accepted: std_logic;
signal data: std_logic_vector(7 downto 0);
signal data_ready: std_logic;

begin
A: unitA port map(clock, reset, start, accepted, data, data_ready);
B: unitB port map(data, data_ready, clock, reset, accepted);

end structural;
```

7.1.3 Simulazione

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica. I sedici valori precaricati nelle due ROM sono:

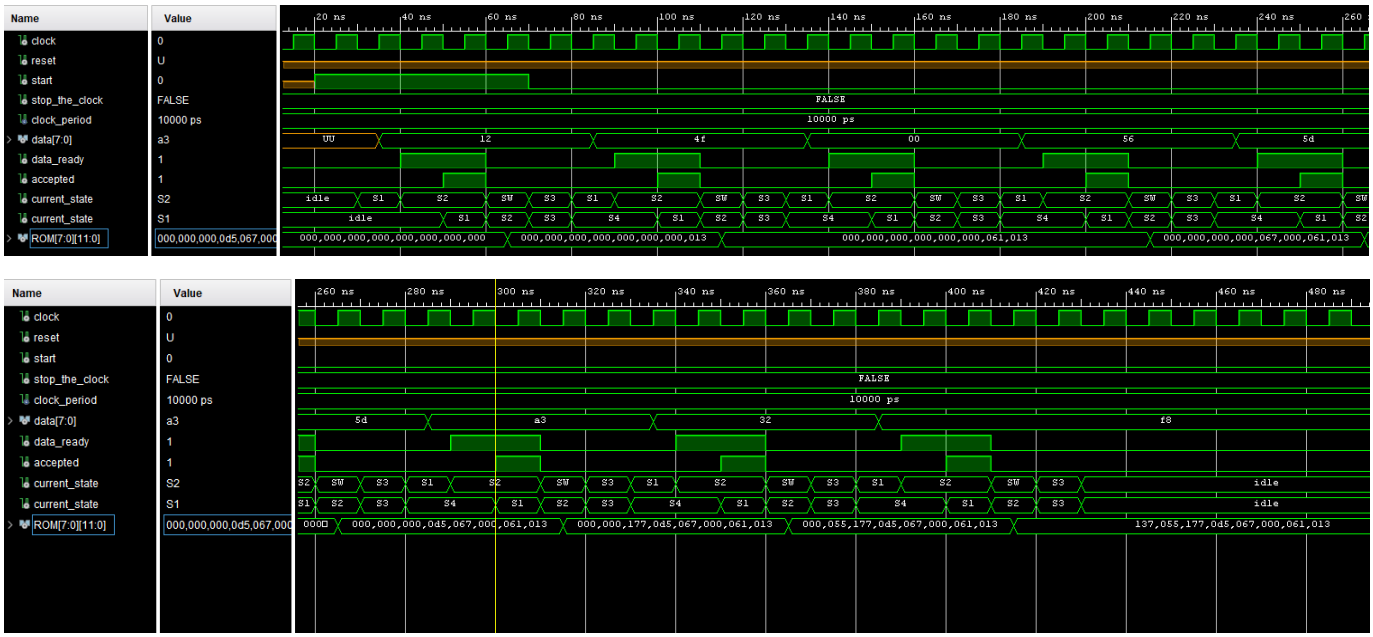
ROM_A:

X"F8", X"32", X"A3", X"5D", X"56",X"00",X"4F", X"12"

ROM_B:

X"3F", X"23", X"D4", X"78", X"11", X"00", X"12", X"01"

Alzando il segnale di start, la simulazione è la seguente:



Capitolo 8

Processore

8.1 Traccia

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

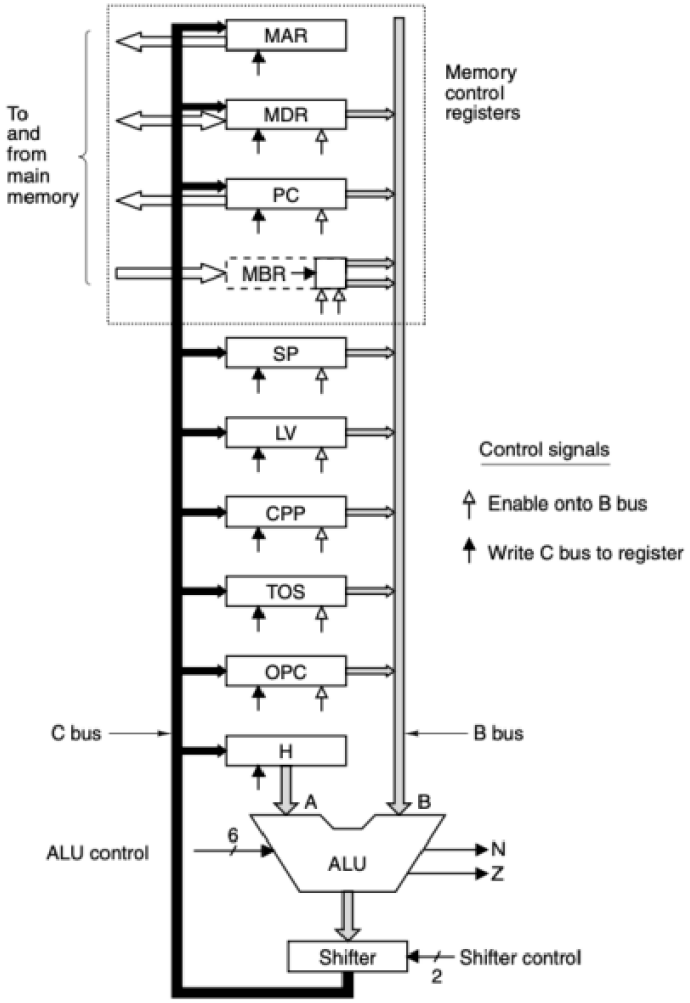
a) si proceda all'analisi dell'architettura mediante simulazione e si approfondisca lo studio del suo funzionamento per due istruzioni a scelta,
b) si modifichi un codice operativo a scelta, documentando tutte le modifiche effettuate,
c) (opzionale) si descriva il funzionamento del processore in merito alle istruzioni di input/output,
d) (solo ove possibile) si sintetizzi il processore su FPGA.

8.1.1 Introduzione

Il set di istruzioni implementato dal Mic-1 è un sottoinsieme di quello della Java Virtual Machine, denominato IJVM in quanto opera unicamente sugli interi. Tale processore presenta un'architettura a stack e una logica microprogrammata. Prevede due parti distinte che comunicano tra loro al fine di implementare le microistruzioni necessarie all'esecuzione di un'istruzione. Per poter lavorare con un'architettura a stack, come in questo caso, è necessario dover disporre dei giusti registri per poterla indirizzare, ma anche per tener traccia dell'operando in testa; nel caso del MIC-1, sono due i registri che assolvono a questo compito, lo "Stack Pointer (SP)", e il "Top of Stack (TOS)", che ci permettono di accedere alla testa dello stack, rispettivamente al valore e all'indirizzo.

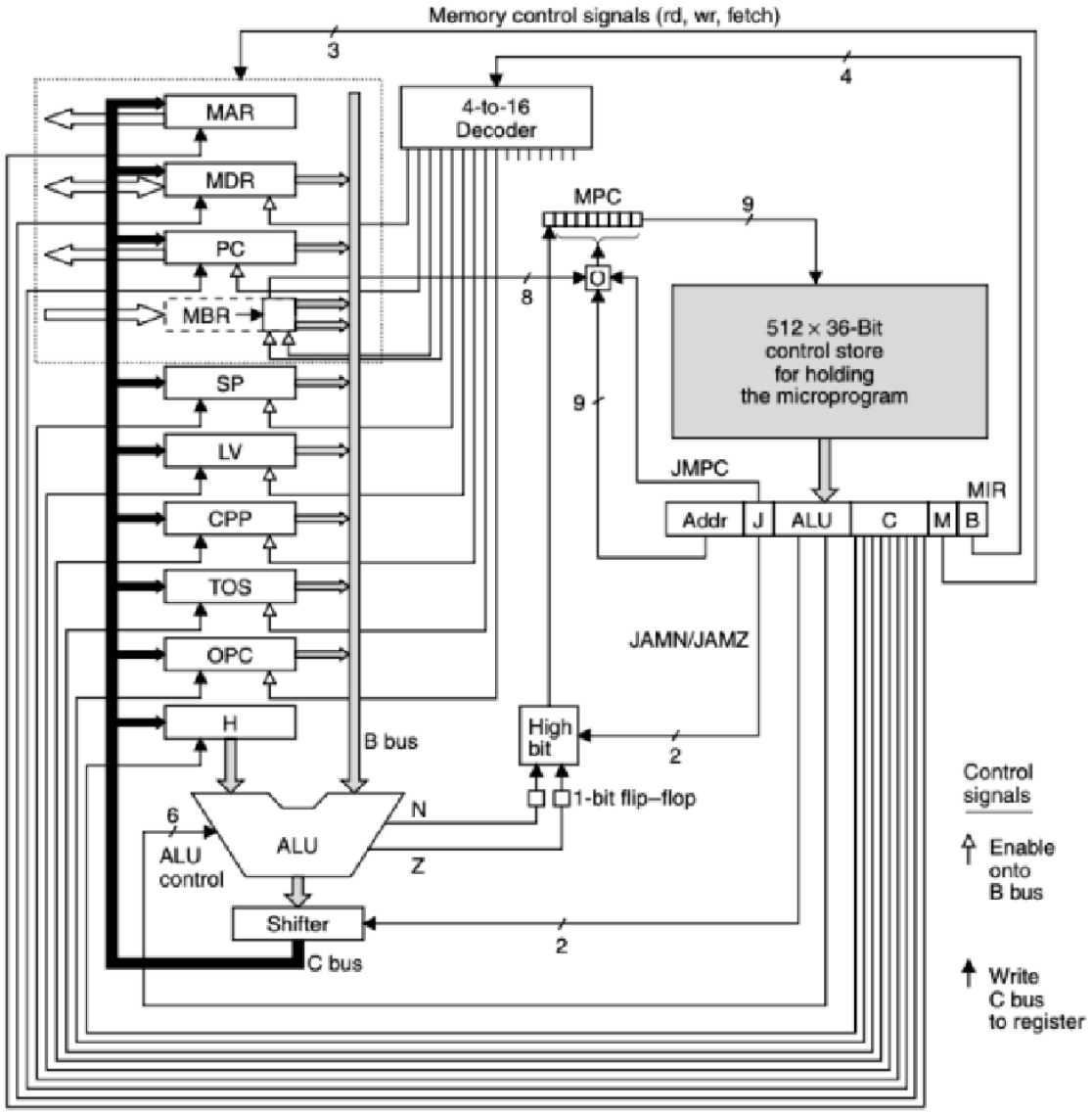
8.1.2 Datapath

Il Datapath del processore MIC-1 è costituito da 10 registri, due bus mediante i quali essi si scambiano i dati, una ALU e, al di sotto di essa, uno shift register. Per ogni registro sono presenti due abilitazioni, una per la lettura da un bus e un'altra per la scrittura da un altro bus. I due bus sono il Bus C che consente la lettura da parte di più registri contemporaneamente, e il Bus B il quale necessita che la scrittura sia effettuata in mutua esclusione.

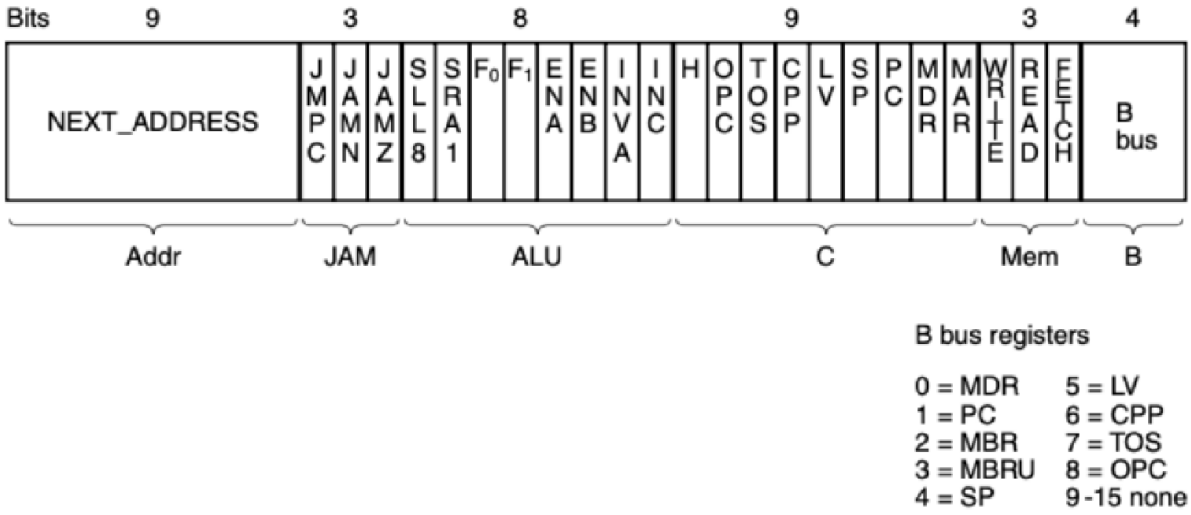


8.1.3 Unità di controllo

L'implementazione del processore Mic-1 che presentiamo è detta in logica microprogrammata: ciascuna istruzione IJVM _e implementata come una sequenza di microistruzioni (detta talvolta microprocedura); tali sequenze compongono il microprogramma.



L'unità di controllo deve generare la sequenza dei segnali di controllo necessari a pilotare l'unità operativa, ad ogni ciclo di clock. Nel caso del processore MIC-1 sono necessari 29 segnali, essi saranno prodotti dall'unità di controllo.



8.2 Approfondimento BIPUSH e IOR

Abbiamo considerato il seguente programma scritto in assembler:

```
.main
BIPUSH 0x7
BIPUSH 0xC
IOR
HALT
.endmethod
```

Le due istruzioni assembler corrispondono alle due seguenti microprocedure in linguaggio mal:

BIPUSH = 0x10:

```
SP = MAR = SP + 1
PC = PC + 1; fetch
MDR = TOS = MBR; wr; goto main
```

IOR = 0xB6

```
MAR = SP = SP-1; rd
H = TOS
MDR = TOS = MDR OR H; wr; goto main
```

8.2.1 BIPUSH

BIPUSH = 0x10:

```
SP = MAR = SP + 1
PC = PC + 1; fetch
MDR = TOS = MBR; wr; goto main
```

Analizziamo l'esecuzione della bipush tramite le microistruzioni mal correlate ad essa. Nella prima microistruzione il registro stack-pointer viene incrementato in modo tale da

predisporsi al salvataggio dell'elemento nello stack. Si predispone inoltre anche la scrittura del dato in memoria andando ad aggiornare il registro MAR.

- ALU: 110101 (incremento dell'operando B);
- C: 000001001 (scrittura su MAR e SP);
- MEM: 000 (nessuna operazione di memoria);
- B: 0100 (lettura da SP);

Al colpo di clock successivo verrà eseguita la seconda microistruzione. Il program counter è incrementato per poter leggere l'istruzione successiva contenente l'operando.

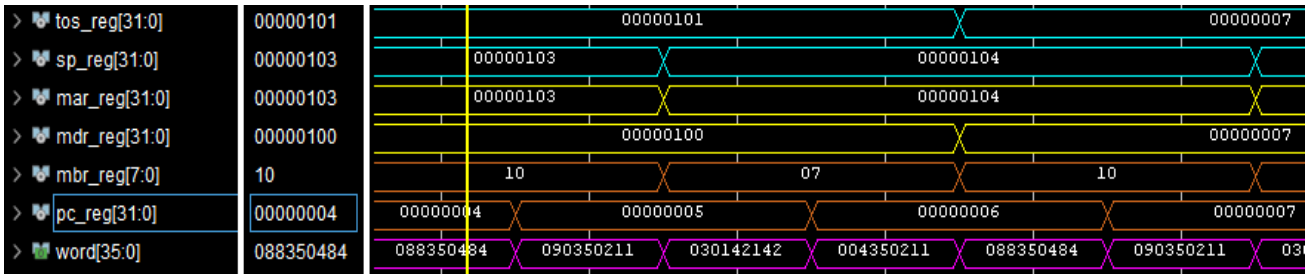
- ALU: 110101 (incremento dell'operando B);
- C: 000000100 (scrittura su PC);
- MEM: 001 (fetch);
- B: 0001 (lettura da PC);

L'operando è memorizzato nell'MBR, copiato nel TOS e nell'MDR, viene poi avviata la scrittura tramite la parola chiave wr.

- ALU: 010100 (l'operando B passa invariato);
- C: 001000010 (scrittura su TOS e MDR);
- MEM: 100 (write);
- B: 0010 (lettura da MBR);

Infine, goto main è la microistruzione con la quale termina ogni microprogramma. la sua funzione è incrementare il program counter, fare il fetch del successivo byte di istruzione e usare il contenuto del registro MBR come prossimo valore del microprogram counter.

Vediamo in simulazione come la control word (in viola) e i vari registri contengono ciò che abbiamo scritto in precedenza:



Notiamo come è presente il valore 7 prima in MBR e poi in TOS e MDR.

8.2.2 IOR

IOR = 0xB6

MAR = SP = SP-1; rd
H = TOS
MDR = TOS = MDR OR H; wr; goto main

Analizziamo in questo caso l'esecuzione dell'istruzione IOR. Con la prima microistruzione si effettua una read dalla memoria, questa microistruzione prevede di decrementare lo stack-pointer per poter leggere il secondo operando implicito nell'operazione (in quanto verrà tratto dalla posizione immediatamente inferiore alla testa dello stack). Ogni microistruzione è tradotta in una stringa di bit, possiamo quindi raggruppare le stringhe di bit e determinare quali sono i bit corrispondenti. In particolare, per la prima microistruzione avremo:

- Per la ALU i sei bit di controllo 110110 (decremento dell'operando B);
- Per il Bus C i nove bit di controllo 000001001 (scrittura su MAR e SP);
- Per le operazioni in memoria i tre bit di controllo 010 (read);
- Per i bit codificati del Bus B i quattro bit di controllo 0100 (lettura da SP).

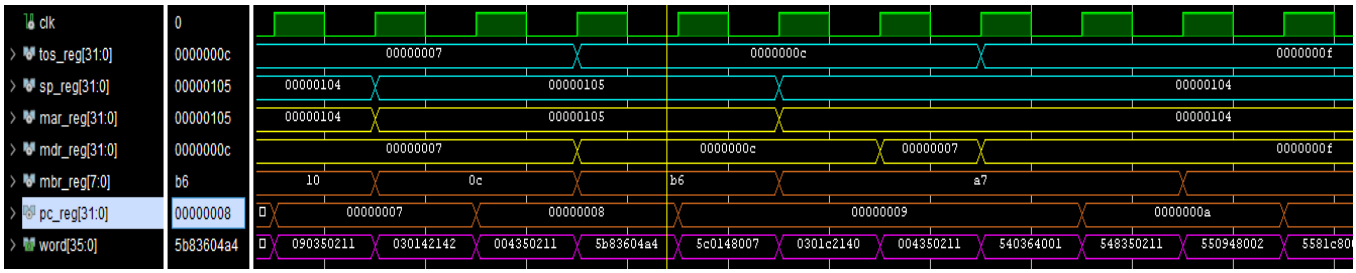
Al colpo di clock successivo si esegue la seconda microistruzione dove si memorizza il valore contenuto in TOS (primo operando) nel registro H, poichè TOS contiene il valore della precedente testa dello stack. In questo caso avremo:

- ALU: 010100;
- C: 100000000;
- MEM: 000;
- B: 0111

Nell'ultima microistruzione traiamo il secondo operando implicito, il quale è stato memorizzato nella precedente operazione di lettura in MDR, ne facciamo la OR tramite l'ALU con il primo operando salvato in H, sovrascriviamo l'attuale valore presente in TOS, e in MDR, e specifichiamo l'operazione di write per memorizzare il risultato, presente in MDR, nell'indirizzo puntato dal MAR.

- ALU: 011100;
- C: 001000010;
- MEM: 100 (write);
- B: 0001

Vediamo in simulazione come la control word (in viola) e i vari registri contengono ciò che abbiamo scritto in precedenza:



Notiamo come sia presente il risultato della OR (0x7 OR 0xC = 0xF) in MDR e TOS.

8.3 Modifica istruzione

La traduzione tra linguaggio assembler e linguaggio mal è effettuato utilizzando il file “avjm.mal”. Andando a modificare questo file è possibile associare ad ogni istruzione assembler un microprogramma diverso da quello predefinito. Nel nostro caso abbiamo modificato il microprogramma associato all’istruzione IOR in modo tale da effettuare una somma (IADD).

PRIMA:

IOR = 0xB6

MAR = SP = SP-1; rd
H = TOS
MDR = TOS = MDR OR H; wr; goto main

DOPO:

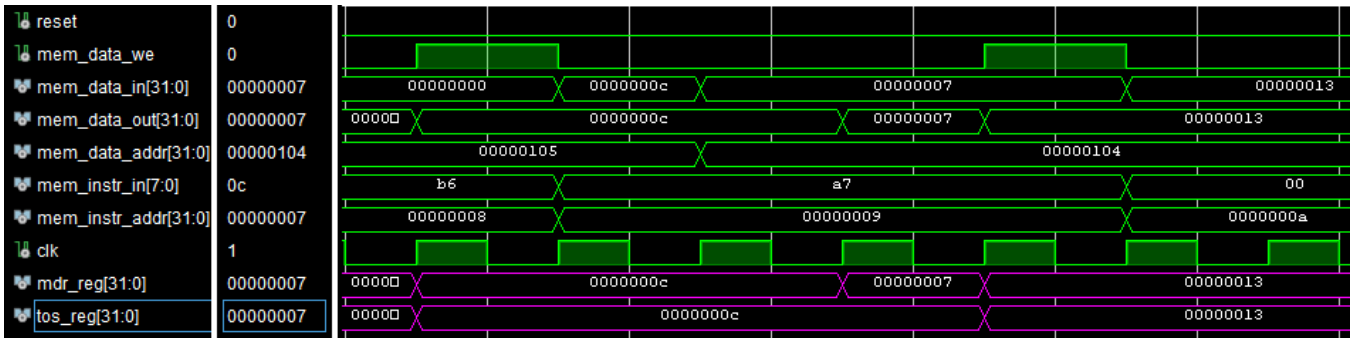
IOR = 0xB6

MAR = SP = SP-1; rd
H = TOS
MDR = TOS = MDR + H; wr; goto main

Come è possibile notare, è stata semplicemente modificata l’ultima microistruzione relativa all’operazione che deve effettuare l’ALU.

Considerato lo stesso programma eseguito precedentemente, il risultato aspettato sarà ora “0x13” invece di “0xf”.

```
.main
BIPUSH 0x7
BIPUSH 0xC
IOR
HALT
.endmethod
```



Notiamo come sia presente il risultato della somma (0x7 + 0xC = 0x13) in MDR e TOS.

Capitolo 9

UART

9.1 Traccia

Esercizio 9.1

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.

Esercizio 9.2

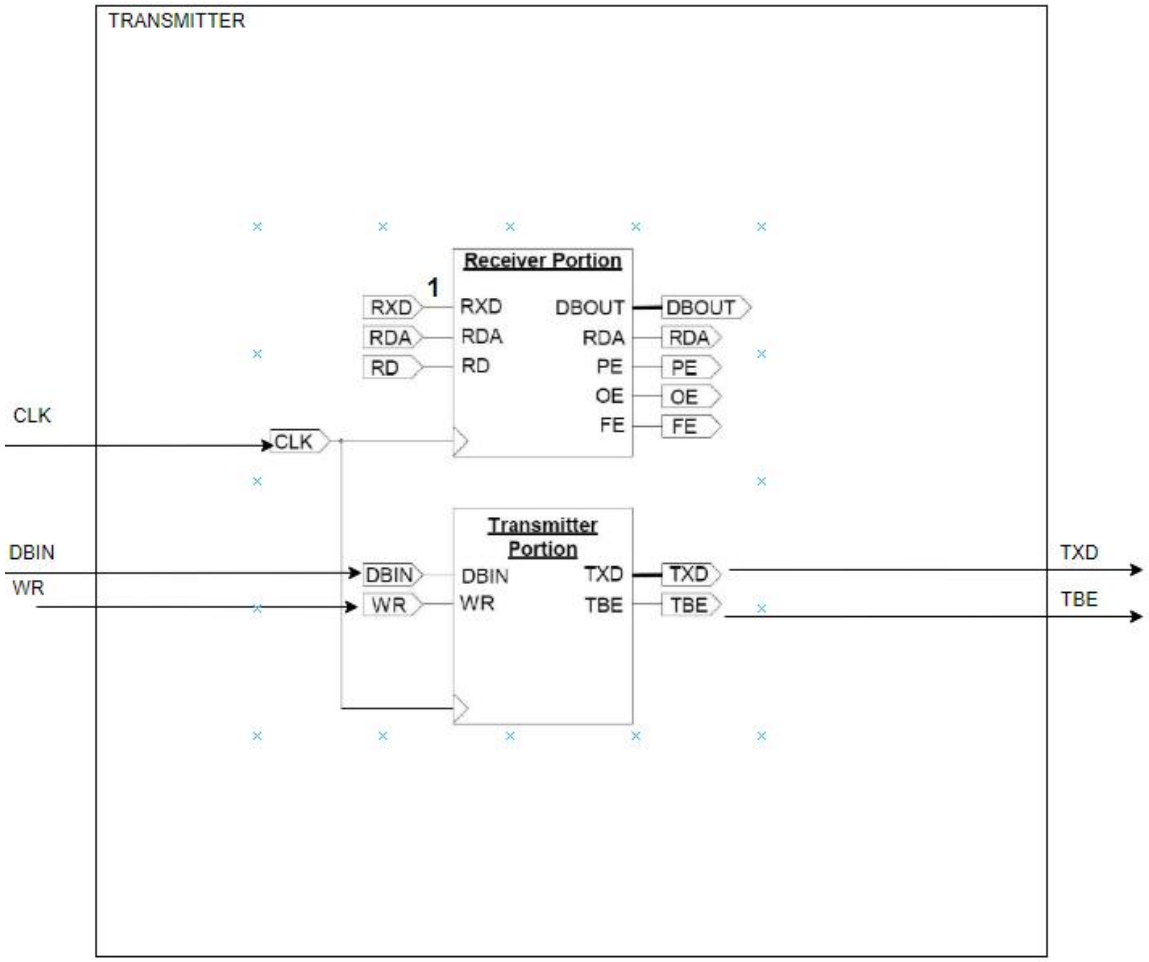
Implementare uno dei seguenti sistemi a scelta dello studente:

a) 2_UART_MEM: 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.
b) UART_PC: il sistema realizza la comunicazione fra un nodo A rappresentato da un componente sintetizzato su un FPGA e un nodo B rappresentato da un terminale seriale in esecuzione su PC (es. Termite), previa connessione di PC e board tramite dispositivo fisico RS232 (uno degli endpoint di comunicazione è rappresentato dal PC). Il componente A acquisisce una stringa di 8 bit che rappresenta un carattere in codifica ASCII fornita dall'utente mediante gli switch della board di sviluppo, e la invia mediante il dispositivo UART al terminale B in esecuzione sul PC, in cui il carattere viene visualizzato. Allo stesso modo, il componente deve essere in grado di ricevere attraverso lo stesso dispositivo UART (oppure una seconda UART) un carattere trasmesso dal terminale e mostrarlo sui led.

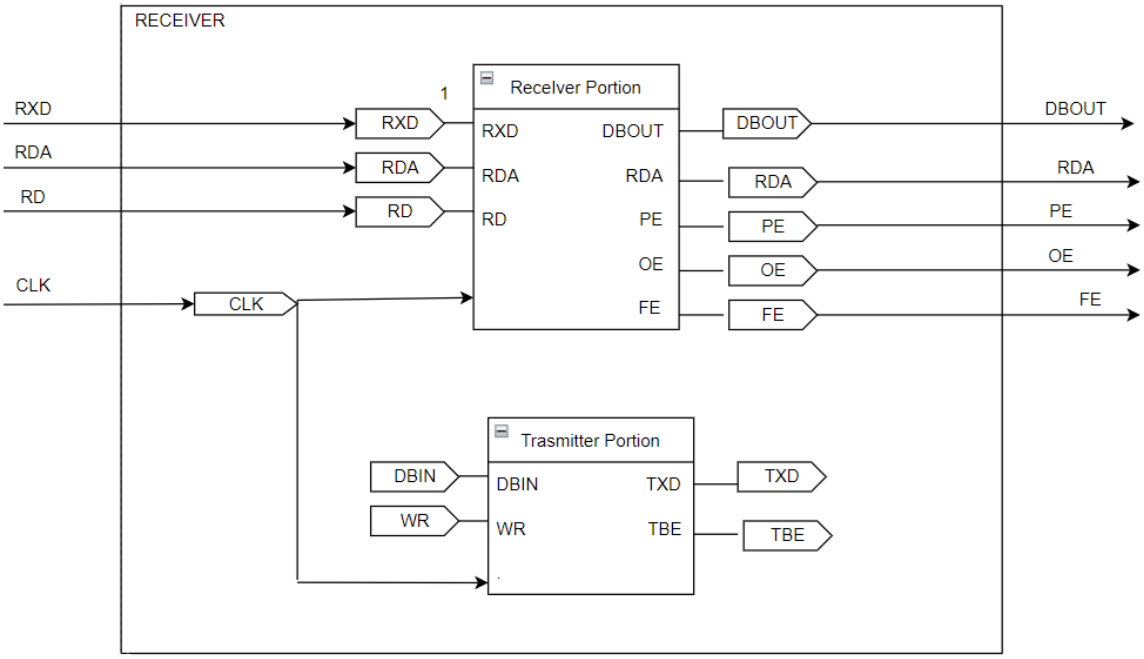
9.2 Soluzione 1

Per poter utilizzare il dispositivo UART messo a disposizione dalla Diligent bisogna prima di tutto analizzarne l'interfaccia e il suo funzionamento. La linea di trasmissione è alta a riposo, la comunicazione inizia a partire da un segnale di start (segnale basso) e termina in corrispondenza di un segnale di stop(segnale alto). Tra questi due segnali è possibile trasmettere in serie una quantità prestabilita di bit nel canale TDX-RDX, nel nostro caso otto.

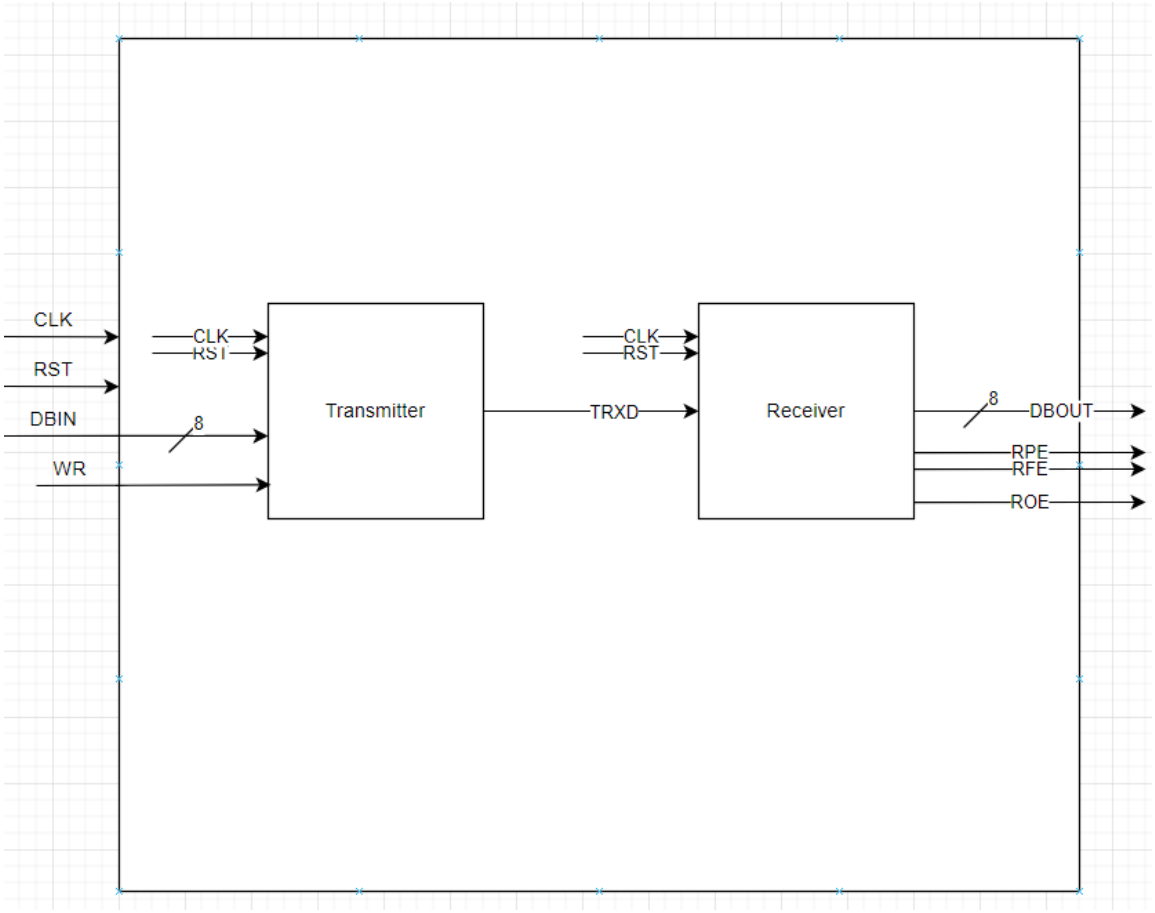
Una volta terminata la comunicazione, il trasmettitore alzerà un flag, TBE, per indicare lo stato di buffer vuoto, mentre il ricevitore alzerà il flag RDA per indicare la presenza di un dato da prelevare.



Per poter pilotare correttamente il trasmettitore bisogna prima di tutto mandare in ingresso il dato da inviare in DBIN e successivamente alzare il segnale WR. Notiamo come RXD=1 altrimenti la parte Receiver dell'UART penserebbe che sia iniziata una comunicazione. Verranno poi inviati i bit di DBIN in modo seriale sul canale TXD a partire da quello meno significativo. Una volta terminata la comunicazione viene alzato il flag TBE per indicare che il buffer è vuoto.



Il ricevitore invece riceve serialmente i bit nel canale RXD, e dopo averli ricevuti tutti sono prelevabili parallelamente da DBOUT. La presenza del dato in DBOUT è indicata dal flag RDA. Nel caso in cui venga iniziata e terminata un'altra comunicazione, sovrascrivendo DBOUT, il receiver alzerà il bit OE (overwrite error). Per evitare che ciò accada bisogna alzare il bit RD quando si legge il dato in DBOUT. I bit PE si alzeranno invece nel caso di errori relativi al bit di parità o errori di framing.



Il receiver e il transmitter sono stati semplicemente collegati in un top module.

9.2.1 Codice 1

TRASMETTITORE:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Transmitter is
    Port (
        DBIN: in std_logic_vector(7 downto 0);
        CLK, RST: in std_logic;
        WR: in std_logic;
        TXD: out std_logic;
        --ERRORE:PE FE OE SERVONO AL RICEVITORE!
        TBE, PE, FE, OE: out std_logic
     );
end Transmitter;

architecture structural of Transmitter is

component UARTcomponent is
    Generic (
        --@48MHz
        BAUD_DIVIDE_G : integer := 26;  --115200
        BAUD_RATE_G   : integer := 417

        --@26.6MHz
--      BAUD_DIVIDE_G : integer := 14;  --115200
--      BAUD_RATE_G   : integer := 231
    );

    Port (
        TXD     : out   std_logic   := '1';             -- Transmitted serial data output
        RXD     : in    std_logic;                      -- Received serial data input
        CLK     : in    std_logic;                      -- Clock signal
        DBIN    : in    std_logic_vector (7 downto 0);  -- Input parallel data to be transmi
        DBOUT   : out   std_logic_vector (7 downto 0);  -- Recevived parallel data output
        RDA     : inout  std_logic;                     -- Read Data Available
        TBE     : out   std_logic   := '1';             -- Transfer Buffer Emty
        RD      : in    std_logic;                      -- Read Strobe
        WR      : in    std_logic;                      -- Write Strobe
        PE      : out   std_logic;                      -- Parity error
        FE      : out   std_logic;                      -- Frame error
        OE      : out   std_logic;                      -- Overwrite error
        RST     : in    std_logic   := '0');            -- Reset signal
end component;

signal useless: std_logic_vector(7 downto 0);
signal RDA: std_logic;

begin

uart: UARTcomponent port map(TXD, '1', CLK, DBIN, useless, RDA, TBE, '0', WR, PE, FE, OE, RST );

end structural;
```

