

Tesina ASDI

Università degli Studi Federico II



UNIVERSITÀ DEGLI STUDI DI NAPOLI

FEDERICO II

Vincenzo Merola, Raffaele Russo, Alessandro Vanacore

Indice

Multiplexer	5
Parte I	5
Mux 2:1	5
Multiplexer 4:1	7
Multiplexer 16:1	9
Parte II	11
Demultiplexer 1:4	11
Rete di interconnessione	12
Parte III	15
Decoder 4:16	15
Encoder BCD	21
Parte I	21
Encoder	21
Arbitro	23
Encoder prioritario	24
Parte II	27
Sintesi	27
Parte III	28
Gestore display	28
Encoder complessivo	30
Riconoscitore di Sequenze	33
Parte I	33
Automa	33
Parte II	39
Debouncer	39
Riconoscitore su board	40

Sintesi	41
Shift Register	42
Descrizione comportamentale	42
Shift register	42
Descrizione strutturale	44
Flip-flop D	44
Shift register	45
Contatore	49
Cronometro	53
Parte I	53
Flip-flop T	53
Contatori	54
Cronometro	61
Parte II	65
Divisore di frequenza	65
Gestore display a sette segmenti	67
Anodes manager	69
Cathodes manager	70
Codificatore	73
Controllore	75
Cronometro su board	76
Sintesi	76
Parte III	78
Contatore	78
RAM	79
Cronometro con intertempi	80
Sistema di Testing	82
Parte I	82
ROM	82
Macchina combinatoria	83
RAM	84
Contatore	86
Clock filter	87
Sistema di testing	89
Parte II	93

Sintesi	93
Comunicazione con handshaking	94
Parte I	94
Handshaking	94
Schema complessivo	95
UC trasmittitore	95
UC ricevitore	98
Processore	101
Analisi architettura	101
Introduzione	101
Data Path	101
Tempificazione del Data Path	102
Introduzione	106
Microistruzioni	107
Unità di controllo	110
Analisi istruzioni	114
IADD	114
DUP	116
Modifica istruzione ISUB	119
UART	121
Parte I	121
Sistema complessivo	121
Parte II	124
Sistema A	124
Sistema B	128
Sistema Complessivo	131
Switch Multistadio	134
Parte I	134
Unità operativa	134
Routing node	135
Unità di controllo	139
Arbitro	141
Arbitro	141
Omega Network	143

Divisore non restoring	146
Parte I	146
Divisore	146
Shift register AQ	147
Registro divisore	149
Flip flop D	149
ADD/SUB	150
RCA	151
FA	152
Unità di controllo	153
Blowfish algorithm	160
Parte I	160
Implementazione python	162
Architettura complessiva	164
Round	165
F block	166
PlainRegister	169
Registro chiave	171
Post processing unit	171
Post processing register	172
Contatore	173
Unità di controllo	174

Multiplexer

Parte I

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.

Mux 2:1

L'unità base è il multiplexer 2:1. Per quest'ultimo utilizziamo una descrizione **dataflow**, mentre per gli altri una descrizione **strutturale**. La tabella di verità 1 e lo schema in figura 1 ne mostrano il funzionamento.

s	y
0	a_0
1	a_1

Tabella 1

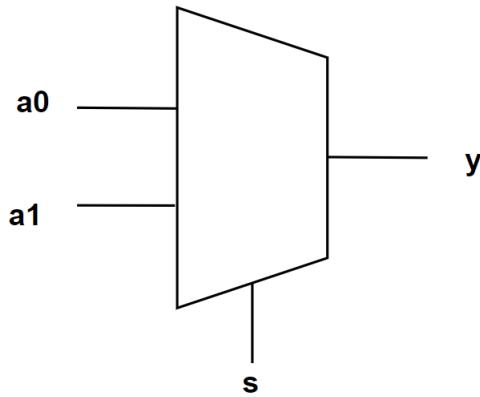


Figura 1: Schema MUX 2:1

Entity e architecture - Mux 2:1

```
entity mux_2_1 is
    port ( s : in STD_LOGIC;
           a0, a1 : in STD_LOGIC;
           y : out STD_LOGIC);
end mux_2_1;

architecture Dataflow of mux_2_1 is
begin
    with s select
        y <= a0 when '0',
        a1 when '1',
        '-' when others;
end Dataflow;
```

Simulazione

Essendo il numero degli ingressi pari a 3, è stato possibile valutare tutte le $2^3 = 8$ permutazioni di essi, ottenendo una coverage del 100%. In figura 2 è mostrato l'esito della simulazione.



Figura 2: Simulazione MUX 2:1

Multiplexer 4:1

Per il multiplexer 4:1 utilizziamo tre multiplexer 2:1 disposti come in figura 3.

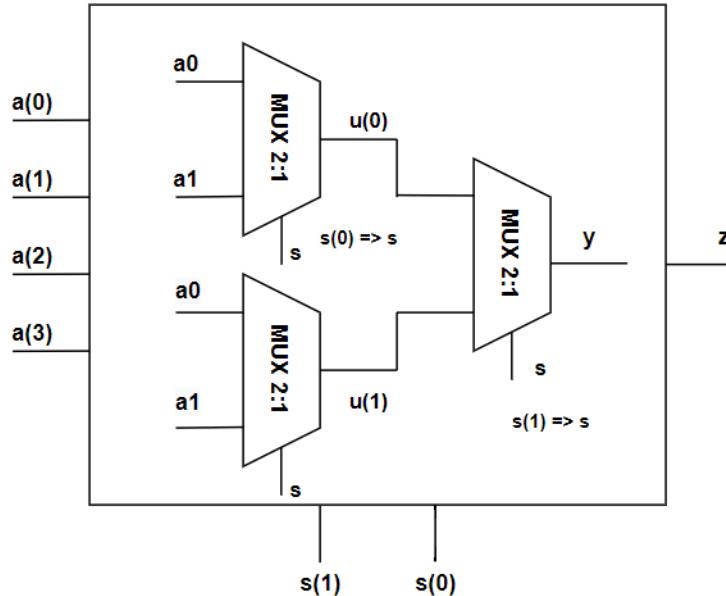


Figura 3: Schema MUX 4:1

Entity e architecture - mux 4:1

```
entity mux_4_1 is
    port ( a : in std_logic_vector(0 to 3);
           s : in std_logic_vector(1 downto 0);
           z : out std_logic);
end mux_4_1;
```

```

architecture Structural of mux_4_1 is
  component mux_2_1 is
    port (s : in std_logic;
          a0, a1 : in std_logic;
          y : out std_logic);
  end component;

  signal u : std_logic_vector(0 to 1);

begin
  mux_0_1: for i in 0 to 1 generate m : mux_2_1
    port map ( a0 => a(i*2),
               a1 => a(i*2 + 1),
               s => s(0),
               y => u(i));
  end generate;

  mux_2: mux_2_1
    port map ( a0 => u(0),
               a1 => u(1),
               s => s(1),
               y => z);
end Structural;

```

Simulazione

In figura 4 è mostrata la simulazione del mux 4:1

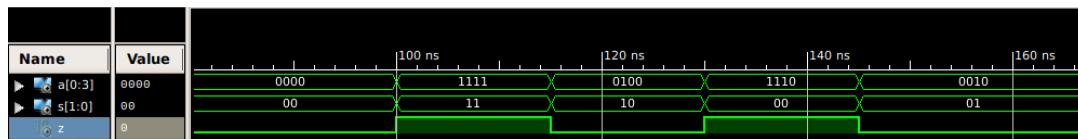


Figura 4: Simulazione mux 4:1

Multiplexer 16:1

Infine si riporta la realizzazione del multiplexer 16:1 con approccio strutturale, in figura 5.

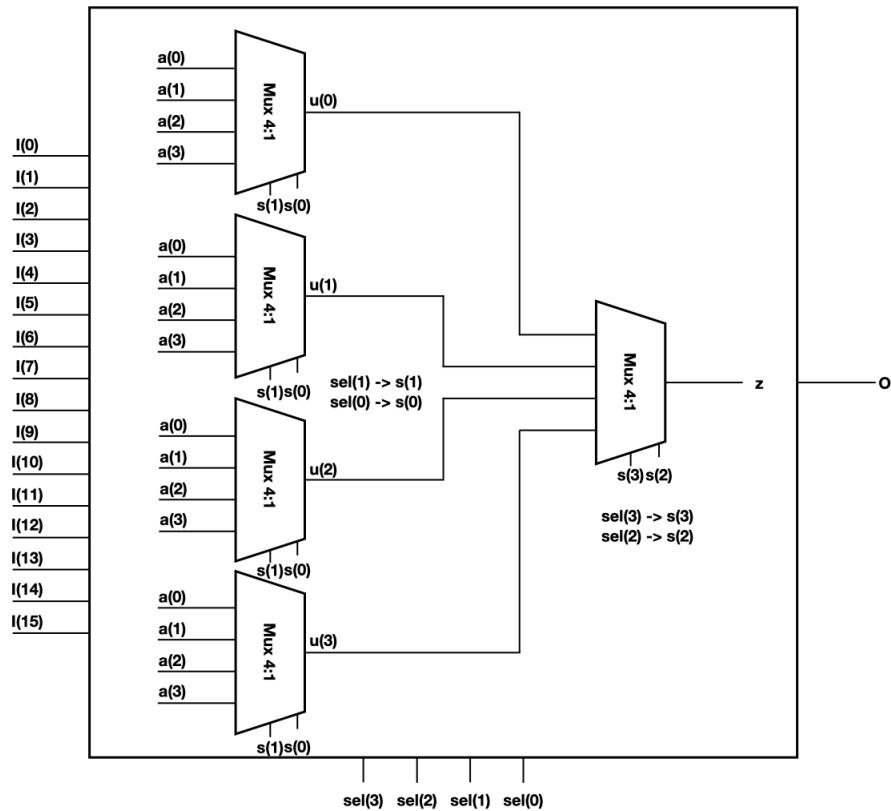


Figura 5: Schema MUX 16:1

Entity e architecture - mux 16:1

```
entity mux16 is
  port ( I : in STD_LOGIC_VECTOR(0 to 15);
        sel : in STD_LOGIC_VECTOR(3 downto 0);
        O : out STD_LOGIC);
end mux16;
```

```

architecture structural of mux16 is

signal u : STD_LOGIC_VECTOR(0 to 3) := (others => '0');

COMPONENT mux4
PORT ( a : in STD_LOGIC_VECTOR(0 to 3);
s : in STD_LOGIC_VECTOR(1 downto 0);
z : out STD_LOGIC);
END COMPONENT;

BEGIN
    mux0_3: for j in 0 to 3 generate
        a : mux4
            port map ( a(0 to 3) => I(j*4 to j*4+3),
            s(1 downto 0) => sel(1 downto 0),
            z => u(j));
    end generate;

    mux_4: mux4
        Port map ( a(0 to 3) => u,
        s(1 downto 0) => sel(3 downto 2),
        z => 0);
end structural;