NOTA: in realtà nel trasmettitore non servono i bit PE,FE,OE!

RICEVITORE:

```
entity Receiver is
    Port (
        RXD: in std_logic;
        CLK: in std_logic;
        DBOUT: out std_logic_vector(7 downto 0);
        RD: in std_logic;
        RDA: inout std_logic;
        PE,FE,OE: out std_logic;
        RST: in std_logic
    );
end Receiver;

architecture structural of Receiver is

component UARTcomponent is
    Generic (
        --@48MHz
        BAUD_DIVIDE_G : integer := 26;  --115200 baud
        BAUD_RATE_G   : integer := 417
    );
    Port (
        TXD     : out   std_logic   := '1';               -- Transmitted serial data output
        RXD     : in    std_logic;                        -- Received serial data input
        CLK     : in    std_logic;                        -- Clock signal
        DBIN    : in    std_logic_vector (7 downto 0);    -- Input parallel data to be transmitted
        DBOUT   : out   std_logic_vector (7 downto 0);    -- Recevived parallel data output
        RDA     : inout  std_logic;                       -- Read Data Available
        TBE     : out   std_logic   := '1';               -- Transfer Buffer Emty
        RD      : in    std_logic;                        -- Read Strobe
        WR      : in    std_logic;                        -- Write Strobe
        PE      : out   std_logic;                        -- Parity error
        FE      : out   std_logic;                        -- Frame error
        OE      : out   std_logic;                        -- Overwrite error
        RST     : in    std_logic   := '0');              -- Reset signal
end component;

  signal TBE: std_logic;
  signal TXD: std_logic;

  begin
  uart: UARTcomponent port map(TXD, RXD, CLK, X"00", DBOUT, RDA, TBE, RD, '0', PE, FE, OE, RST);

end structural;
```

TOP MODULE:

```
entity TOP_MODULE is
    Port (
        CLK, RST: in std_logic;
        DBIN: in std_logic_vector(7 downto 0);
        WR: in std_logic;
        DBOUT: out std_logic_vector(7 downto 0);
        --transmitter errors
        TBE, TPE, TFE, TOE: out std_logic;
        --receiver errors
        RPE,RFE,ROE: out std_logic
     );
end TOP_MODULE;

architecture structural of TOP_MODULE is
signal TRXD: std_logic;
signal RDA: std_logic;

begin

t: Transmitter port map(DBIN, CLK, RST, WR, TRXD, TBE, TPE, TFE, TOE);
r: Receiver port map(TRXD, CLK, DBOUT, '0', RDA, RPE, RFE, ROE, RST);

end structural;
```

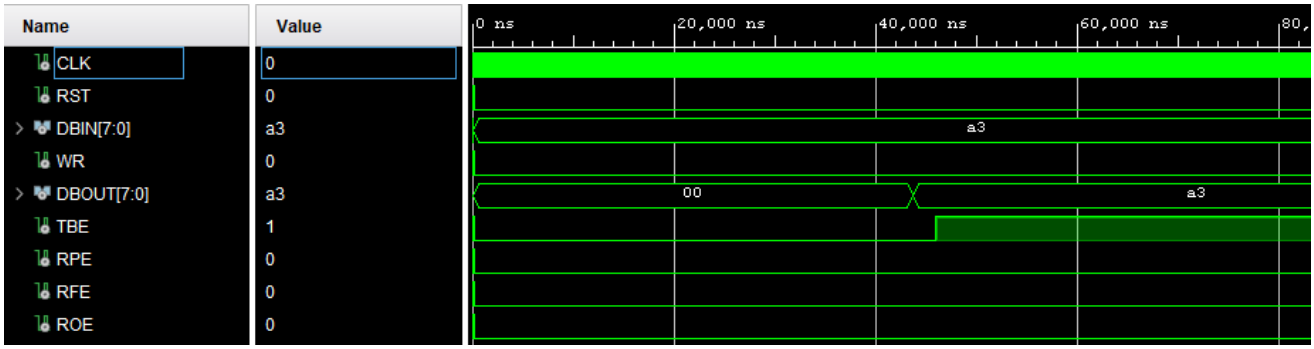
NOTA: Ovviamente anche nel top module non dovrebbero esserci i flag di errori relativi al trasmettitore TPE,TFE,TOE.

9.2.2 Simulazione 1

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica.

```
 stimulus: process
 begin

 wait for 20ns;
 RST<='1';
 DBIN<=X"A3";
 wait for 20ns;
 RST<='0';
 WR<='1';
 wait for 60ns;
 WR<='0';

   wait;
 end process;
```

Il risultato della simulazione è il seguente:



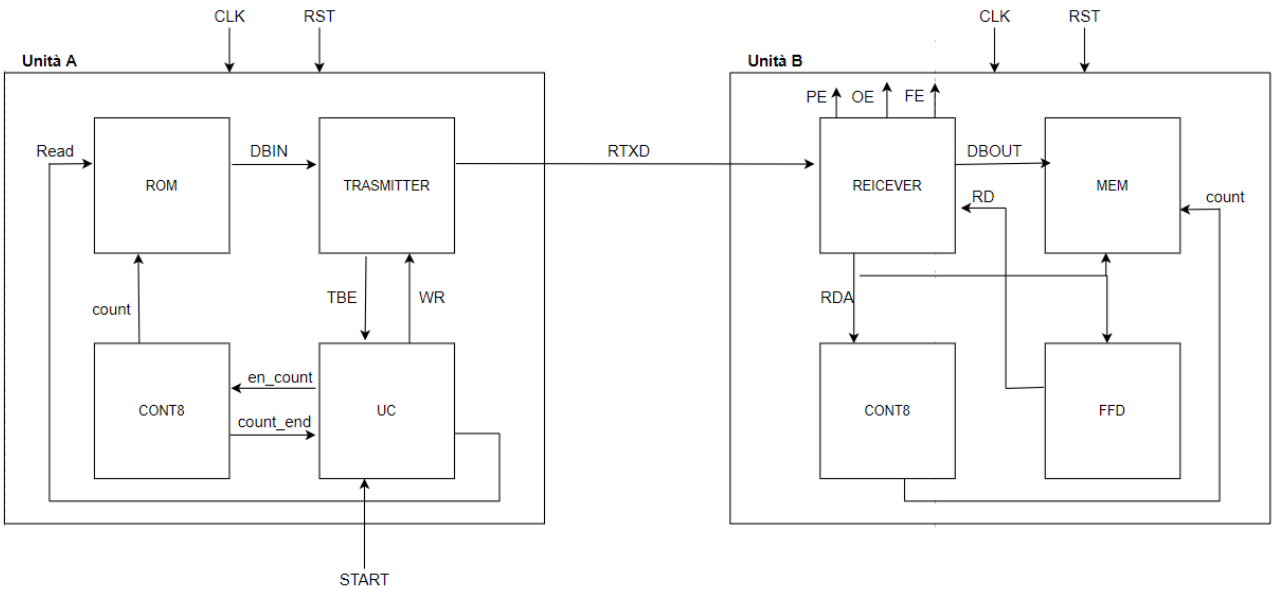
9.2.3 Sintesi 1

Per poter correttamente sintetizzare l'architettura su board è stato semplicemente necessario modificare correttamente il file di constraint.

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


##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { DBIN[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { DBIN[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { DBIN[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { DBIN[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { DBIN[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { DBIN[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { DBIN[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { DBIN[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 { DBOUT[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { DBOUT[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { DBOUT[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { DBOUT[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { DBOUT[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { DBOUT[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { DBOUT[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { DBOUT[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { LED[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { ROE }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { RFE }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { RPE }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { TOE }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { TFE }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { TPE }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { TBE }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

##Buttons
#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { BTNC }]; #IO_L9P_T1_DQS_14 Sch=btnc
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { BTNU }]; #IO_L4N_T0_D05_14 Sch=btnu
#set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { BTNL }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { RST }]; #IO_L10N_T1_D15_14 Sch=btnr
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { WR }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

9.3 Soluzione 2

Come secondo esercizio abbiamo implementato l'UART MEM. In questo caso è stato necessario implementare un contatore, utile a scandire le stringhe da inviare, ed una memoria ROM all'interno dell'unità A, che trasmette i dati. Inoltre, abbiamo implementato sempre all'interno di essa, un'unità di controllo mediante un automa a stati finiti. Invece, all'interno dell'unità B troviamo ancora un contatore modulo 8, un ricevitore e, oltre alla memoria, è stato necessario implementare un Flip Flop D. Successivamente viene mostrato lo schema complessivo dell'UART MEM.