```

Simulazione

Essendo il numero degli ingressi pari a 20, non è stato possibile valutare le 2^{20} permutazioni di essi. Si è optato per testare soltanto alcune combinazioni significative degli ingressi, come si evince dalla figura 6.

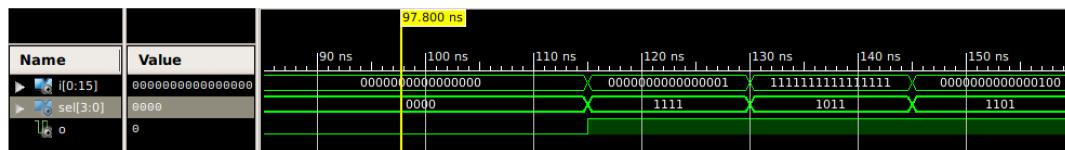


Figura 6: Simulazione MUX 16:1

Parte II

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.

La rete di interconnessione può essere descritta strutturalmente, utilizzando il mux 16:1 dell'esercizio precedente e il demultiplexer 1:4 in figura 7.

Demultiplexer 1:4

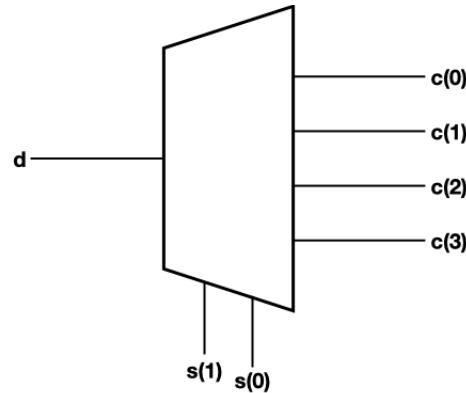


Figura 7: Schema DEMUX 1:4

Entity e architecture - demux 1:4

```
entity DEMUX_1_4 is
  port ( d : in STD_LOGIC;
         s : in STD_LOGIC_VECTOR(1 downto 0);
         c : out STD_LOGIC_VECTOR(0 to 3));
end DEMUX_1_4;

architecture Dataflow of DEMUX_1_4 is
begin
  c(0) <= d when (s = "00") else
    '1';
  c(1) <= d when (s = "01") else
```

```

'-' ;
c(2) <= d when (s = "10") else
'-' ;
c(3) <= d when (s = "11") else
'-' ;
end Dataflow;

```

Simulazione

Si è provveduto anche a testare quasi tutte le combinazioni di input possibili, come si può evincere dalla simulazione in figura 8.

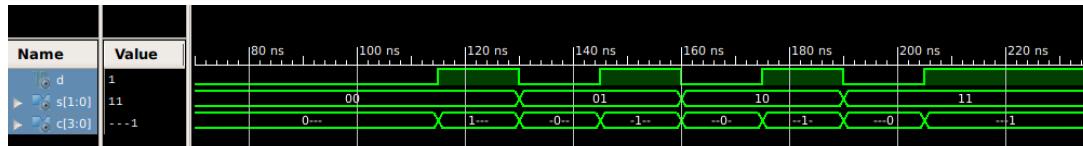


Figura 8: Simulazione DEMUX 1:4

Rete di interconnessione

In figura 9 è mostrata l'architettura complessiva della nostra rete di interconnessione a 16 sorgenti e 4 destinazioni.

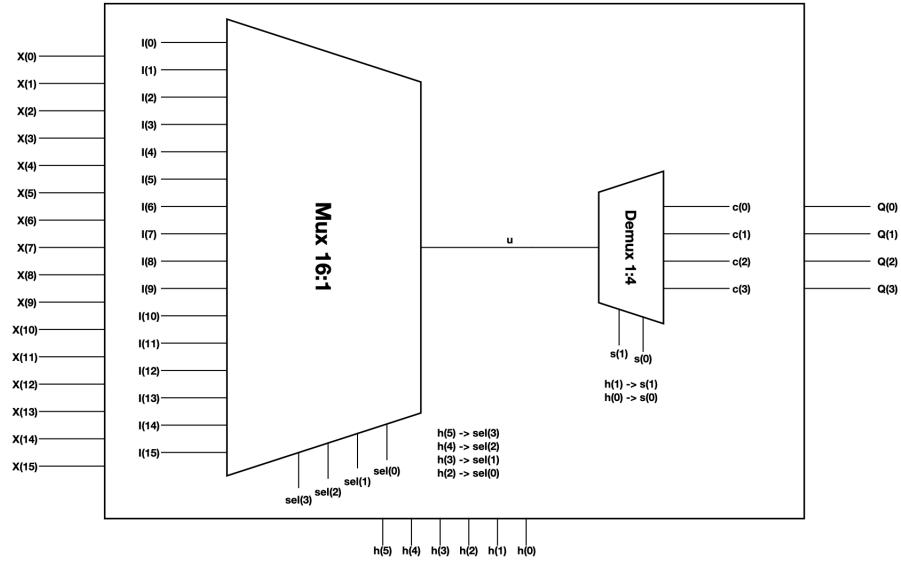


Figura 9: Rete di interconnessione 16:4

Entity e architecture - rete di interconnessione

```

entity rete_intercon is
  port ( X : in STD_LOGIC_VECTOR(0 to 15);
         h : in STD_LOGIC_VECTOR(5 downto 0);
         Q : out STD_LOGIC_VECTOR(3 downto 0));
end rete_intercon;

architecture structural of rete_intercon is
  signal u : STD_LOGIC;
  COMPONENT mux16
    PORT ( I : in STD_LOGIC_VECTOR(0 to 15);
           sel : in STD_LOGIC_VECTOR(3 downto 0);
           O : out STD_LOGIC);
  END COMPONENT;

  COMPONENT DEMUX_1_4 is
    PORT ( d : in STD_LOGIC;
           s : in STD_LOGIC_VECTOR(1 downto 0);
           c : out STD_LOGIC_VECTOR(0 to 3));
  END COMPONENT;

```

```

begin
    mux_0: mux16
        Port map ( I => X,
                    sel => h(5 downto 2),
                    0 => u);

    demux_0: DEMUX_1_4
        Port map ( d => u,
                    s => h(1 downto 0),
                    c => Q);
end structural;

```

Simulazione

In figura 10 è possibile osservare il comportamento del sistema tramite la simulazione effettuata.

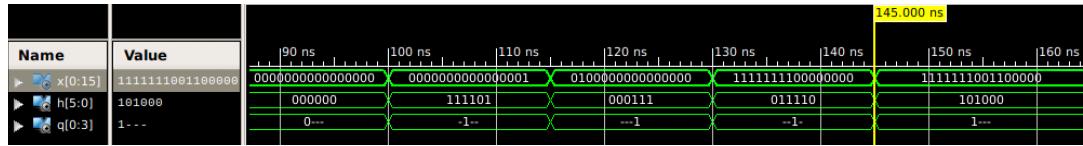


Figura 10: Simulazione rete di Interconnessione 16:4

Parte III

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

Decoder 4:16

Essendo il numero di ingressi maggiore del numero di switch disponibili sulla board, è necessario utilizzare un decodificatore come in figura 11 per trasformare il codice di ingresso nell'uscita prevista.

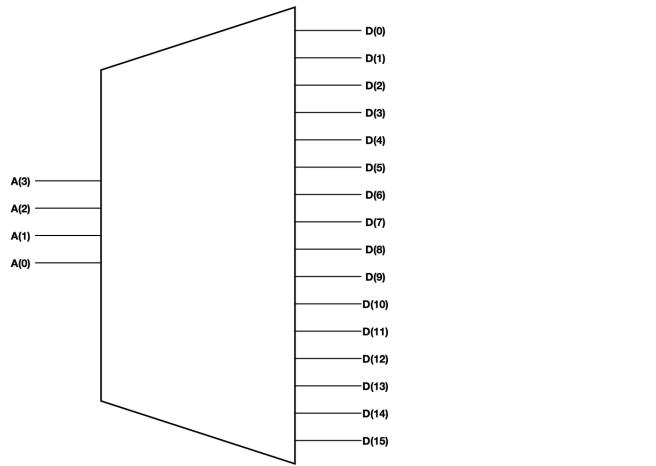


Figura 11: Decoder 4:16

Entity e architecture - decoder 4:16

```
entity decoder_4_16 is
  PORT ( A : in STD_LOGIC_VECTOR(3 downto 0);
         D : out STD_LOGIC_VECTOR(0 to 15));
end decoder_4_16;
```

```

architecture Dataflow of decoder_4_16 is

begin
    D <= "1000000000000000" when (A = "0000") else
        "0100000000000000" when (A = "0001") else
        "0010000000000000" when (A = "0010") else
        "0001000000000000" when (A = "0011") else
        "0000100000000000" when (A = "0100") else
        "0000010000000000" when (A = "0101") else
        "0000001000000000" when (A = "0110") else
        "0000000100000000" when (A = "0111") else
        "0000000010000000" when (A = "1000") else
        "0000000001000000" when (A = "1001") else
        "0000000000100000" when (A = "1010") else
        "0000000000010000" when (A = "1011") else
        "0000000000001000" when (A = "1100") else
        "0000000000000100" when (A = "1101") else
        "0000000000000010" when (A = "1110") else
        "0000000000000001" when (A = "1111") else
        "-----";
end Dataflow;

```

Simulazione

In figura 12 è possibile visualizzare alcuni esempi di funzionamento del decodificatore progettato.

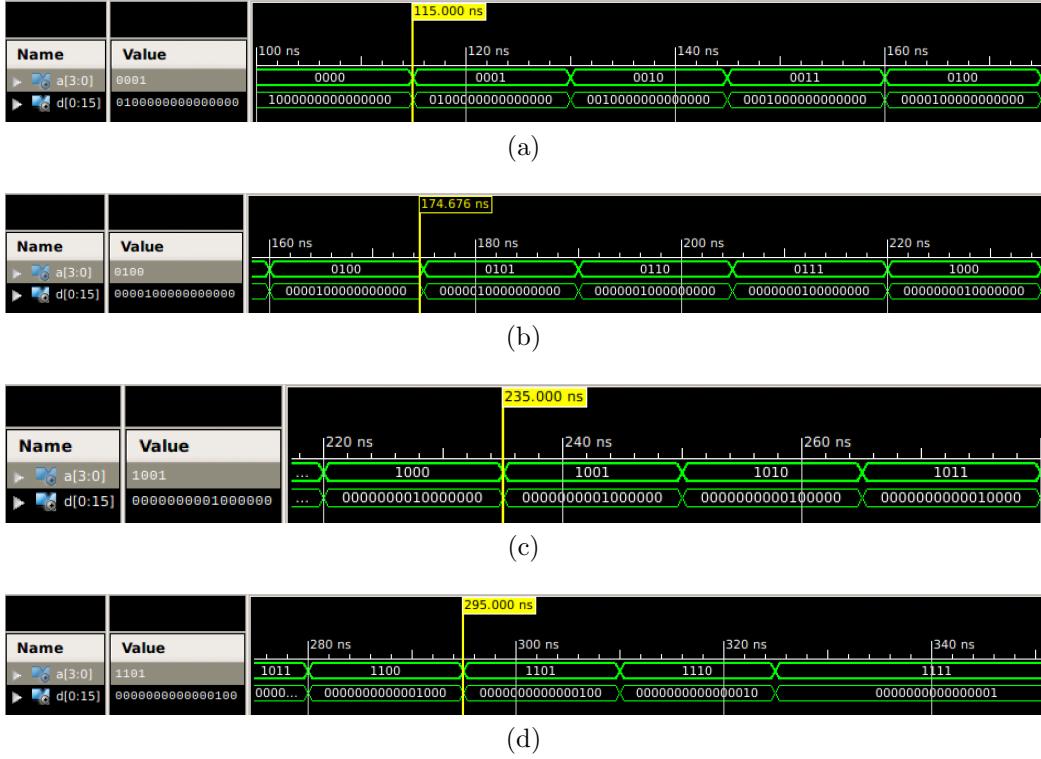


Figura 12: Simulazione decoder

In figura 13 è illustrato lo schematico del progetto complessivo.

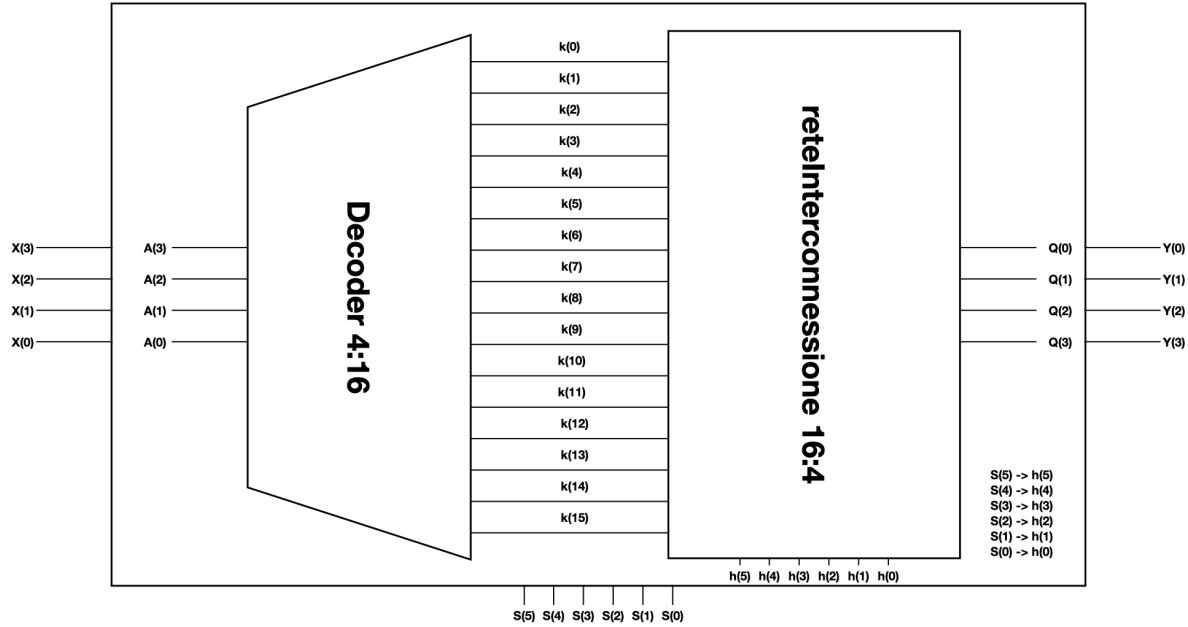


Figura 13: Rete di interconnessione con decoder

Entity e architecture - rete con decoder

```

entity rete_decoder is
  PORT ( X : IN STD_LOGIC_VECTOR(3 downto 0);
         S : IN STD_LOGIC_VECTOR(5 downto 0);
         Y : OUT STD_LOGIC_VECTOR(0 to 3));
end rete_decoder;

architecture Behavioral of rete_decoder is

  signal k : STD_LOGIC_VECTOR(0 to 15);

  COMPONENT rete_intercon
    PORT ( X : in STD_LOGIC_VECTOR(0 to 15);
           h : in STD_LOGIC_VECTOR(5 downto 0);
           Q : out STD_LOGIC_VECTOR(3 downto 0));
  END COMPONENT;

```

```

COMPONENT decoder_4_16
    PORT ( I : in STD_LOGIC_VECTOR(3 downto 0);
            C : out STD_LOGIC_VECTOR(0 to 15));
    END COMPONENT;

begin
    rete_0 : rete_intercon
        Port map ( X => k,
                    h => S,
                    Q => Y);

decoder_0: decoder_4_16
    Port map ( I => X,
                C => k);
end Behavioral;

```

Simulazione

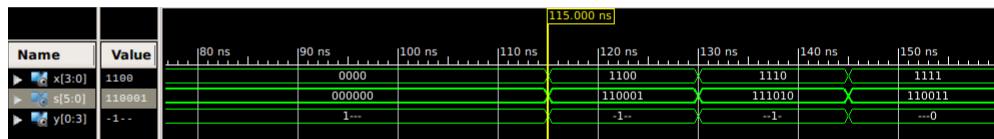


Figura 14: Simulazione rete di interconnessione con decoder

Sintesi

Per mancanza di switch abbiamo utilizzato anche 2 pulsanti per l'ingresso. In figura 15 sono mostrati 4 esempi di sintesi su board, nei quali, alzando opportunamente switch e bottoni relativi agli ingressi e le selezioni, si visualizza la destinazione selezionata tramite led.



Figura 15: Sintesi su board

Encoder BCD

Parte I

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

Encoder

L'entità cardine di questo progetto è l'encoder o codificatore. Come mostrato in figura 16 il componente è in grado di ricevere in ingresso un stringa di 10 bit fornendo la sua codifica BCD su 4 bit in uscita.

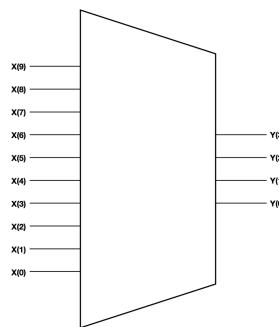


Figura 16: Schema encoder 10:4

Entity e architecture - encoder

```

entity encoder_10_4 is
    port(X : in STD_LOGIC_VECTOR(9 downto 0);
         Y : out STD_LOGIC_VECTOR(3 downto 0));
end encoder_10_4;

architecture Dataflow of encoder_10_4 is
begin
with X select
    Y <= "0000" when "0000000001",
              "0001" when "0000000010",
              "0010" when "0000000100",
              "0011" when "0000001000",
              "0100" when "0000010000",
              "0101" when "0000100000",
              "0110" when "0001000000",
              "0111" when "0010000000",
              "1000" when "0100000000",
              "1001" when "1000000000",
              "----" when others;
end Dataflow;

```

Simulazione

In figura 17 è mostrata la simulazione al 100% di coverage dell'encoder BCD.