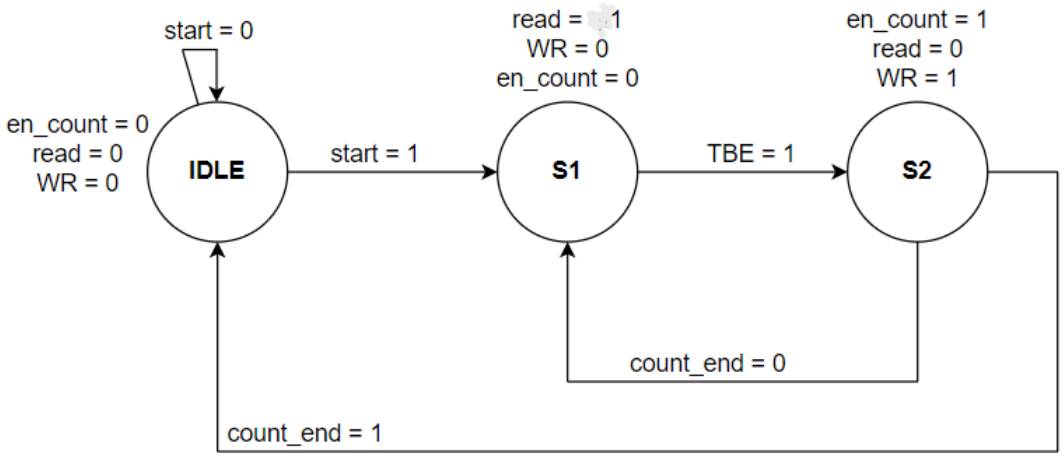
9.3.1 Codice 2

Partiamo dal definire l'unità A. Essa è stata implementata in modo strutturale come composizione di più componenti. In particolare, si compone di una memoria Rom, di un contatore modulo 8, di un trasmettitore e di un'unità di controllo. Sono poi stati definiti dei segnali e mappati opportunamente i componenti all'interno dell'architettura. Di seguito ne vediamo il codice:

```
component Transmitter is
    Port (
        DBIN: in std_logic_vector(7 downto 0);
        CLK, RST: in std_logic;
        WR: in std_logic;
        TXD: out std_logic;
        TBE, PE, FE, OE: out std_logic
     );
end component;

component ROM is
port(
    CLK : in std_logic; -- clock della board
    RST : in std_logic;
    READ : in std_logic; -- segnale che abilita la lettura, inser
    ADDR : in std_logic_vector(2 downto 0); --3 bit di indirizzo
                                            --sono inseriti trami
    DATA : out std_logic_vector(7 downto 0) -- dato su 8 bit lett
    );
end component;

component Cont8 is
    Port (
        clock, reset: in std_logic;
        count_in: in std_logic;
        count_end: out std_logic;
        count: out std_logic_vector(2 downto 0)
    );
end component;

signal DBIN: std_logic_vector(7 downto 0);
signal count: std_logic_vector(2 downto 0);
signal read, en_count,count_end,WR: std_logic;
signal TBE, PE, FE, OE: std_logic;

begin
r: ROM port map(CLK, RST, read, count, DBIN);
cont: Cont8 port map(CLK,RST,en_count,count_end,count);
uc: UCT port map(CLK,RST,START,count_end,TBE,en_count,read,WR);
t: Transmitter port map(DBIN,CLK,RST,WR,TXD,TBE,PE,FE,OE);
```

Per quanto riguarda il trasmettitore è stata sfruttata l'implementazione dell'esercizio precedente, mentre per la Rom e il contatore modulo 8 sono state riprese implementazioni già viste in precedenza.

Invece, l'unità di controllo è stata implementata mediante un automa a stati finiti. Tale automa si compone di tre stati e ne vediamo la rappresentazione di seguito.



Abbiamo poi implementato l'unità B, composta da una memoria, un contatore modulo 8, un ricevitore e da un flip flop.

```
component Cont8 is
    Port (
        clock, reset: in std_logic;
        count_in: in std_logic;
        count_end: out std_logic;
        count: out std_logic_vector(2 downto 0)
    );
end component;

component MEM is
port(
    CLK : in std_logic; -- clock della board
    RST : in std_logic;
    WRITE,READ : in std_logic;
    ADDR : in std_logic_vector(2 downto 0);

    DATA : in std_logic_vector(7 downto 0);
    Y: out std_logic_vector(7 downto 0)
    );
end component;

component Receiver is
    Port (
        RXD: in std_logic;
        CLK: in std_logic;
        DBOUT: out std_logic_vector(7 downto 0);
        RD: in std_logic;
        RDA: inout std_logic;
        PE,FE,OE: out std_logic;
        RST: in std_logic
    );
end component;
```

```
component FFD is
    Port (
        CLK,RST,D: in std_logic;
        Y: out std_logic
     );
end component;

signal DBOUT,Y: std_logic_vector(7 downto 0);
signal RD, RDA, PE, FE, OE: std_logic;
signal count_end: std_logic;
signal count: std_logic_vector(2 downto 0);

begin
c: Cont8 port map(CLK,RST,RDA,count_end,count);
m: MEM port map(CLK,RST,RDA,'0',count,DBOUT,Y);
r: Receiver port map(RXD,CLK,DBOUT,RD,RDA,PE,FE,OE,RST);
ff: FFD port map(CLK,RST,RDA,RD);

end structural;
```

Abbiamo ritenuto necessario inserire il flip flop in quanto, in sua assenza, avremmo mandato RDA direttamente in RD, in questo caso RDA non si sarebbe mai acceso, mentre con la presenza del flip flop risulta essere alto per un colpo di clock. Ne mostriamo l'implementazione:

```
entity FFD is
    Port (
        CLK,RST,D: in std_logic;
        Y: out std_logic
     );
end FFD;

architecture Behavioral of FFD is

begin

FF_D: process(CLK,RST)
begin
if(CLK'event and CLK='1')then
    if(RST='1') then
        Y<='0';
    else
        Y<=D;
    end if;
end if;
end process;

end Behavioral;
```

Infine, abbiamo definite l'entità Top Module in cui andiamo semplicemente a collegare tra loro le due entità:

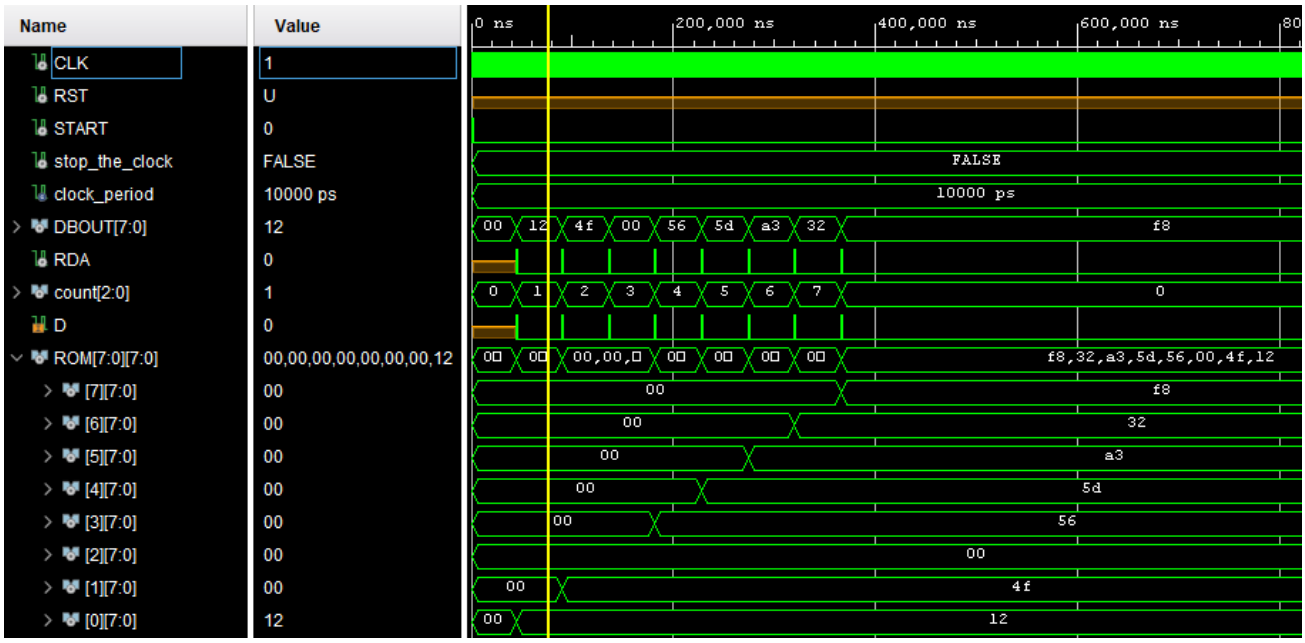
```
entity TOP_MODULE is
    Port (
        CLK,RST,START: in std_logic
     );
end TOP_MODULE;

architecture structural of TOP_MODULE is

component EntityA is
    Port (
        CLK,RST,START: in std_logic;
        TXD: out std_logic
     );
end component;

component EntityB is
    Port (
        CLK,RST,RXD: in std_logic
     );
end component;

signal RTXD: std_logic;
begin

A: EntityA port map(CLK,RST,START,RTXD);
B: EntityB port map(CLK,RST,RTXD);

end structural;
```

9.3.2 Simulazione 2

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica.

```
uut: TOP_MODULE port map ( CLK   => CLK,
                           RST   => RST,
                           START => START );

stimulus: process
begin
  wait for 20ns;
  START<='1';
  wait for 100ns;
  START<='0';


  wait;
end process;
clocking: process
begin
  while not stop_the_clock loop
    CLK <= '0', '1' after clock_period / 2;
    wait for clock_period;
  end loop;
  wait;
end process;
```

Il risultato della simulazione è il seguente:



Capitolo 10

Switch Multistadio

10.1 Traccia

Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network. Lo switch progettato deve operare come segue:

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

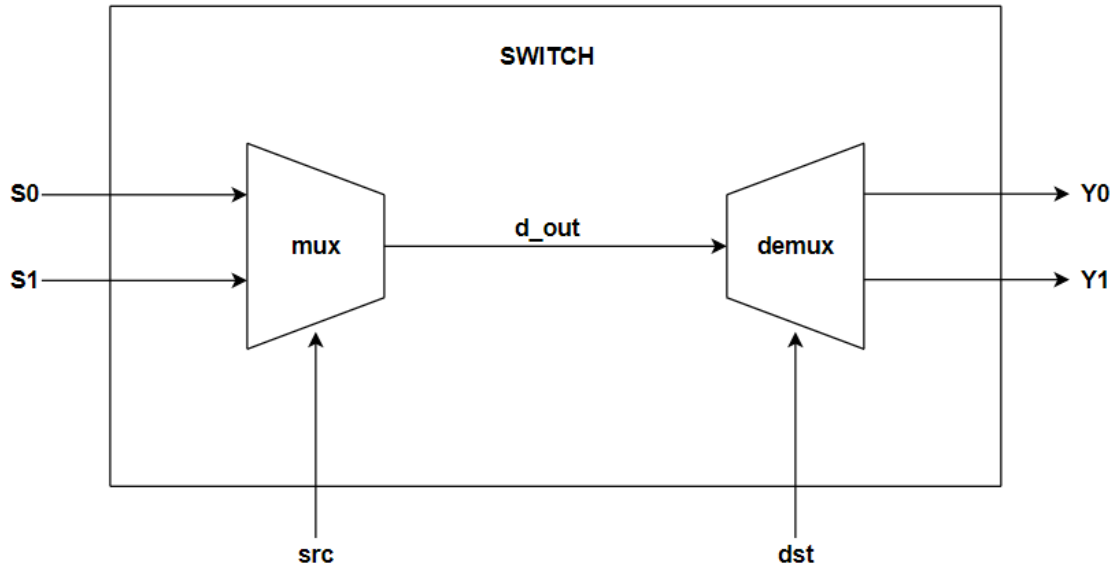
10.1.1 Soluzione

Decomponiamo la macchina in unità operativa, che realizza la rete di switch secondo il modello Omega Network, e unità di controllo, che realizza la rete di priorità per permettere ad un solo nodo alla volta di effettuare la comunicazione.

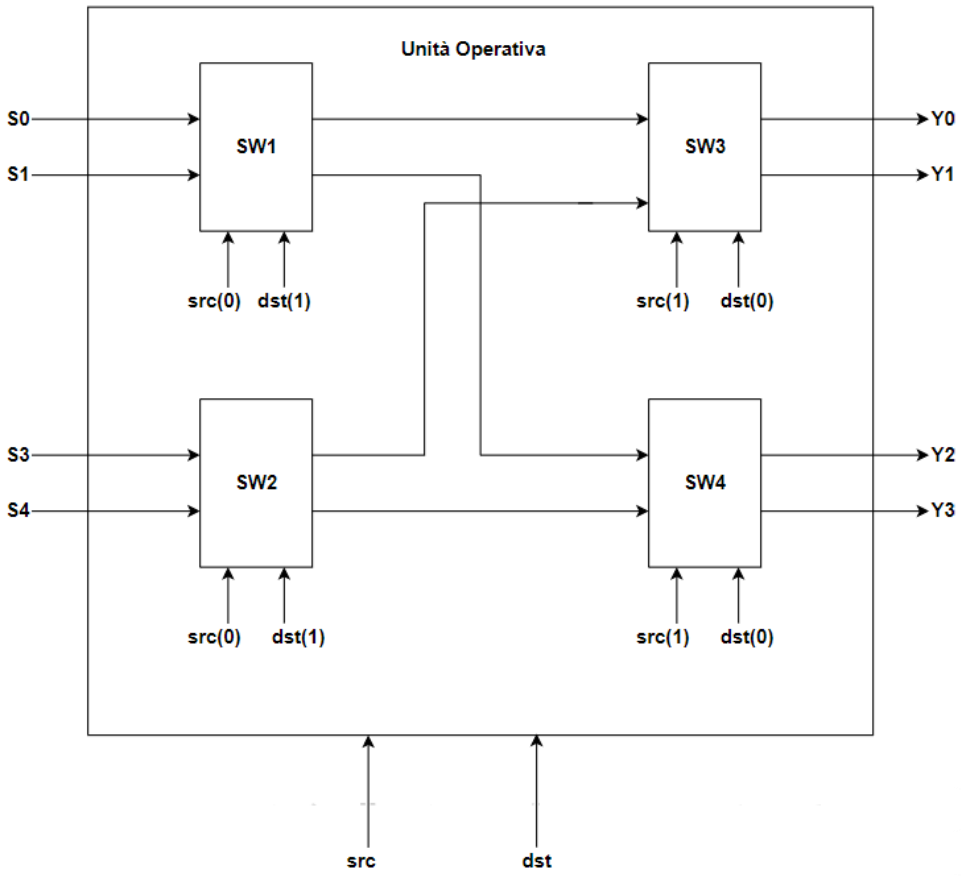
L'elemento alla base dell'unità operativa è lo switch, realizzato tramite composizione di un mux 2:1 e un demux 1:2, prende in ingresso un bit di indirizzo sorgente e un bit di indirizzo destinazione.

L'Omega Network è una rete realizzata mediante la composizione di più switch, quindi di più mulitplexer e demultiplexer, composti secondo lo schema definito dal "perfect shuffling" che ci permette di ottimizzare i percorsi da seguire, indicando i modi di collegamento tra i diversi stadi.