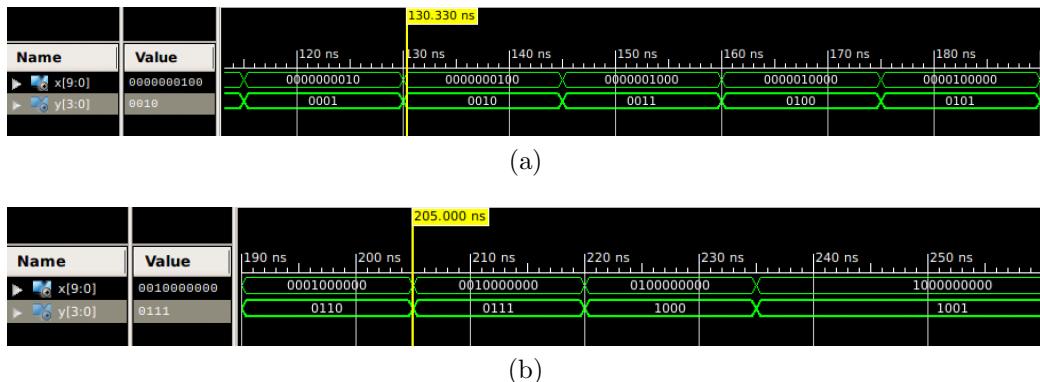


Figura 17: Simulazione encoder

Arbitro

Per gestire l'eventualità di più ingressi alti in ingresso all'encoder, è stato anteposto a esso un arbitro. Quest'ultimo, la cui architettura è mostrata in figura 18, a fronte di più ingressi alti simultanei, fornisce la sola uscita corrispondente all'ingresso alto più significativo.

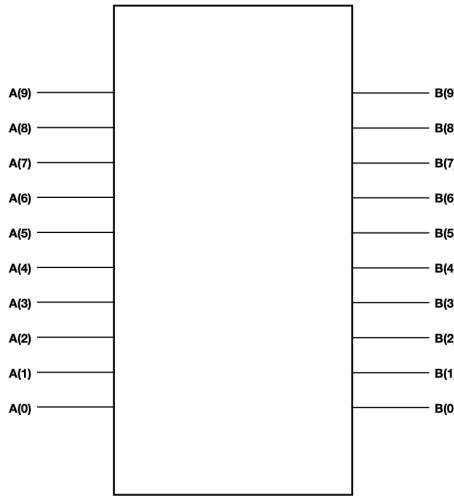


Figura 18: Schema arbitro

Entity e architecture - arbitro

```
entity Arbitro is
  port(A : in STD_LOGIC_VECTOR(9 downto 0);
       B : out STD_LOGIC_VECTOR(9 downto 0));
end Arbitro;

architecture Dataflow of Arbitro is

begin

  B <= "1000000000" when A(9) = '1' else
    "0100000000" when A(8) = '1' else
    "0010000000" when A(7) = '1' else
    "0001000000" when A(6) = '1' else
    "0000100000" when A(5) = '1' else
```

```

    "0000010000" when A(4) = '1' else
    "0000001000" when A(3) = '1' else
    "0000000100" when A(2) = '1' else
    "0000000010" when A(1) = '1' else
    "0000000001" when A(0) = '1' else
    "-----";
end Dataflow;

```

Simulazione

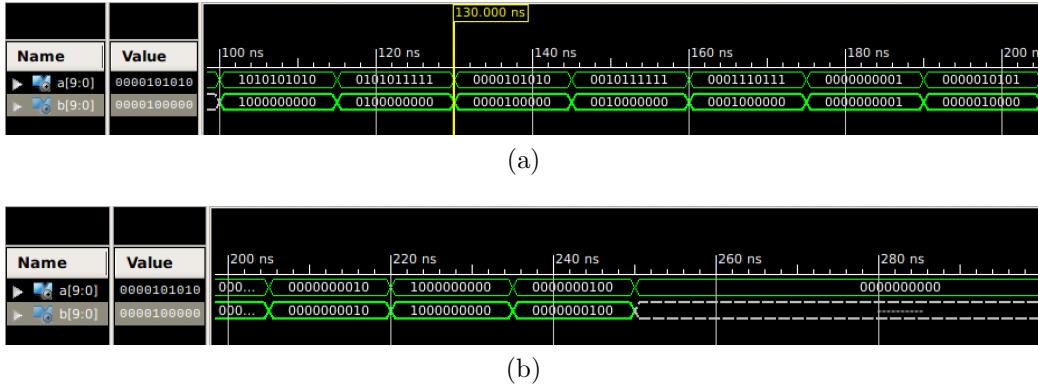


Figura 19: Simulazione Arbitro

Per la simulazione, come possibile osservare dalla figura 19, sono state selezionate 11 configurazioni in ingresso, a fronte delle 2^{10} combinazioni possibili.

Encoder prioritario

Il risultato complessivo è illustrato in figura 20.

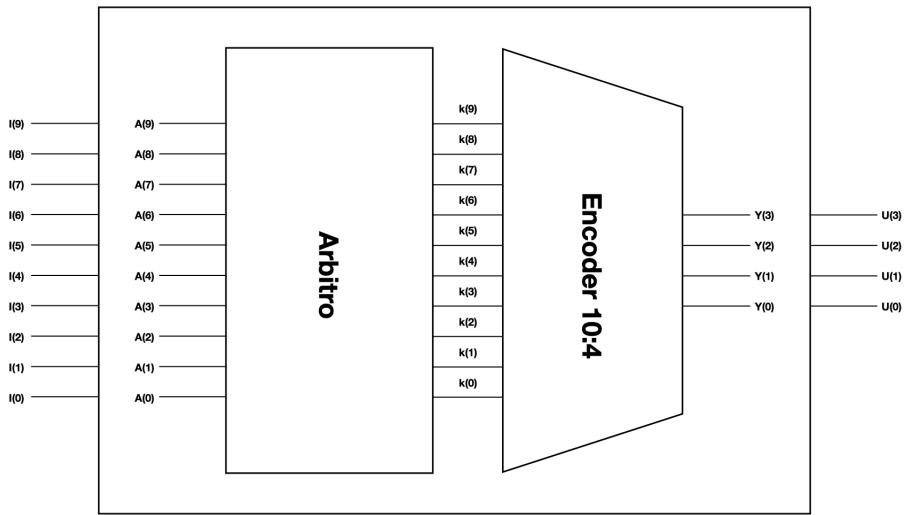


Figura 20: Schema encoder prioritario

Entity e architecture - encoder prioritario

```

entity Encoder_prioritario is
  port ( I : in STD_LOGIC_VECTOR(9 downto 0);
         U : out STD_LOGIC_VECTOR(3 downto 0));
end Encoder_prioritario;

architecture Structural of Encoder_prioritario is
  COMPONENT Arbitro IS
    Port ( A : in STD_LOGIC_VECTOR(9 downto 0);
           B : out STD_LOGIC_VECTOR(9 downto 0));
  END COMPONENT;

  COMPONENT encoder_10_4 IS
    Port ( X : in STD_LOGIC_VECTOR(9 downto 0);
           Y : out STD_LOGIC_VECTOR(3 downto 0));
  END COMPONENT;

  signal k : STD_LOGIC_VECTOR(9 downto 0);

```

```

begin

    A : Arbitro
        PORT MAP ( A => I,
                    B => k);

    E : encoder_10_4
        PORT MAP ( X => k,
                    Y => U);

end Structural;

```

Simulazione

In figura 21 è mostrata la simulazione del sistema complessivo sfruttando il comportamento dell'arbitro per la gestione delle priorità.

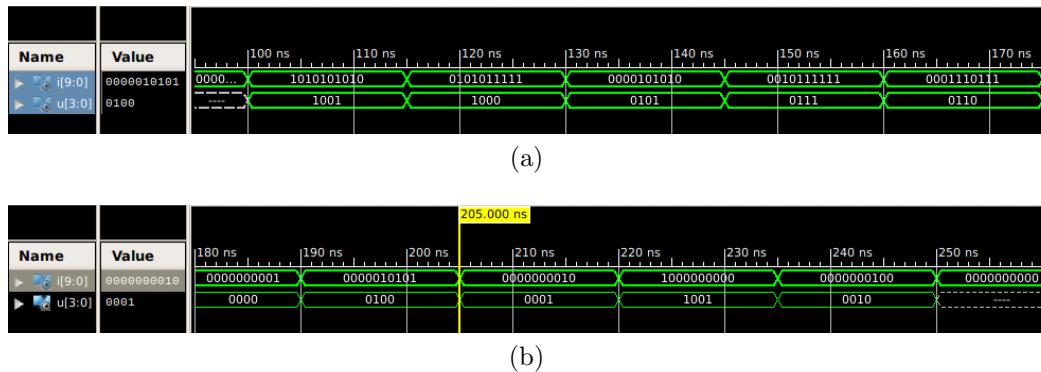


Figura 21: Simulazione encoder prioritario

Parte II

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

Sintesi

Per sintetizzare sulla board lo schema complessivo in figura 20, avendo a disposizione una board con 8 switch, riduciamo a 8 il numero di bit di X in ingresso e a 3 il numero di bit dell'uscita Y. Procediamo, dunque, mappando rispettivamente gli ingressi e le uscite con gli switch e con i led della scheda, come riportato in figura 22.