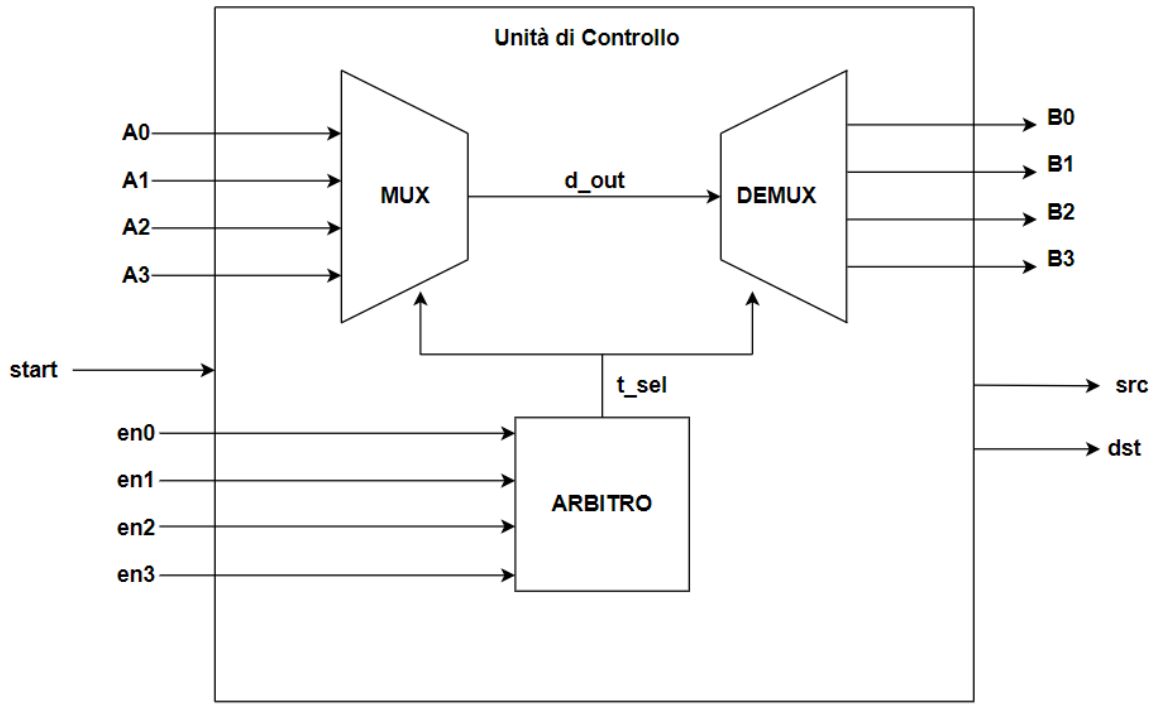
Vediamo, nell'immagine sottostante, lo schema dello switch:



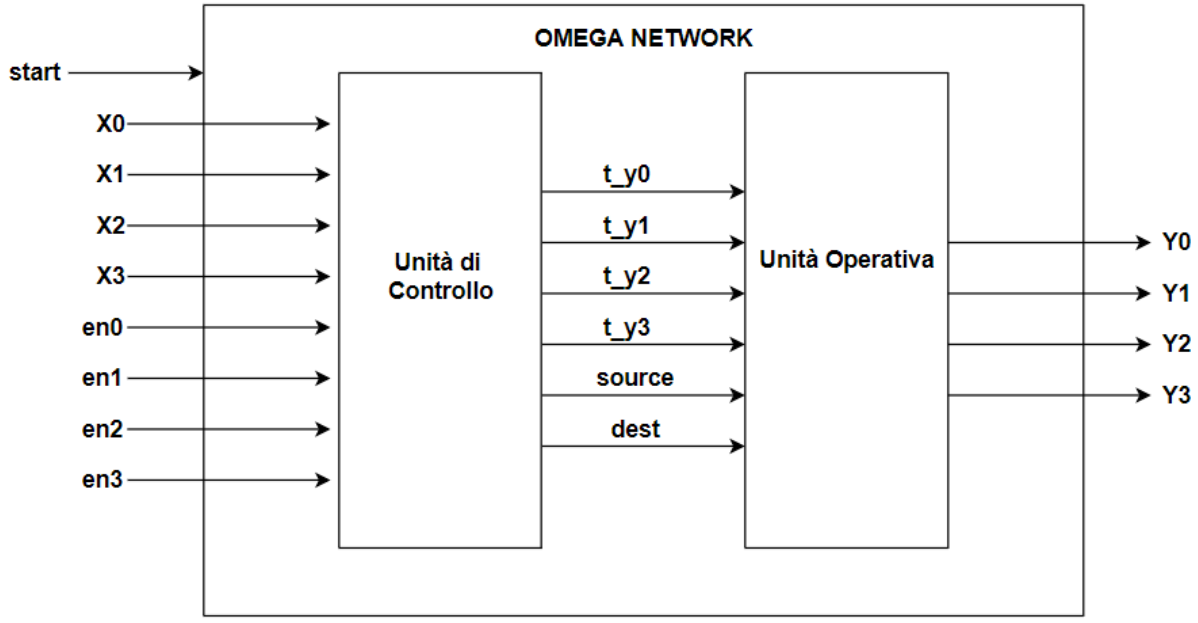
Successivamente rappresentiamo l'unità operativa,nella quale viaggiano solo i bit di dato. Viene realizzata con approccio strutturale, componendo gli switch e collegandoli in accordo con il perfect shuffling. In questo caso abbiamo 4 nodi, quindi 2 stadi con 2 switch ciascuno. Le stringhe contenenti gli indirizzi sono lunghe 2 bit in quanto ci sono 4 possibili indirizzi.



L'unità di controllo, invece, realizza la rete di priorità costituita da una coppia di multiplexer 4:1 e demultiplexer 1:4 e da un arbitro che di volta in volta decide, in base alla maggiore priorità, quale nodo può trasmettere. Inoltre, l'unità di controllo, produce le informazioni su sorgente e destinatario, necessarie all'unità operativa, prelevando i primi 2 bit per il sorgente e i successivi 2 per il destinatario.



Infine, descriviamo l'Omega Network come composizione di unità di controllo e unità operativa.



10.1.2 Codice

Abbiamo implementato dapprima lo switch, realizzato come composizione di un mux 2:1 e un demux 1:2. Sfruttiamo, in tutti i codici, il costrutto generic che ci permette di modificare

eventualmente il numero di bit di effettiva informazione trasmessa, ma non i bit di sorgente o destinatario che invece dipendono dal numero di nodi che devono comunicare.

```
entity switch is

 generic ( N: natural := 2 );  --dimensione del pacchetto dati
    Port ( S0, S1 : in std_logic_vector ((N-1) downto 0);
           sm, sd : in STD_LOGIC;  --selezione mux e demux
           Y0, Y1 : out std_logic_vector ((N-1) downto 0)
           );
end switch;

architecture structural of switch is

component mux_21 is
generic ( N: natural := 2 );
port (S0, S1: in std_logic_vector ((N-1) downto 0);
      s: in std_logic;  --ingresso di selezione
      Y: out std_logic_vector ((N-1) downto 0)
      );
end component;

component demux_12 is
generic ( N: natural := 2 );
port ( X: in std_logic_vector ((N-1) downto 0);
       s: in std_logic; --ingresso di selezione
       Y0, Y1: out std_logic_vector ((N-1) downto 0)
       );
end component;

signal d_out: std_logic_vector ((N-1) downto 0);

begin
      mux: mux_21 generic map(N)
                  port map (S0, S1, sm, d_out);

  demux: demux_12 generic map(N)
                  port map (d_out, sd, Y0, Y1);

end structural;
```

Successivamente abbiamo implementato l'unità operativa, realizzata ancora con approccio strutturale, componendo 4 switch.

```
architecture structural of unita_operativa is

component switch is

generic ( N: natural := 2 );  --dimensione del pacchetto dati
    Port ( S0, S1 : in std_logic_vector ((N-1) downto 0);
           sm, sd : in STD_LOGIC;  --selezione mux e demux
           Y0, Y1 : out std_logic_vector ((N-1) downto 0)
           );
end component;

signal t10, t11, t20, t21: std_logic_vector ((N-1) downto 0);

begin
sw1: switch generic map(N)
           port map (S0, S1, src(0), dst(1), t10, t11);
sw2: switch generic map(N)
           port map (S2, S3, src(0), dst(1), t20, t21);
sw3: switch generic map(N)
           port map (t10, t20, src(1), dst(0), Y0, Y1);
sw4: switch generic map(N)
           port map (t11, t21, src(1), dst(0), Y2, Y3);

end structural;
```

Abbiamo implementato poi l'unità di controllo in modo strutturale, composta da un arbitro, un mux 4:1 e un demux 1:4

```
entity unita_controllo is
generic (N: natural := 2);
    Port ( start: in std_logic;
           A0, A1, A2, A3 : in std_logic_vector ((4+(N-1)) downto 0);
           en0, en1, en2, en3 : in STD_LOGIC;  --segnali di enable
           B0, B1, B2, B3 : out std_logic_vector ((N-1) downto 0);
           s, d : out std_logic_vector (1 downto 0)
           );
end unita_controllo;

architecture structural of unita_controllo is
component mux_41 is
 generic (N: natural := 2);
 port ( start: in std_logic;
        S0, S1, S2, S3: in std_logic_vector ((N-1) downto 0);
        s: in std_logic_vector (1 downto 0);  --selezione
        Y: out std_logic_vector ((N-1) downto 0)
        );
 end component;
```

```
signal t_sel: std_logic_vector ( 1 downto 0 );
signal d_out: std_logic_vector ((N-1) downto 0);

begin
mux: mux_41 generic map(N)
           port map ( start, A0(1 downto 0), A1(1 downto 0), A2( 1 downto 0),
                     A3(1 downto 0), t_sel, d_out );
demux: demux_14 generic map(N)
               port map ( start, d_out, t_sel, B0, B1, B2, B3 );

arb: arbitro port map ( en0, en1, en2, en3, t_sel );

s <= A0 ( 5 downto 4 ) when t_sel = "00" else
     A1 ( 5 downto 4 ) when t_sel = "01" else
     A2 ( 5 downto 4 ) when t_sel = "10" else
     A3 ( 5 downto 4 ) when t_sel = "11" else
     "--";

d <= A0 ( 3 downto 2 ) when t_sel = "00" else
     A1 ( 3 downto 2 ) when t_sel = "01" else
     A2 ( 3 downto 2 ) when t_sel = "10" else
     A3 ( 3 downto 2 ) when t_sel = "11" else
     "--";
```

Mostriamo successivamente anche l'implementazione dell'arbitro:

```
entity arbitro is
    Port ( en0, en1, en2, en3 : in STD_LOGIC;
           y : out std_logic_vector ( 1 downto 0 )
           );
end arbitro;

architecture dataflow of arbitro is

begin
      y <= "00" when en0 = '1' else  --in uscita ho 00 se il nodo 0 vuole trasmettere
           "01" when en1 = '1' else  --01 se il nodo 0 non vuole trasmettere e il nodo 1 si
           "10" when en2 = '1' else  --10 se 0 e 1 non vogliono trasmettere ma il 2 si
           "11" when en3 = '1' else  --11 se 0, 1 e 2 non vogliono trasmettere e 3 si
           "--";
end dataflow;
```

Infine, abbiamo implementato l'Omega Network collegando opportunamente unità di controllo e unità operativa:

```
    entity omega_network is
    generic (N: natural := 2);
        Port ( start : in STD_LOGIC;
               X0, X1, X2, X3 : in std_logic_vector ((N-1) downto 0);
               en0, en1, en2, en3 : in STD_LOGIC;
               Y0, Y1, Y2, Y3 : out std_logic_vector ((N-1) downto 0)
               );
    end omega_network;

    architecture structural of omega_network is

    component unita_controllo is
    generic (N: natural := 2);
        Port ( start: in std_logic;
               A0, A1, A2, A3 : in std_logic_vector ((4+(N-1)) downto 0);
               en0, en1, en2, en3 : in STD_LOGIC;  --segnali di enable
               B0, B1, B2, B3 : out std_logic_vector ((N-1) downto 0);
               s, d : out std_logic_vector (1 downto 0)
               );
    end component;

component unita_operativa is
generic ( N: natural := 2 ); --dimensione del pacchetto dati
   Port ( S0, S1, S2, S3 : in std_logic_vector ((N-1) downto 0);
          src, dst : in std_logic_vector ( 1 downto 0 );  --2 bit sorgente e desti
          Y0, Y1, Y2, Y3 : out std_logic_vector ((N-1) downto 0)
          );
end component;

signal t_y0, t_y1, t_y2, t_y3: std_logic_vector ((N-1) downto 0);
signal source: std_logic_vector ( 1 downto 0 );
signal dest: std_logic_vector ( 1 downto 0 );

begin
     contr: unita_controllo generic map(N)
                           port map ( start, X0, X1, X2, X3, en0, en1, en2, en3,
                                      t_y0, t_y1, t_y2, t_y3, source, dest);

     op: unita_operativa generic map(N)
                         port map ( t_y0, t_y1, t_y2, t_y3, source, dest,
                                    Y0, Y1, Y2, Y3);

end structural;
```

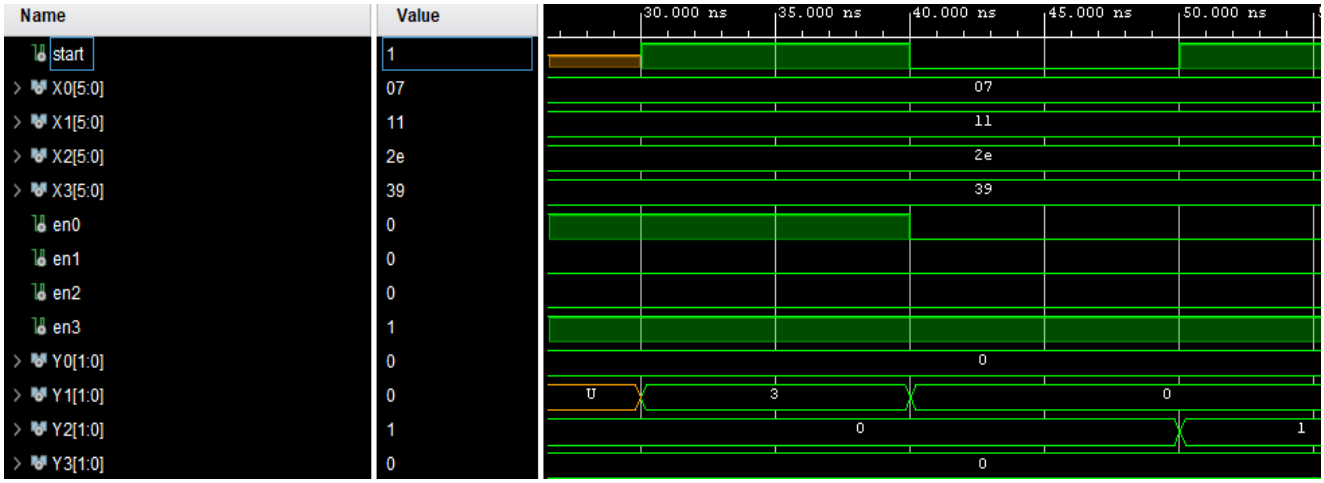
10.1.3 Simulazione

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica.

```
stimulus: process
begin

wait for 20ns;
en0 <= '1';
en1 <= '0';
en2 <= '0';
en3 <= '1';
X0 <= "000111";
X1 <= "010001";
X2 <= "101110";
X3 <= "111001";
wait for 10ns;
start <= '1';
wait for 10ns;
start <= '0';
en0 <= '0';
wait for 10ns;
start <= '1';
```

Il valore atteso è “11” nella destinazione 1 e poi successivamente il valore “01” nella destinazione 2.

Il risultato della simulazione è il seguente:



Capitolo 11

Macchine Aritmetiche

11.1 Traccia

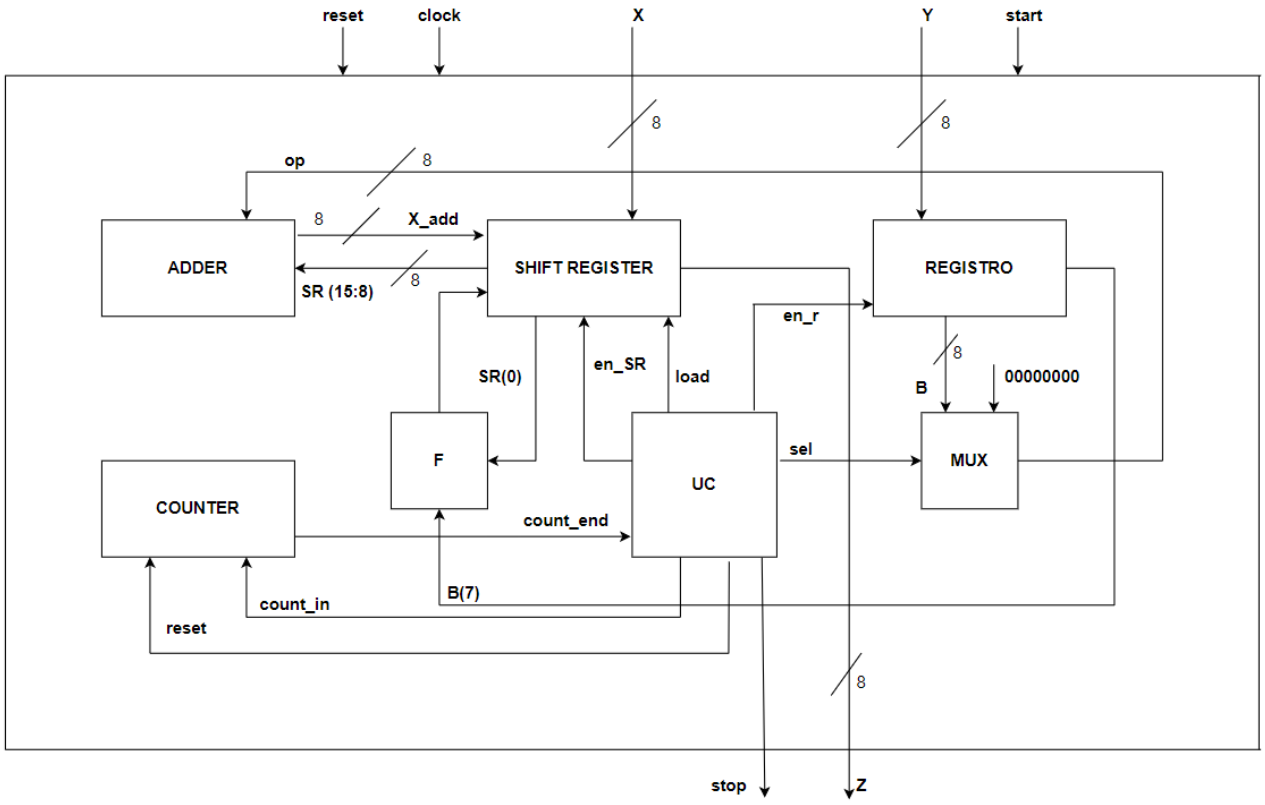
Progettare ed implementare in VHDL una macchina aritmetica sequenziale a scelta fra le seguenti:

- 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.

11.1.1 Soluzione

Abbiamo deciso di implementare il moltiplicatore di Robertson che è un moltiplicatore sequenziale. Per implementarlo si è ritenuto utile scomporre l'architettura in parte operativa e parte di controllo. La parte operativa è stata realizzata con un approccio strutturale, mentre la parte di controllo è stata realizzata con un automa a stati finiti. I blocchi che costituiscono l'unità operativa sono: un registro, uno shift register, un adder, un multiplexer e un contatore. Di seguito vediamo illustrato lo schema del moltiplicatore:



11.1.2 Codice

Come primo componente abbiamo implementato l'adder. Mediante esso riusciamo a calcolare la somma parziale; è un ripple carry adder che effettua anche la sottrazione, questo serve per gestire il passo di correzione. è formato da due blocchi, il ripple carry adder e un blocco che effettua il complemento delle cifre. Infatti tale adder ha in ingresso una XOR tra B e il riporto in ingresso. Nel caso in cui si voglia sommare sommare A e B prende il riporto in ingresso pari a 0, in modo tale che nell'adder entra B e la somma viene effettuata normalmente; viceversa, per effettuare la sottrazione, l'operando B viene convertito in -B in complemento a due, avendo come riporto in ingresso 1.

```
    Port ( X, Y : in std_logic_vector ( 7 downto 0 );
           cin : in STD_LOGIC;
           cout : out STD_LOGIC;
           Z : out std_logic_vector ( 7 downto 0 )
           );
end adder_sub;

architecture structural of adder_sub is

component ripple_carry is
  Port ( X, Y : in std_logic_vector ( 7 downto 0 );
         c_in : in STD_LOGIC;
         c_out : out STD_LOGIC;
         Z : out std_logic_vector ( 7 downto 0 )
         );
end component;

signal complementoy: std_logic_vector ( 7 downto 0 );

begin
complemento_y: for i in 0 to 7 generate
               complementoy(i) <= Y(i) xor cin;
               end generate;

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

Come secondo componente abbiamo implementato lo shift-register, il quale mantiene il valore del moltiplicatore. Inizialmente contiene una stringa di 8 zeri e X (moltiplicatore), poi ad ogni passo effettua lo shift e sfrutta le posizioni che sono state liberate dal moltiplicatore per memorizzare le cifre del risultato parziale. In questo registro se il reset è alto l'uscita sarà 0, se il segnale di load è alto carico l'ingresso, se invece è alto load_add carico nei primi 8 bit il risultato della somma parziale ottenuta dall'adder.

```
entity shif_register is
    port( X: in std_logic_vector(15 downto 0);
          F: in std_logic;
          clock, reset, load, en, load_add, shift, f_shift:in std_logic;
          x_add: in std_logic_vector(7 downto 0);
          Y: out std_logic_vector(15 downto 0));
end shif_register;

architecture behavioural of shif_register is

    signal temp: std_logic_vector(15 downto 0);
    signal temp1: std_logic_vector(15 downto 0);
    begin
```

```
architecture Behavioral of shift_register is

signal temp: std_logic_vector ( 15 downto 0 );

begin
 SR: process ( clock, reset )
     begin
     if ( en = '1' ) then
          if ( reset = '1' ) then
               temp <= ( others => '0' );
       elsif ( clock' event and clock = '1' ) then
          if ( load = '1' ) then
               temp <= X;
       elsif ( load_add = '1' ) then
               temp ( 15 downto 8 ) <= x_add;
               --x_add è la somma parziale ricevuta dall'adder
       elsif ( shift = '1' ) then
               temp ( 14 downto 0 ) <= temp ( 15 downto 1 );
               temp (15) <= F;
       elsif ( f_shift = '1' ) then
               temp ( 14 downto 0 ) <= temp ( 15 downto 1 );

        end if;
       end if;
      end if;
```

Successivamente è stato implementato il registro, di tipo parallelo-parallelo, che mantiene il valore del moltiplicando Y, espresso su 8 bit. Quando il segnale di reset è alto si pone l'uscita pari a 0, invece ad ogni colpo di clock si pone l'uscita pari all'ingresso.

```
entity registro is  --registro parallelo-parallelo
    Port ( A : in std_logic_vector ( 7 downto 0 );
           clk, res, en : in STD_LOGIC;
           B : out std_logic_vector ( 7 downto 0 )
           );
end registro;

architecture Behavioral of registro is

begin

R_PP: process ( clk, res )
      begin
      if ( en = '1' ) then
      if ( res = '1' ) then
           B <= ( others => '0' );
   elsif ( clk' event and clk = '1' ) then
           B <= A;
       end if;
      end if;
end process;
```

Implementiamo poi il contatore, che è un contatore modulo 8, il quale conta i passi dell'unità di controllo per capire quando si arriva al passo finale (count = 111). Dopodiché l'UC passa ad effettuare il passo di correzione e lo shift finale.

```
entity counter is
    Port ( clock, reset : in STD_LOGIC;
           count_in : in STD_LOGIC; --glielo passa l'UC ad ogni iterazione
           count_end : out STD_LOGIC;
           count : out std_logic_vector ( 2 downto 0 )
           );
end counter;

architecture Behavioral of counter is

signal c: std_logic_vector ( 2 downto 0 );
--numero di passi eseguiti dall'UC, arrivati al passo 7 il contatore si resetta
begin
CM8: process ( clock, reset, count_in )
begin
     if ( reset = '1' ) then
          c <= ( others => '0' );
          count_end <= '0';
  elsif ( clock' event and clock = '1' and count_in = '1' ) then
          c <= std_logic_vector ( unsigned(c) + 1 );
          if ( c = "111" ) then count_end <= '1';
          else count_end <= '0' ;
  end if;
  end if;
  end process;
  count <= c;
```

Infine, nella parte operativa, sono presenti un flip flop D e un multiplexer, per effettuare la moltiplicazione, e se la cifra è 1 il risultato sarà il moltiplicando stesso, altrimenti avremo una stringa di 0. Invece il flip flop D serve per mantenere il segno.

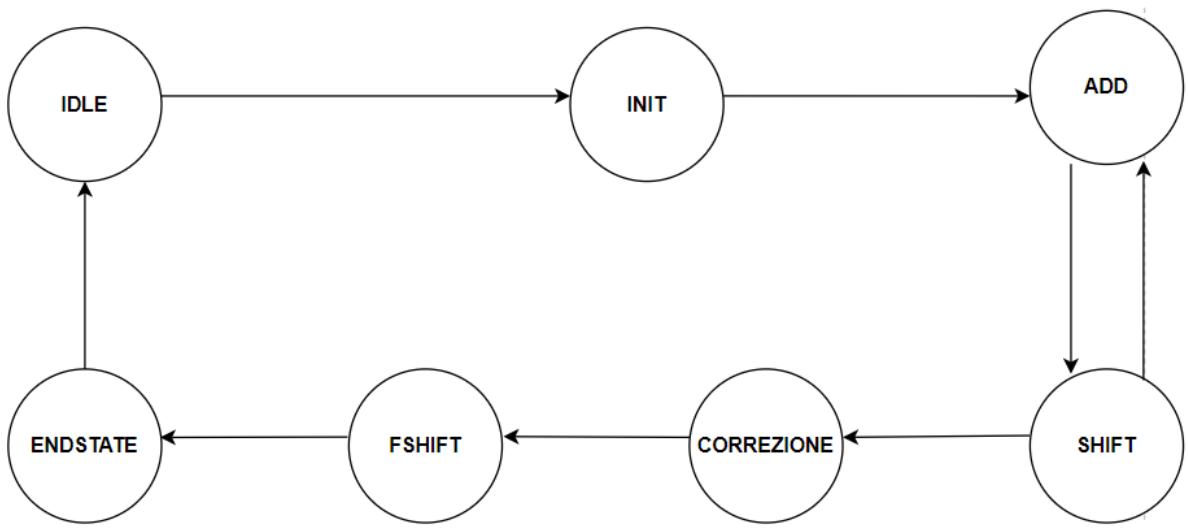
```
entity FFD is
    Port ( clock, reset : in STD_LOGIC;
           d : in STD_LOGIC;
           en : in STD_LOGIC;
           y : out STD_LOGIC
           );
end FFD;

architecture Behavioral of FFD is

begin

FF_D: process ( clock, reset )
      begin
      if ( en = '1' ) then
       if ( reset = '1' ) then
        y <= '0';
      elsif ( clock' event and clock = '1' ) then
       y <= d;
      end if;
     end if;
end process;

end Behavioral;
```

Unità operativa.

```
entity unita_operativa is
    port( X, Y: in std_logic_vector(7 downto 0);
          clock, reset, load, sub: in std_logic;
          en_SR, en_R8, en_D, count_in, load_add, shift, f_shift: in std_logic;
          count: out std_logic_vector(2 downto 0);
          P: out std_logic_vector(15 downto 0));
end unita_operativa;

architecture structural of unita_operativa is
```

```
    signal temp1: std_logic_vector(7 downto 0); --segnale temporaneao tra reg8 M e mux 21
    signal op1: std_logic_vector(7 downto 0); --segnale temporaneo di uscita dal multiplexer tra mux e adder
    signal temp2: std_logic_vector(15 downto 0); --segnale temporaneo per definire input all'SR
    signal temp_p: std_logic_vector(15 downto 0); --segnale temporaneo uscita dell'SR
    signal temp_d: std_logic :='0';
    signal temp_F: std_logic;
    signal tempadd: std_logic_vector(7 downto 0); --uscita del parallel adder
    signal riporto: std_logic; -- riporto in uscita dell'adder che non utilizziamo
    signal countend: std_logic;
begin

    M: registro8 port map(Y, clock, reset, en_R8,temp1);
    MUX: mux_21 port map("00000000", temp1, temp_p(0), op1);

    temp2<="00000000" & X;
    SR: shif_register port map(temp2, temp_F, clock, reset, load, en_SR, load_add, shift, f_shift, tempadd, temp_p);
    temp_d<=(temp1(7) and temp_p(0)) or temp_F; --aggiorno F
    D: FFD port map(clock, reset, temp_d, en_D, temp_F);
    ADD_SUB: adder_sub port map(temp_p(15 downto 8), op1, sub, tempadd, riporto);
    CONT: cont_mod8 port map(clock, reset, count_in, countend, count);
    P<=temp_p;


end structural;
```

Come ultimo componente resta l'Unità di Controllo, implementata con un automa a stati finiti.



L'UC fornisce i segnali di controllo all'unità operativa per effettuare correttamente la moltiplicazione. Vediamo successivamente il codice che descrive l'automa rappresentato precedentemente:

```
entity unita_controllo is
    port( q0, clock, reset, start: in std_logic;
            count: in std_logic_vector(2 downto 0);
          en_SR, en_R8, en_D, count_in, load_add, en_shift: out std_logic;
          reset_out, subtract, load, en_fshift, finish: out std_logic);
end unita_controllo;

architecture structural of unita_controllo is
    type state is (idle, init, add, shift, correzione, f_shift, endstate);
    signal current_state, next_state: state;

    begin
    reg_stato: process(clock, reset)
              begin
              if(reset='1') then
                current_state<=init;
              elsif(clock'event and clock='0') then
                current_state<=next_state;
              end if;
              end process;

    comb: process(current_state, count, start)
          begin
          en_D<='0';
          en_R8<='0';
          en_SR<='0';
          count_in<='0';
          subtract<='0';
          load_add<='0';
          load<='0';
          finish<='0';
          en_fshift<='0';
          en_shift<='0';
          reset_out<='0';

          CASE current_state is
          WHEN idle => reset_out<='1';
                       en_D<='1'; --serve l'abilitazione alta altrimenti non resetta
                       en_R8<='1';
                       en_SR<='1';
                       if(start='1') then next_state<=init;
                       else next_state<=idle;
                       end if;
```

```
  WHEN init =>
               en_D<='1';
               en_R8<='1'; --carica y in acquisizione
               en_SR<='1';
               load<='1'; --carichiamo x
               next_state<=add;
 --aggiorna sr[15:8] con il risultato dell'adder
 WHEN add =>  en_SR<='1';
              en_D<='1';
              load_add<='1'; --aggiorna sr[15:8] con la somma parziale
              en_R8<='1';
              next_state<=shift;
--esegue lo shift e controlla count
WHEN shift => en_shift<='1';
              en_SR<='1'; --sr abilitato senza caricare x
              en_R8<='1';
              count_in<='1';
              if(count="111") then
                           subtract<='1';
                           next_state<=correzione;
                            --predispone la sottrazione
              else next_state<=add;
              end if;
--in correzione fa la sottrazione e aggiorna SR[15:8]
WHEN correzione =>  subtract<='1';
                    load_add<='1';
                    en_SR<='1';
                    en_R8<='1';
                    next_state<=f_shift;
--final shift
WHEN f_shift => en_SR<='1';
                en_fshift<='1';
                en_R8<='1';
                next_state<=endstate;
WHEN endstate => finish<='1';
                 next_state<=idle;
  end CASE;
  end process;
  end structural;
```

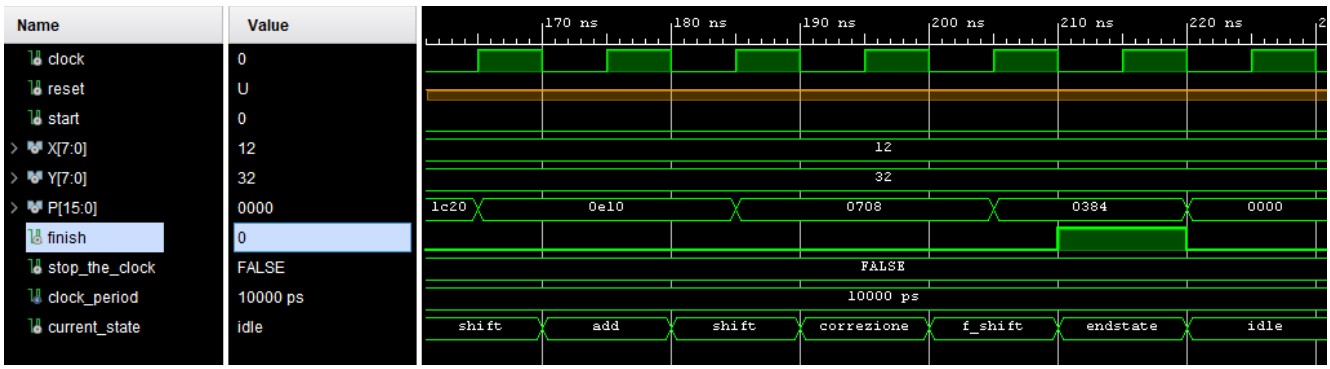
In definitiva il moltiplicatore di Robertson è stato costruito in questo modo.