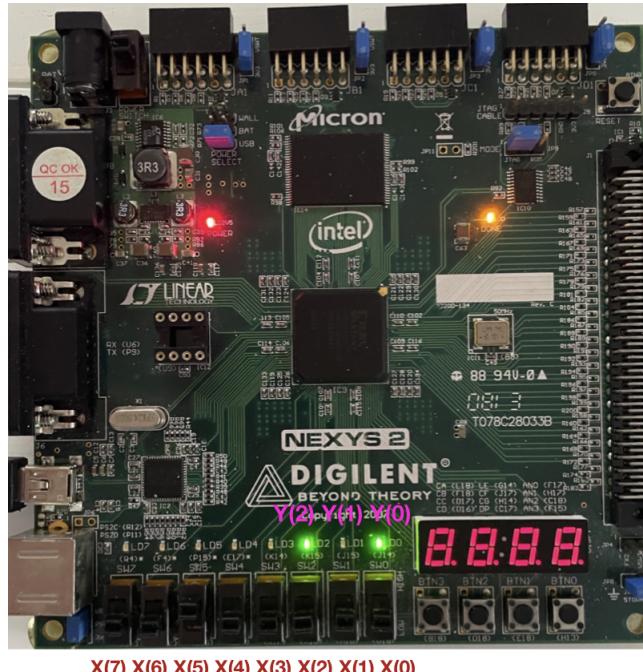


Figura 22: Sintesi BCD

Parte III

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

La differenza con l'esercizio precedente sta nel fatto che l'uscita dell'encoder non è mappata direttamente con l'uscita dello schema complessivo, ma va a pilotare i catodi del componente "seven segment display".

Gestore display

La progettazione di quest'ultimo è stata semplificata, potendo visualizzare la rappresentazione BCD delle cifre decimali su un unico display. Infatti, i meccanismi di refresh dei display sono in questo caso trascurabili; la gestione degli anodi è superflua, potendo abilitare all'atto dell'inizializzazione solo l'anodo (display) da visualizzare.

Entity e architecture - gestore display

```
entity display_seven_segments is
    Port ( VALUE : in  STD_LOGIC_VECTOR (3 downto 0);
           ENABLE : in  STD_LOGIC_VECTOR (3 downto 0);
           DOTS : in  STD_LOGIC_VECTOR (5 downto 0);
           ANODES : out  STD_LOGIC_VECTOR (3 downto 0);
           CATHODES : out  STD_LOGIC_VECTOR (7 downto 0));
end display_seven_segments;

architecture Structural of display_seven_segments is

COMPONENT cathodes_manager
    PORT ( value : IN std_logic_vector(3 downto 0);
           dots : IN std_logic_vector(5 downto 0);
           cathodes : OUT std_logic_vector(7 downto 0));
END COMPONENT;

begin

    cathodes_instance: cathodes_manager
```

```

port map ( value => value,
dots => dots,
cathodes => cathodes);

ANODES <= NOT ENABLE;
end Structural;

entity cathodes_manager is
Port(value : in STD_LOGIC_VECTOR (3 downto 0);
dots : in STD_LOGIC_VECTOR (5 downto 0);
cathodes : out STD_LOGIC_VECTOR (7 downto 0));
end cathodes_manager;

architecture Behavioral of cathodes_manager is

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

alias digit_0 is value (3 downto 0);

signal cathodes_for_digit : std_logic_vector(6 downto 0) := (others => '0');
signal dot :std_logic := '0';

```

```

begin
    dot <= dots(0);

    nomeproc : process(value)

begin
case value is
    when "0000" => cathodes_for_digit <= zero;
    when "0001" => cathodes_for_digit <= one;
    when "0010" => cathodes_for_digit <= two;
    when "0011" => cathodes_for_digit <= three;
    when "0100" => cathodes_for_digit <= four;
    when "0101" => cathodes_for_digit <= five;
    when "0110" => cathodes_for_digit <= six;
    when "0111" => cathodes_for_digit <= seven;
    when "1000" => cathodes_for_digit <= eight;
    when "1001" => cathodes_for_digit <= nine;
    when "1010" => cathodes_for_digit <= a;
    when "1011" => cathodes_for_digit <= b;
    when "1100" => cathodes_for_digit <= c;
    when "1101" => cathodes_for_digit <= d;
    when "1110" => cathodes_for_digit <= e;
    when "1111" => cathodes_for_digit <= f;
    when others => cathodes_for_digit <= (others => '0');
end case;
end process;

cathodes <= (not dot) & cathodes_for_digit;

end Behavioral;

```

Encoder complessivo

Entity e architecture - encoder complessivo

```

entity encoder_board is
port ( X : in STD_LOGIC_VECTOR(7 downto 0);

```

```

anodes_out : out STD_LOGIC_VECTOR (3 downto 0);
cathodes_out : out STD_LOGIC_VECTOR (7 downto 0));
end encoder_board;

architecture Behavioral of encoder_board is

signal u : std_logic_vector(3 downto 0);

component PriorityEncoder
    Port ( X : in STD_LOGIC_VECTOR(7 downto 0);
           Y : out STD_LOGIC_VECTOR(3 downto 0));
end component;

component display_seven_segments is
    Port ( VALUE : in STD_LOGIC_VECTOR (3 downto 0);
           ENABLE : in STD_LOGIC_VECTOR (3 downto 0);
           DOTS : in STD_LOGIC_VECTOR (5 downto 0);
           ANODES : out STD_LOGIC_VECTOR (3 downto 0);
           CATHODES : out STD_LOGIC_VECTOR (7 downto 0));
end component;

begin

seven_segment_array: display_seven_segments

PORT MAP(value => u,
enable => "0001",
dots => "000000",
anodes => anodes_out,
cathodes => cathodes_out);

PriorityEncoder_0 : PriorityEncoder
PORT MAP ( X => X,
Y => u);
end Behavioral;

```

Sintesi

In figura 23 è raffigurato un esempio di sintesi su board del sistema complesso con visualizzazione tramite display.

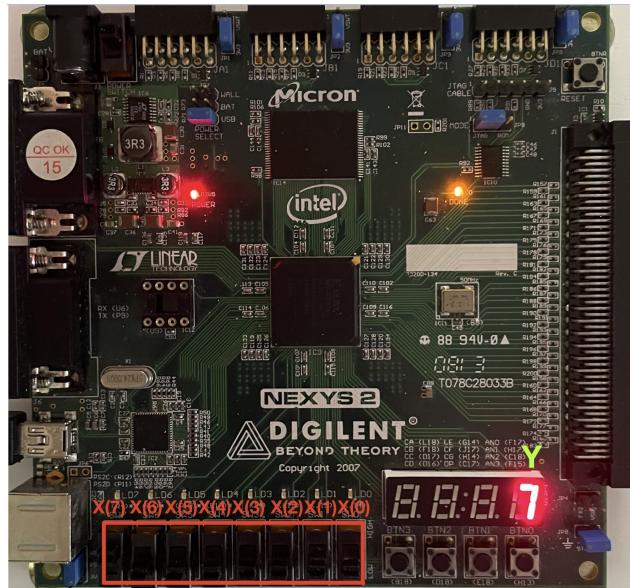


Figura 23: Sintesi BCD display

Riconoscitore di sequenza

Parte I

Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza 1001. La macchina prende in ingresso un segnale binario i che rappresenta il dato, un segnale A di temporizzazione e un segnale M di modo, che ne disciplina il funzionamento, e fornisce un'uscita Y alta quando la sequenza viene riconosciuta. In particolare, se $M=0$, la macchina valuta i bit seriali in ingresso a gruppi di 4, se $M=1$, la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta.

Automa

L'automa in figura 24 descrive il comportamento della macchina. Il cambio della modalità di funzionamento è valutato al reset. All'accensione la macchina rimane nello stato di attesa Q_M . Quando il segnale di reset si alza la macchina inizia a evolvere a seconda del valore corrente del bit di modo M . L'eventuale discesa e risalita del reset durante il funzionamento della macchina comporta il ritorno allo stato di attesa e l'evoluzione riparte.

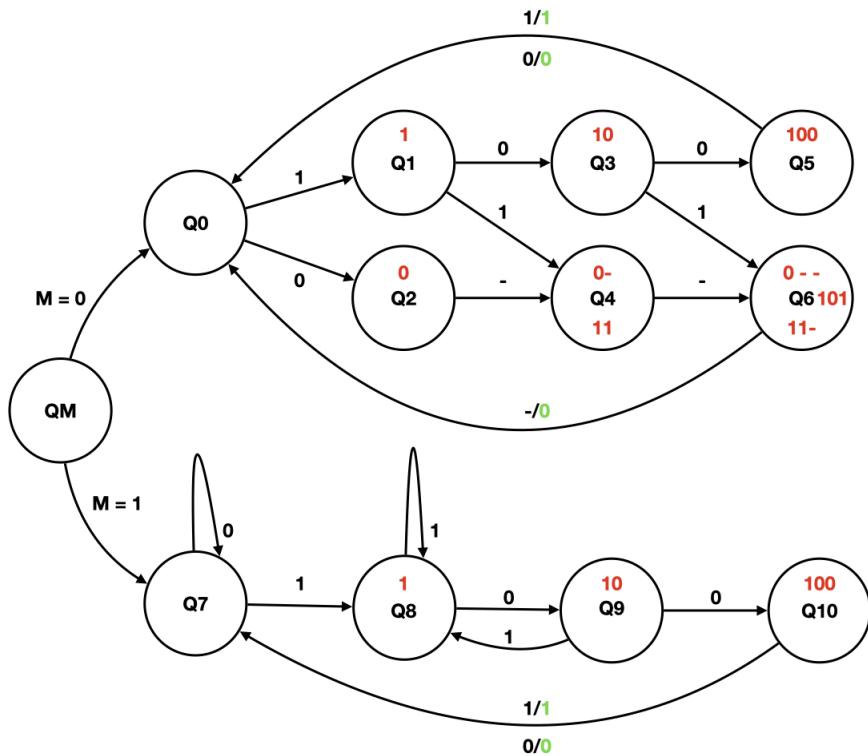


Figura 24: Automa

Entity e architecture - riconoscitore di sequenza

```

entity riconoscitore_sequenza is
  port ( I : in std_logic;
         CLK : in std_logic;
         M : in std_logic := '0';
         RST : in std_logic := '0';
         B_M : in std_logic := '0';
         B_IN : in std_logic := '0';
         Y : out std_logic := '0');
end riconoscitore_sequenza;

architecture Behavioral of riconoscitore_sequenza is
type stato is (QM,Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q10);
  signal stato_corrente : stato := QM;