```
entity Robertson is
     port( clock, reset, start: in std_logic;
           X, Y: in std_logic_vector(7 downto 0);
           P: out std_logic_vector(15 downto 0);
           finish: out std_logic);
end Robertson;

architecture structural of Robertson is
    component unita_controllo is
        port( q0, clock, reset, start: in std_logic;
             count: in std_logic_vector(2 downto 0);
          en_SR, en_R8, en_D, count_in, load_add, en_shift: out std_logic;
          reset_out, subtract, load,en_fshift, finish: out std_logic);
    end component;

    component unita_operativa is
    port( X, Y: in std_logic_vector(7 downto 0);
          clock, reset, load, sub: in std_logic;
          en_SR, en_R8, en_D, count_in, load_add, shift, f_shift: in std_logic;
          count: out std_logic_vector(2 downto 0);
          P: out std_logic_vector(15 downto 0));
    end component;
```

```
signal tempq0, temp_clock, temp_res, temp_sub,temp_load: std_logic;
signal temp_count: std_logic_vector(2 downto 0);
signal temp_p: std_logic_vector(15 downto 0);
signal ab_sr, ab_r8, ab_d, ab_c, t_load_add: std_logic;
signal fine_conteggio: std_logic;
signal temp_shift, temp_fshift: std_logic;

begin
UC: unita_controllo port map(tempq0, clock, reset, start, temp_count, ab_sr, ab_r8, ab_d, ab_c, t_load_add, temp_shift, temp_res, temp_sub, temp_load, temp_fshift, finish);
UO: unita_operativa port map(X, Y, clock, temp_res, temp_load, temp_sub, ab_sr, ab_r8, ab_d,ab_c,t_load_add, temp_shift, temp_fshift, temp_count, temp_p);
tempq0<=temp_p(0);
P<=temp_p;
end structural;
```

11.1.3 Simulazione

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica.
X=X”12”, Y=X”32” => P=X”0384”



Capitolo 12

Esercizio Libero

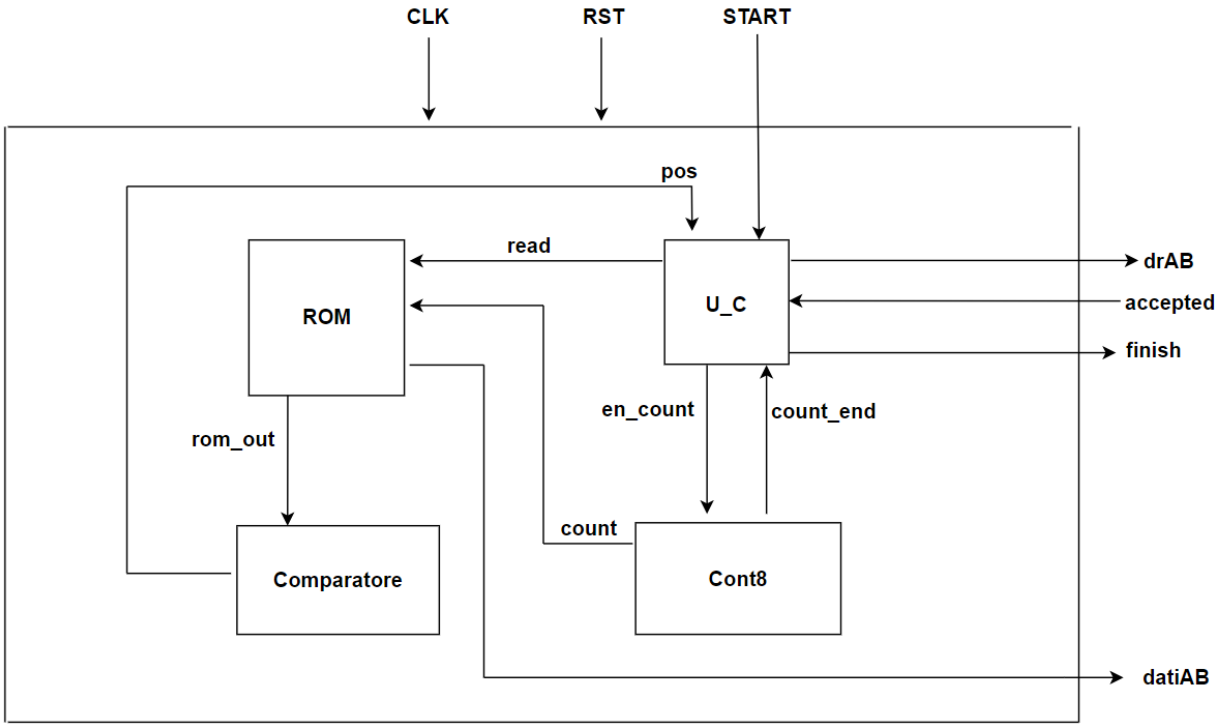
12.1 Traccia

Si consideri un nodo A che contiene una memoria ROM di N (N>=4) locazioni da 8 bit ciascuna. Progettare un sistema in grado di trasmettere mediante handshaking completo tutti i valori strettamente positivi contenuti nella memoria di A ad un nodo B. Il nodo B, ricevuti i valori di A, li trasmetterà ad un nodo C mediante una comunicazione parallela con handshaking.

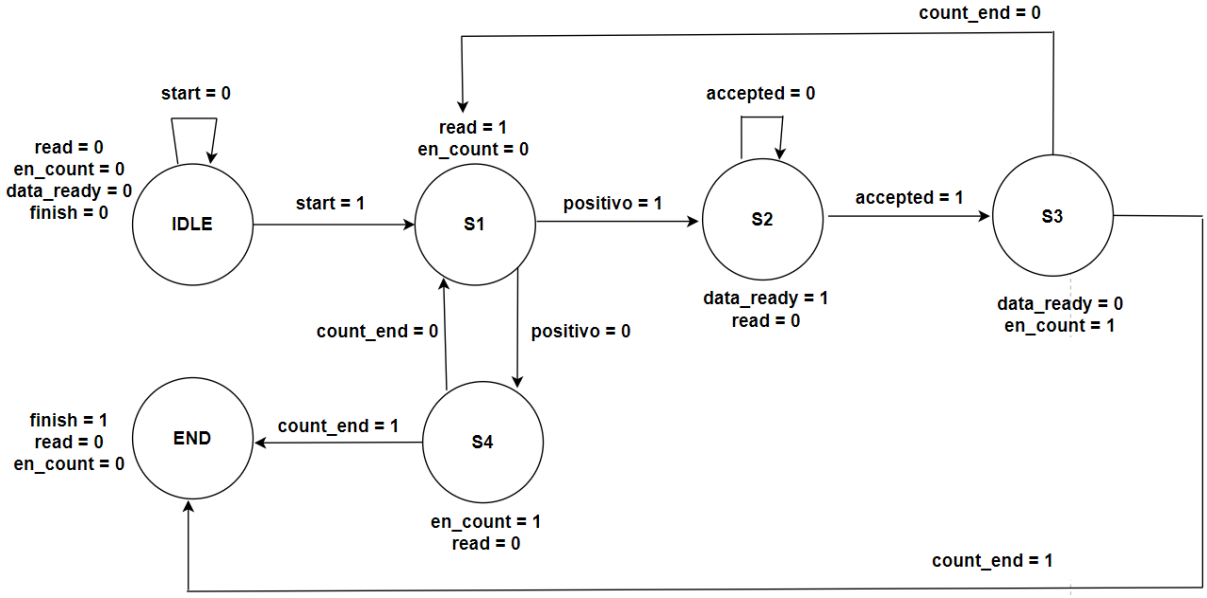
12.2 Soluzione

L'architettura di tutte le unità è stata decomposta in unità operativa e unità di controllo.

Unità A

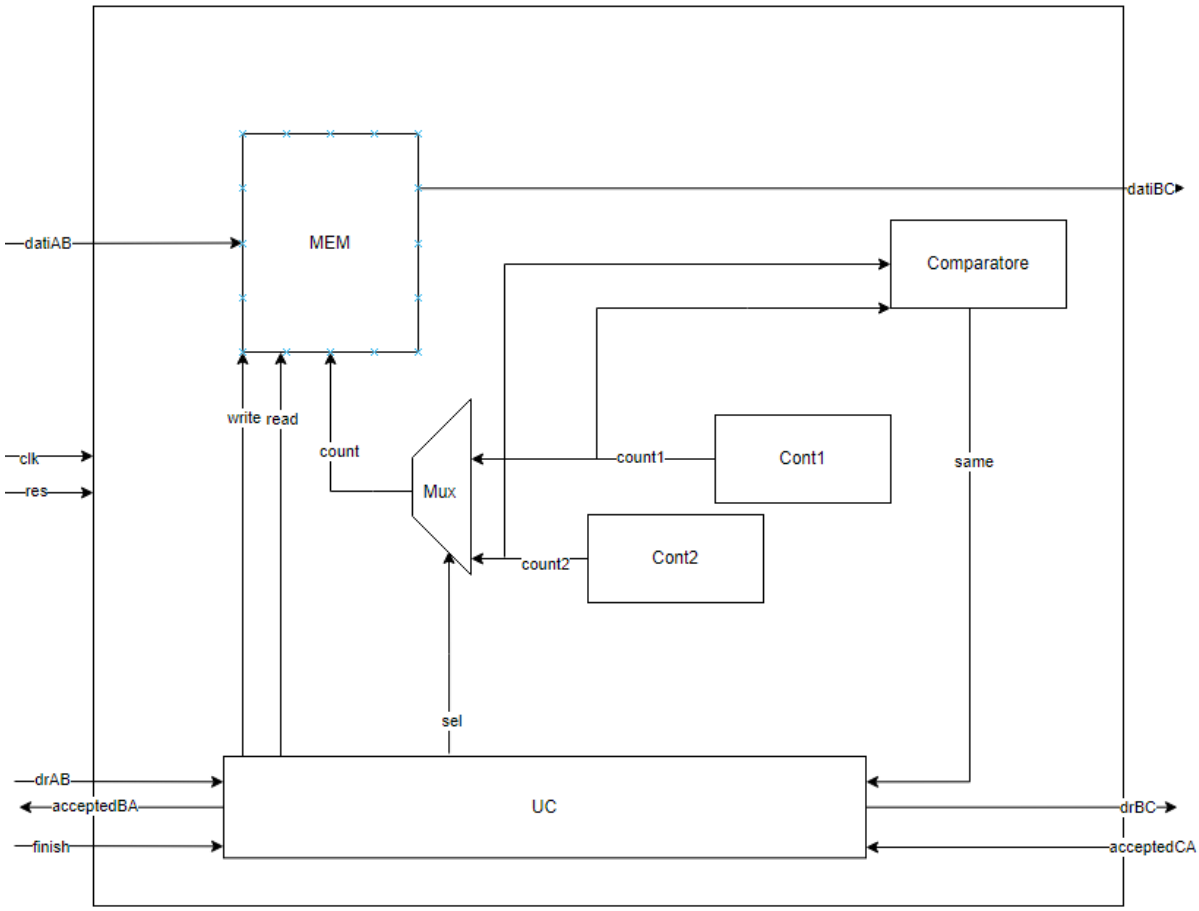


L'unità operativa è composta da tre elementi fondamentali: una ROM, un contatore modulo otto e un comparatore per determinare se il valore è strettamente maggiore di zero.
L'unità di controllo è stata implementata mediante logica cablata a partire dal seguente automa.

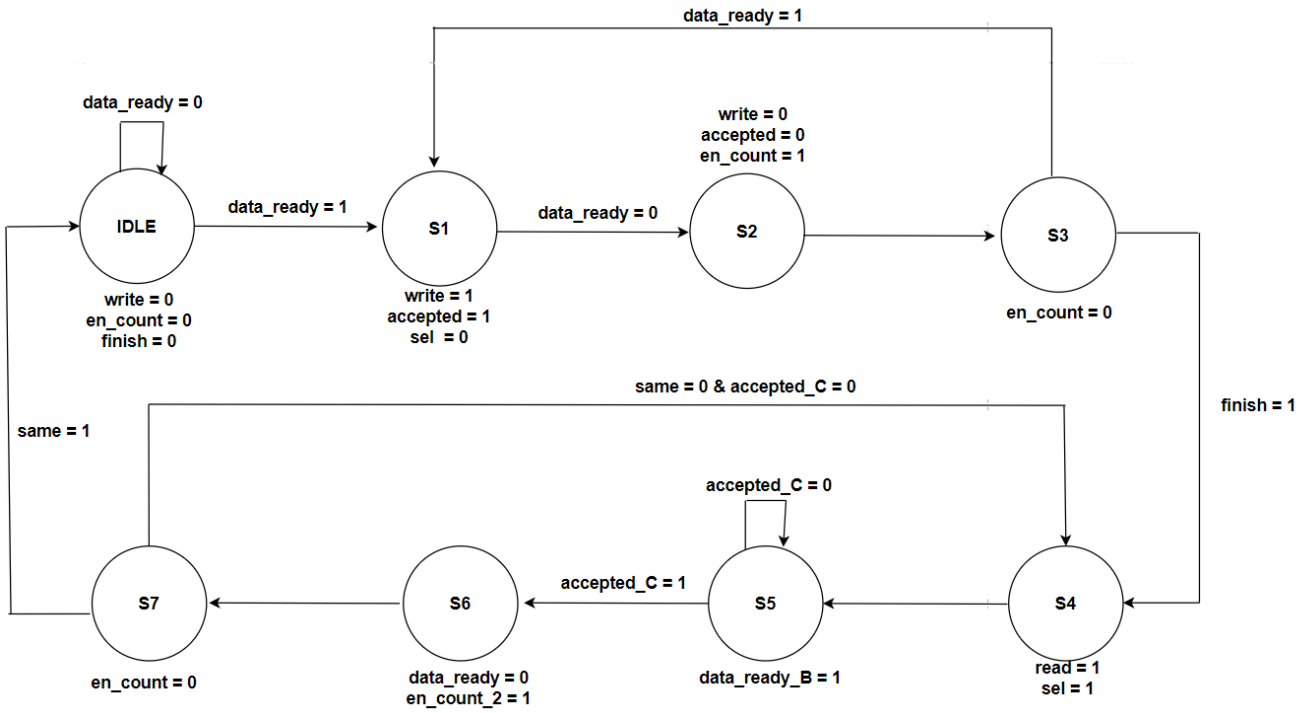


Il protocollo di handshake è realizzato tramite i segnali data_ready e accepted, oltre ad un segnale di finish per comunicare lo stato di terminazione all'unità B.

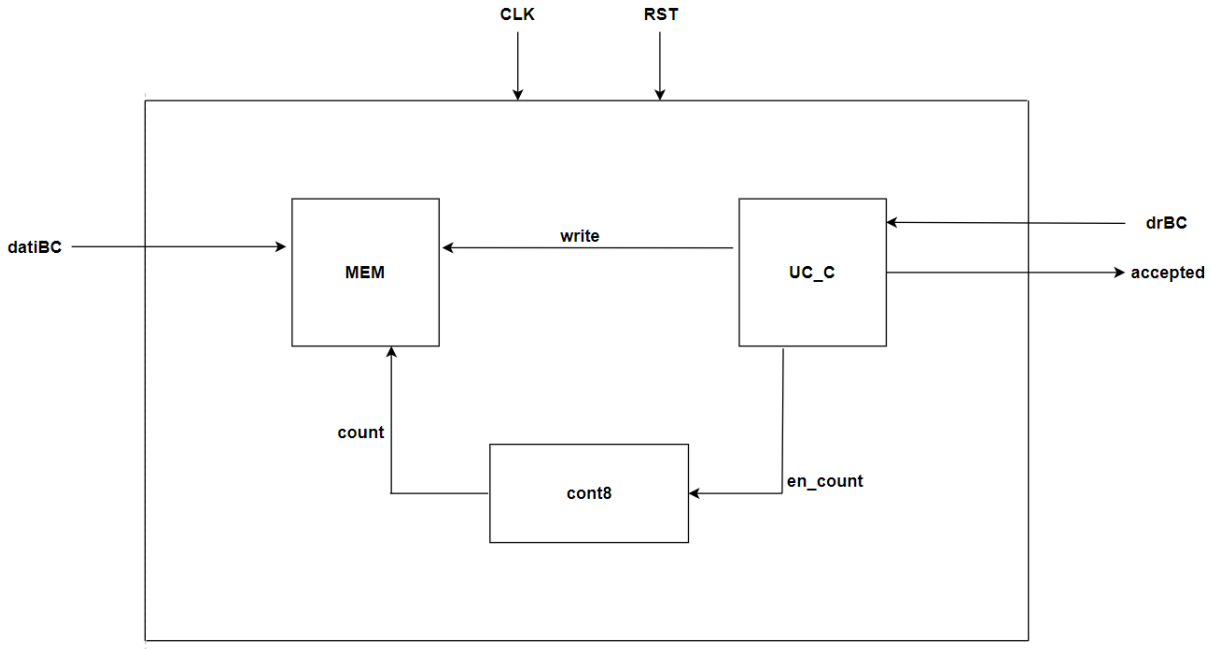
Unità B



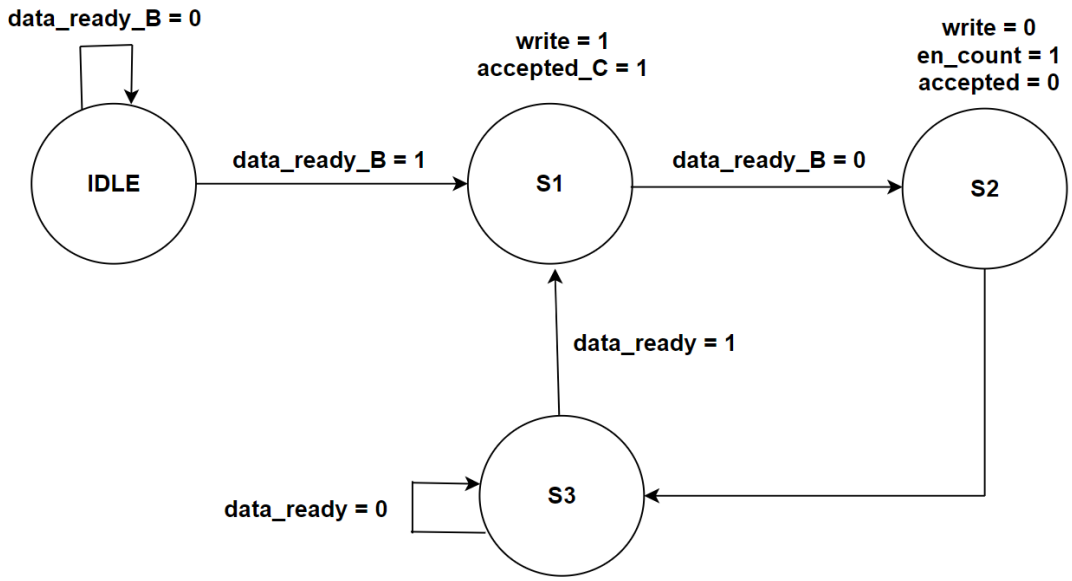
L’unità operativa leggermente più complessa in questa unità. Abbiamo una memoria, due contatori modulo otto, un multiplexer e un comparatore la cui uscita è alta quando il valore dei due contatori è lo stesso. L’unità di controllo è stata implementata mediante logica cablata a partire dal seguente automa.



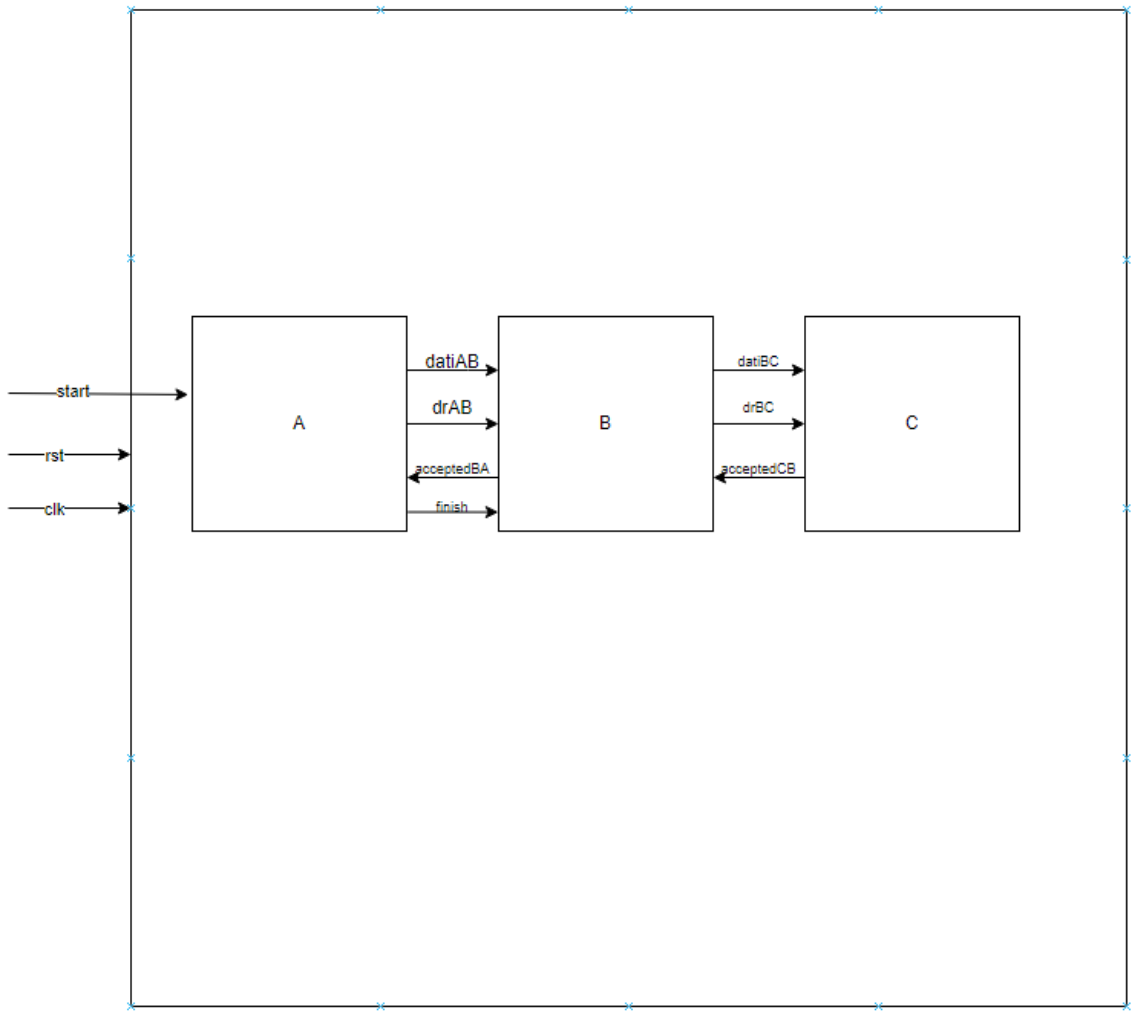
Unità C



L’unità operativa è composta solamente da una memoria e da un contatore modulo otto. L’unità di controllo invece implementa il seguente automa.



Queste tre unità sono state collegate all’interno di un top module.



12.3 Codice

Riportiamo le parti di codice più di interesse.

12.3.1 Unità A

Comparatore comportamentale per determinare se X>0:

```
entity Comparator0 is
    Port (
        X: in std_logic_vector(7 downto 0);
        Y: out std_logic --1 se X>0
     );
end Comparator0;

architecture Behavioral of Comparator0 is

begin

process(X)
begin
if(signed(X)>0) then
    Y<='1';
else
    Y<='0';
end if;
end process;

end Behavioral;
```

Unità di controllo:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Control_unitA is
    Port (
        clk, rst, start: in std_logic;
        accepted, pos, count_end: in std_logic;
        en_count, read, dr, finish: out std_logic
     );
end Control_unitA;

architecture automa of Control_unitA is

type state is(idle, S1, S2, S3, S4, endstate);
signal current_state: state := idle;
signal next_state: state;

begin

reg:process(clk)
begin
if (clk'event and clk='0') then
    if(rst='1') then
        current_state<=idle;
    else
    current_state<=next_state;
    end if;
end if;
end process;
```

```
comb: process(start, accepted, count_end, pos, current_state)
begin
    en_count<='0';
    read<='0';
    dr<='0';
    finish<='0';
    case current_state is
        when idle => if(start='1') then
                        next_state<=S1;
                     end if;
        when S1 =>   read<='1';
                     if(pos='1') then
                        next_state<=S2;
                     else
                        next_state<=S4;
                     end if;
        when S2 =>   dr<='1';
                     if(accepted='1') then
                        next_state<=S3;
                     end if;
        when S3 =>   en_count<='1';
                     if(count_end='1') then
                        next_state<=endstate;
                     else
                        next_state<=S1;
                     end if;
        when S4 =>   en_count<='1';
                     if(count_end='1') then
                        next_state<=endstate;
                     else
                        next_state<=S1;
                     end if;
        when endstate =>
                     finish<='1';
    end case;
end process;


end automa;
```

12.3.2 Unità B

Comparatore per determinare se cont1=cont22.

```
entity Comparatore is
    Port (
        in1, in2: in std_logic_vector(2 DOWNTO 0);
        y: out std_logic --1 se in1=in2
     );
end Comparatore;

architecture Behavioral of Comparatore is

begin

process(in1, in2)
begin
if(unsigned(in1)=unsigned(in2)) then
    y<='1';
else
    y<='0';
end if;
end process;
end Behavioral;
```

Unità di controllo B

```
entity Control_unitB is
   Port (
        clk,rst: in std_logic;
        drA, finish, same, acceptedC: in std_logic;
        sel, acceptedB, drB: out std_logic;
        en_count1, en_count2, read, write: out std_logic
    );
end Control_unitB;

architecture automa of Control_unitB is

type state is(idle, S1, S2, S3, S4, S5, S6, S7);
signal current_state: state := idle;
signal next_state: state;

begin

reg:process(clk)
begin
if (clk'event and clk='0') then
    if(rst='1') then
        current_state<=idle;
    else
    current_state<=next_state;
    end if;
end if;
end process;

comb: process(drA, finish, same, acceptedC, current_state)
begin
    en_count1<='0';
    en_count2<='0';
    read<='0';
    acceptedB<='0';
    write<='0';
    drB<='0';
```

```
    case current_state is
        when idle => sel<='0';
                     if(drA='1') then
                        next_state<=S1;
                     end if;
        when S1 =>   write<='1';
                     acceptedB<='1';
                     sel<='0';
                     if(drA='0') then
                        next_state<=S2;
                     end if;
        when S2 =>   en_count1<='1';
                     next_state<=S3;
        when S3 =>   en_count1<='0';
                     if(finish='1') then
                        next_state<=S4;
                     elsif (drA='1') then
                        next_state<=S1;
                     end if;
        when S4 =>   read<='1';
                     sel<='1';
                     next_state<=S5;
        when S5 =>   drB<='1';
                     if(acceptedC='1') then
                        next_state<=S6;
                     end if;
        when S6 =>   en_count2<='1';
                     next_state<=S7;
        when S7 =>   if(acceptedC='0' and same='0') then
                        next_state<=S4;
                     elsif(same='1') then
                        next_state<=idle;
                     end if;
    end case;
end process;

end automa;
```

12.3.3 Unità C

Unità di controllo C:

```
entity UC_C is
Port (
    clock, reset: in std_logic;
    data_ready_B: in std_logic;
    write: out std_logic;
    accepted: out std_logic;
    en_count: out std_logic
);
end UC_C;

architecture automa of UC_C is

type state is ( IDLE, S1, S2, S3 );
signal current_state, next_state: state;

begin

reg: process (clock)
begin
if (clock'event and clock='0') then
    if ( reset = '1' ) then
        current_state <= IDLE;
    else
        current_state <= next_state;
    end if;
end if;
end process;
```

```
comb: process(data_ready_B, current_state)
begin
case current_state is
    when IDLE =>
        write <= '0';
        accepted <= '0';
        en_count <= '0';
        if( data_ready_B = '1' ) then
        next_state <= S1;
    end if;

    when S1 =>
        write <= '1';
        accepted <= '1';
        if ( data_ready_B = '0' ) then
            next_state <= S2;
    end if;

    when S2 =>
        write <= '0';
        en_count <= '1';
        accepted <= '0';
        next_state <= S3;

    when S3 =>
        en_count <= '0';
        if ( data_ready_B = '1' ) then
            next_state <= S1;
        end if;
    end case;
end process;
end automa;
```

12.3.4 Top Module

```
entity TopModule is
    Port (
        clk, rst: in std_logic;
        start: in std_logic
     );
end TopModule;

architecture structural of TopModule is


 signal acceptedBA, drAB, finish: std_logic;
 signal datiAB, datiBC: std_logic_vector(7 downto 0);
 signal acceptedCB, drBC: std_logic;

 begin
 A: UnitA port map(clk, rst, start, acceptedBA, drAB, finish, datiAB);
 B: UnitB port map(clk, rst, datiAB, drAB, finish, acceptedCB, datiBC, acceptedBA, drBC);
 C: Unita_C port map(clk, rst, datiBC, drBC, acceptedCB);

 end structural;
```

12.4 Simulazione

Per poter effettuare la simulazione è prima di tutto necessario scrivere un testbench. Lo scheletro del testbench è stato ottenuto tramite uno strumento di generazione automatica.

La ROM di A è stata precaricata con i seguenti valori:

```
X"AA",
X"49",
X"11",
X"DD",
X"34",
X"02",
X"FE",
X"21");
```

Nella memoria di C ci aspettiamo i valori positivi 21, 02, 34, 11, 49.