```

```

signal stato_prossimo : stato := Q0;
signal Y_prossimo : std_logic := '0';
begin
    mem : process(CLK)
begin
    if(rising_edge(CLK)) then
        if(RST = '1' or B_M = '1') then
            stato_corrente <= QM;
            if(M = '1') then
                stato_corrente <= Q7;
            else
                stato_corrente <= Q0;
            end if;
        end if;
    else
        if(B_IN = '1') then
            stato_corrente <= stato_prossimo;
            Y <= Y_prossimo;
        end if;
    end if;
end process;

f_stato_uscita : process(stato_corrente,i)
begin
    case stato_corrente is
        when Q0 =>
            if(I = '0') then
                stato_prossimo <= Q2;
                Y_prossimo <= '0';
            else
                stato_prossimo <= Q1;
                Y_prossimo <= '0';
            end if;
        when Q1 =>
            if(I = '0') then
                stato_prossimo <= Q3;
                Y_prossimo <= '0';
            else

```

```

        stato_prossimo <= Q4;
        Y_prossimo <= '0';
    end if;
when Q2 =>
    stato_prossimo <= Q4;
    Y_prossimo <= '0';
when Q3 =>
    if(I = '0') then
        stato_prossimo <= Q5;
        Y_prossimo <= '0';
    else
        stato_prossimo <= Q6;
        Y_prossimo <= '0';
    end if;
when Q4 =>
    stato_prossimo <= Q6;
    Y_prossimo <= '0';
when Q5 =>
    if(I = '0') then
        stato_prossimo <= Q0;
        Y_prossimo <= '0';
    else
        stato_prossimo <= Q0;
        Y_prossimo <= '1';
    end if;
when Q6 =>
    stato_prossimo <= Q0;
    Y_prossimo <= '0';
when Q7 =>
    if(I = '0') then
        stato_prossimo <= Q7;
        Y_prossimo <= '0';
    else
        stato_prossimo <= Q8;
        Y_prossimo <= '0';
    end if;
when Q8 =>
    if(I = '0') then

```

```

        stato_prossimo <= Q9;
        Y_prossimo <= '0';
    else
        stato_prossimo <= Q8;
        Y_prossimo <= '0';
    end if;
when Q9 =>
    if(I = '0') then
        stato_prossimo <= Q10;
        Y_prossimo <= '0';
    else
        stato_prossimo <= Q8;
        Y_prossimo <= '0';
    end if;
when Q10 =>
    if(I = '0') then
        stato_prossimo <= Q7;
        Y_prossimo <= '0';
    else
        stato_prossimo <= Q7;
        Y_prossimo <= '1';
    end if;
when others =>
    stato_prossimo <= Q0;
    Y_prossimo <= '0';
end case;
end process;
end Behavioral;

```

Simulazione

In figura 25 è mostrata la simulazione del riconoscitore di sequenza in entrambe le modalità di funzionamento.

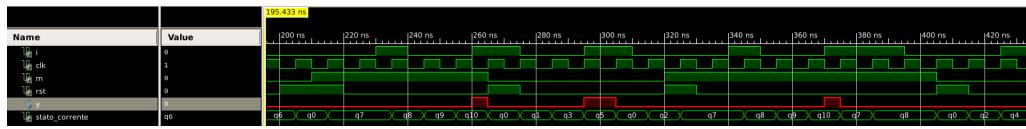


Figura 25: Simulazione Riconoscitore di sequenza

Parte II

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

Al progetto precedente sono stati aggiunti due debouncer per gestire i due pulsanti: *B_IN* e *B_M*, utilizzati allo scopo di rendere significativi l'ingresso e il bit di modo, che vengono inseriti tramite due switch.

Debouncer

La frequenza del clock della scheda è pari a 50 MHz ed è stato ipotizzato un tempo di bouncing di 2s (2.000.000.000 ns). Di conseguenza i colpi di clock di attesa sono 100.000.000. La struttura del debouncer è quella di un automa a due stati: *PRESSED* e *NOT_PRESSED*, gestiti da un contatore. Al termine del numero di colpi prestabilito l'uscita si alza e va a pilotare la macchina sequenziale.

Entity e architecture

```
entity ButtonDebouncer is
    generic ( CLK_period: integer := 20;
              btn_noise_time: integer := 2000000000);

    Port ( CLK : in STD_LOGIC;
            BTN : in STD_LOGIC;
            CLEARED_BTN : out STD_LOGIC);
end ButtonDebouncer;

architecture Behavioral of ButtonDebouncer is
    type stato is (NOT_PRESSED, PRESSED);
    signal BTN_state : stato := NOT_PRESSED;
    constant max_count : integer := btn_noise_time/CLK_period;
```

```

begin
    deb: process (CLK)
        variable count: integer := 0;
    begin
        if (CLK'event and CLK = '1') then
            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;
    end process;
end Behavioral;

```

Riconoscitore su board

Riportiamo in figura 26 l'architettura del sistema complessivo che consentirà successivamente la sintesi e il corretto funzionamento sulla board di sviluppo.

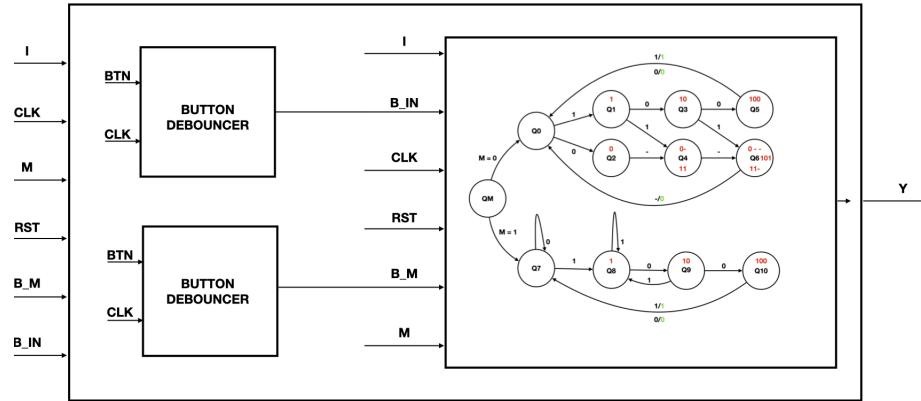


Figura 26: Schema riconoscitore on board

Sintesi

In figura 27 è mostrata la sintesi su fpga del riconoscitore di sequenza. Per motivi di debugging abbiamo aggiunto due segnali per distinguere le due modalità di funzionamento, mappati su due led della scheda.

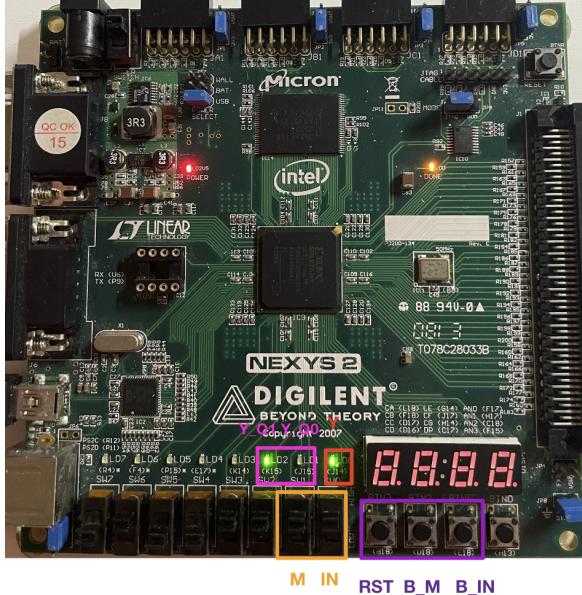


Figura 27: Sintesi riconoscitore

Shift register

Progettare, implementare in VHDL e testare mediante simulazione un registro a scorrimento di N bit in grado di shiftare a destra o a sinistra di un numero Y variabile di posizioni a seconda di una opportuna selezione. Il componente deve essere realizzato utilizzando sia a) approccio comportamentale sia b) approccio strutturale.

Descrizione comportamentale

Per gestire il numero di shift da effettuare viene utilizzata una variabile $nshift$ inizializzata con il valore di Y . La direzione in cui effettuare lo shift è stabilita dal bit di modo M .

A ogni shift il valore di $nshift$ viene decrementato e la macchina si arresta al raggiungimento dello zero; in seguito resta in attesa che si alzi il SET , necessario a riportare $nshift$ al valore più recente di Y . L'eventuale segnale di RST riporta a zero tutti i bit memorizzati nello shift register.

Shift register

A seguire è riportata la descrizione VHDL dello shift register per mezzo di un process mentre la simulazione è visibile in figura 28.

Entity e architecture - shift register

```
entity Shift_register is
    generic ( N : integer := 4);
    port ( CLK : in std_logic;
           Y : in std_logic_vector (0 to 3) := "0000";
           M : in std_logic;
```

```

RST : in std_logic;
SET : in std_logic;
INPUT : in std_logic;
L_OUT : out std_logic;
R_OUT : out std_logic);
end Shift_register;

architecture Behavioral of Shift_register is
    signal T : std_logic_vector(0 to N-1):=(others => '0');
begin
    Processo: process(CLK, RST)
        variable nshift : integer : to_integer(unsigned(Y));
        begin
            if (nshift = 0 and SET = '1') then
                nshift := to_integer(unsigned(Y));
            end if;
            if (RST = '1') then
                T <= (others => '0');
            elsif (CLK'event and CLK = '1') then
                if (M = '0' and nshift > 0) then
                    T(0) <= INPUT;
                    T(1 to N-1) <= T(0 to N-2);
                    nshift := nshift-1;
                elsif(M = '1' and nshift > 0) then
                    T(N-1) <= INPUT;
                    T(0 to N-2) <= T(1 to N-1);
                    nshift := nshift-1;
                end if;
            end if;
        end process;
        R_OUT <= T(N-1);
        L_OUT <= T(0);
    end Behavioral;

```

Simulazione

In figura 28 è mostrata la simulazione dello shift register comportamentale.

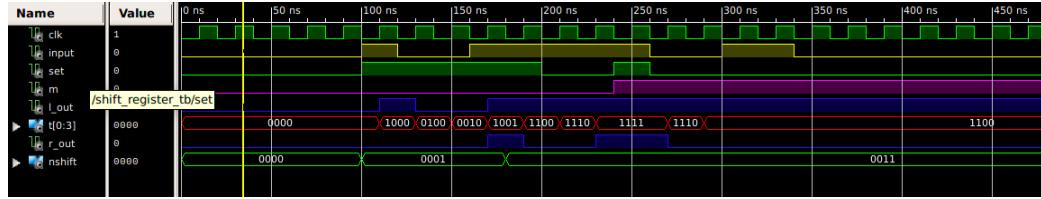


Figura 28: Simulazione shift register comportamentale

Descrizione strutturale

Per realizzare il registro a scorrimento in forma strutturale sono stati utilizzati dei flip-flop di tipo D e i multiplexer 2:1 del primo esercizio per poter gestire le due modalità di funzionamento.

Flip-flop D

Un flip-flop di tipo D presenta la tabella di verità 2 ed è descritto circuitalmente come in figura 29.

I	Q
0	0
1	1

Tabella 2

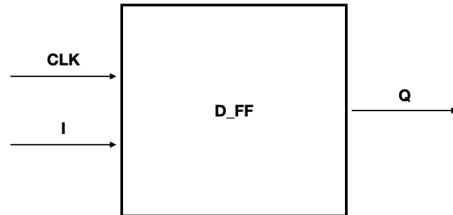


Figura 29: Flip-Flop D

Abbiamo scelto di complicare lo schema circuitale, aggiungendo un segnale di reset asincrono, che fissa il valore memorizzato a 0, indipendentemente dal clock.

```

entity D_FF is
    port ( E : in std_logic;
           CLK : in std_logic;
           RST : in std_logic;
           I : in std_logic;
           Q : out std_logic);
end D_FF;

architecture Behavioral of D_FF is
begin
    proc : process(CLK, RST)
    begin
        if(RST = '1') then
            Q <= '0';
        elsif(rising_edge(CLK) and E = '1') then
            Q <= I;
        end if;
    end process;
end Behavioral;

```

Per completezza, in figura 30 è riportata la simulazione del flip-flop D utilizzato per la composizione dello shift register.

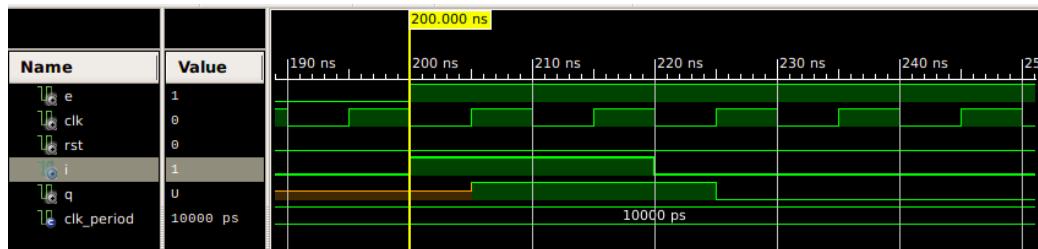


Figura 30: Simulazione flip-flop D

Shift register

Infine è rappresentata in figura 31 la descrizione strutturale del registro a scorrimento in termini di multiplexer e flip-flop.

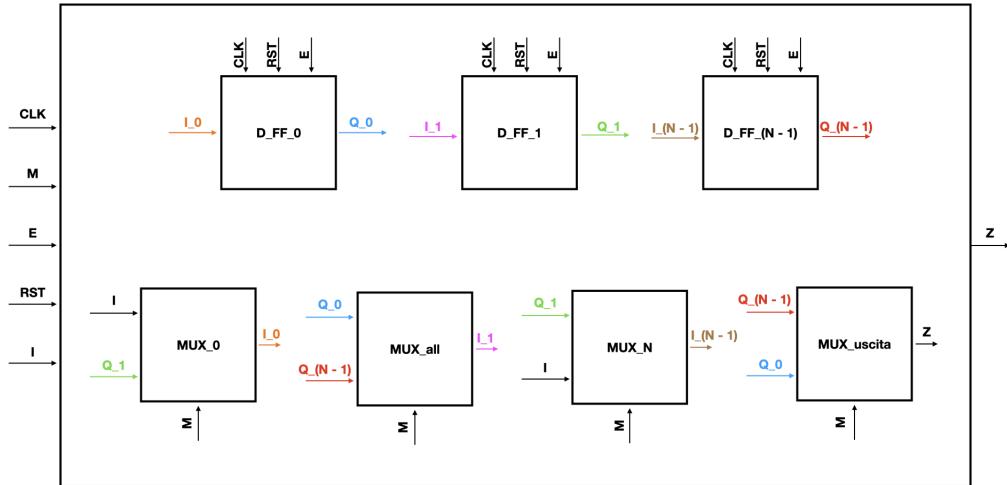


Figura 31: Schema shift register strutturale

Entity e architecture - shift register

```

entity Shift_register is
    generic ( N : integer := 3);
    port ( CLK : in std_logic;
           M : in std_logic;
           E : in std_logic;
           RST : in std_logic;
           I : in std_logic;
           Z : out std_logic);
end Shift_register;

architecture Structural of Shift_register is
    signal U : std_logic_vector(0 to N - 1);
    signal T : std_logic_vector(0 to N - 1);

    component mux_2_1
        port( s : in std_logic;
              a0, a1 : in std_logic;
              y : out std_logic);
    end component;

```

```

end component;

component D_FF
    port ( E : in std_logic;
           CLK : in std_logic;
           RST : in std_logic;
           I : in std_logic;
           Q : out std_logic);
end component;

begin
    mux_all: for j in 1 to N - 2 generate
        mux : mux_2_1
            port map ( s => M,
                       a0 => U(j - 1),
                       a1 => U(j + 1),
                       Y => T(j));
    end generate;

    mux_0 : mux_2_1
        port map ( s => M,
                   a0 => I,
                   a1 => U(1),
                   Y => T(0));

    mux_N : mux_2_1
        port map ( s => M,
                   a0 => U(N - 2),
                   a1 => I,
                   Y => T(N - 1));

    mux_uscita : mux_2_1
        port map ( s => M,
                   a0 => U(N - 1),
                   a1 => U(0),
                   Y => Z);

D_FF_all : for j in 1 to N - 2 generate

```

```

D_FF_i : D_FF
    port map ( E => E,
                CLK => CLK,
                RST => RST,
                I => T(j),
                Q => U(j));
end generate;

D_FF_0 : D_FF
    port map ( E => E,
                CLK => CLK,
                RST => RST,
                I => T(0),
                Q => U(0));

D_FF_N : D_FF
    port map ( E => E,
                CLK => CLK,
                RST => RST,
                I => T(N - 1),
                Q => U(N - 1));
end Structural;

```

Simulazione

In figura 32 è mostrata la simulazione dello shift register strutturale.

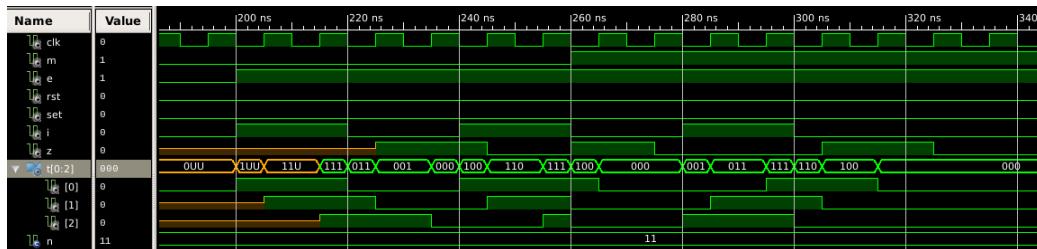


Figura 32: Simulazione shift register strutturale

Contatore

Per tener conto della variabilità del numero di shift da effettuare, viene usato un opportuno contatore, che abilita i flip-flop a evolvere fino a quando non si raggiunge il conteggio massimo, che coincide con il valore di Y . Il segnale di $START$ ci consente di azzerare il conteggio e di aggiornare il conteggio massimo con il valore più recente di Y .

Il contatore riceve quindi in ingresso il valore di Y e abilita i flip-flop del registro a scorrimento Y volte.

L'entity e l'architettura sono i seguenti:

```
entity Counter is
    port ( CLK : in std_logic;
           RST : in std_logic;
           START : in std_logic;
           Y : in std_logic_vector(0 to 3);
           C : out std_logic);
end Counter;

architecture Behavioral of Counter is
begin
    proc : process(RST, CLK)
        variable count : integer := 0;
        variable count_max : integer := to_integer(unsigned(Y));
    begin
        if(RST = '1') then
            C <= '0';
            count := 0;
        elsif(CLK'event and CLK = '1') then
            if(count /= count_max) then
                C <= '1';
                count := count + 1;
            else
                C <= '0';
                if(START = '1') then
                    count := 0;
                    count_max := to_integer(unsigned(Y));
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

```

        end if;
    end if;
end process;
end Behavioral;
```

Interconnettendo opportunamente, come visibile dalla figura 33, il contatore e lo shift register strutturale otteniamo la composizione finale del sistema complessivo.

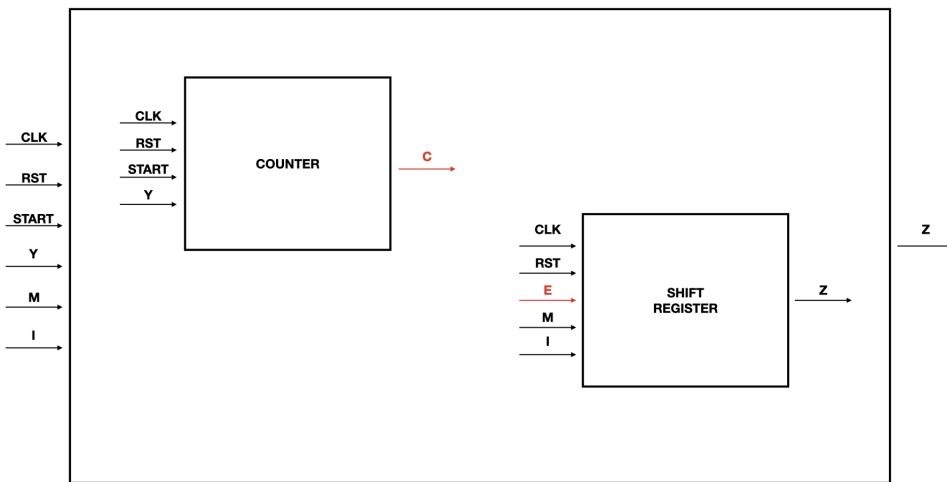


Figura 33: Shift Register Complessivo

Entity e architecture - shift register complessivo

```

entity Shift_register_mod is
    port ( CLK : in std_logic;
           RST : in std_logic;
           START : in std_logic;
           M : in std_logic;
           Y : in std_logic_vector(0 to 3);
           I : in std_logic;
           Z : out std_logic);
end Shift_register_mod;

architecture Behavioral of Shift_register_mod is
    signal T : std_logic;
```

```

component Counter
    port ( CLK : in std_logic;
           RST : in std_logic;
           START : in std_logic;
           Y : in std_logic_vector(0 to 3);
           C : out std_logic);
end component;

component Shift_register
    port ( CLK : in std_logic;
           M : in std_logic;
           E : in std_logic;
           RST : in std_logic;
           I : in std_logic;
           Z : out std_logic);
end component;

component Shift_register
    port ( CLK : in std_logic;
           M : in std_logic;
           E : in std_logic;
           RST : in std_logic;
           I : in std_logic;
           Z : out std_logic);
end component;

begin
    Counter_0 : Counter
        port map ( CLK => CLK,
                   RST => RST,
                   START => START,
                   Y => Y,
                   C => T);

    Shift_register_0 : Shift_register
        port map ( CLK => CLK,
                   M => M,
                   E => T,
                   RST => RST,
                   I => I,

```

```

    Z => Z);
end Behavioral;

```

Simulazione

In figura 34 è riportata la simulazione del comportamento dello shift register complessivo.

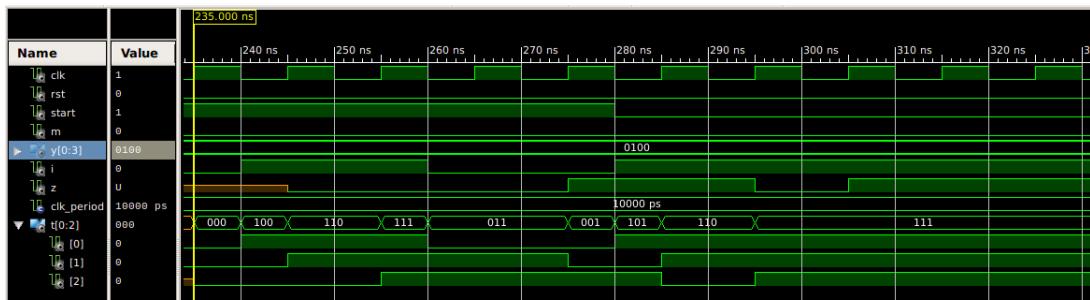


Figura 34: Simulazione Shift Register Complessivo

Cronometro

Parte I

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.

Per la realizzazione del cronometro abbiamo adottato un approccio strutturale. L'elemento base della nostra architettura è un Flip-flop T asincrono che lavora sul fronte di discesa.

Flip-flop T

Entity e architecture - Flip-flop T

```
entity FFT is
port(
  clk : in std_logic;
  en : in std_logic;
  rst : in std_logic;
  set : in std_logic;
  I : in std_logic;
  Y : out std_logic
);
```

```

end FFT;

architecture Behavioral of FFT is
    signal T : std_logic := '0';
begin
    Y <= T;
    proc : process(clk, rst, set) --rst asincrono
    begin
        if (rst = '1') then
            T <= '0';
        elsif (set = '1') then
            T <= I;
        elsif (falling_edge(clk) and en = '1') then
            T <= not T;
        end if;
    end process;

```

Simulazione - Flip-flop T

In figura 35 è possibile osservare il comportamento del flip-flop T mediante simulazione.

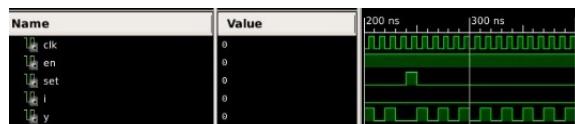


Figura 35: Simulazione fft

Contatori

Abbiamo scelto di utilizzare una configurazione parallela, in cui il clock viene dato a tutti i flip-flop T per evitare i problemi di propagazione dello schema seriale. A ogni colpo di clock il primo flip-flop commuta sempre. Gli altri commutano sul fronte di discesa del clock se il precedente ha raggiunto il conteggio massimo. In figura 36 è mostrato un contatore mod8 realizzato con la configurazione parallela, ma considerazioni analoghe valgono per un contatore modulo 60 o 24.

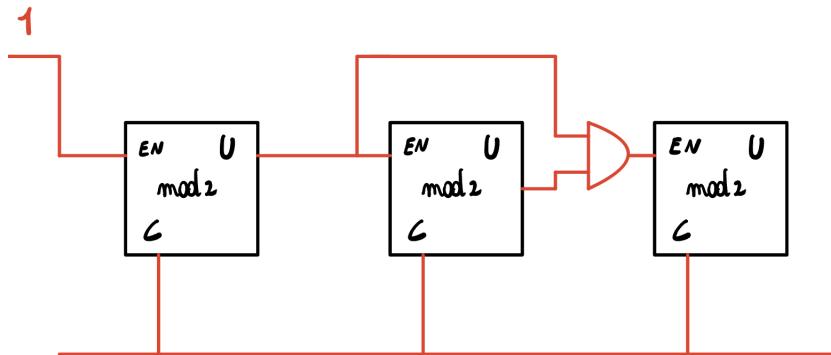


Figura 36: Schema parallelo - contatore modulo 8

Entity e architecture - Contatore modulo 24

```

entity Contatore_24 is
port(
    clk : in std_logic;
    en : in std_logic;
    rst : in std_logic;
    set : in std_logic;
    I : in std_logic_vector(4 downto 0);
    Y : out std_logic_vector(4 downto 0);
    DIV : out std_logic
);
end Contatore_24;

architecture Structural of Contatore_24 is

signal t : std_logic_vector(3 downto 0) := (others => '0');
signal usc : std_logic_vector(4 downto 0) := (others => '0');
signal r : std_logic := '0';

component FFT is
port(
    clk : in std_logic;
    en : in std_logic;
    rst : in std_logic;

```

```

    set : in std_logic;
    I : in std_logic;
    Y : out std_logic
);
end component;

begin
r <= rst or
(usc(4) and usc(3) and not usc(2) and
not usc(1) and not usc(0)); --24
F1 : FFT
port map(
clk => clk,
en => en,
rst => r,
set => set,
I => I(0),
Y => usc(0)
);
t(0) <= clk and usc(0);

F2 : FFT
port map(
clk => t(0),
en => en,
rst => r,
set => set,
I => I(1),
Y => usc(1)
);
t(1) <= clk and usc(0) and usc(1);

F3 : FFT
port map(
clk => t(1),
en => en,

```

```

rst => r,
set => set,
I => I(2),
Y => usc(2)
);

t(2) <= clk and usc(0) and usc(1) and usc(2);

F4 : FFT
port map(
clk => t(2),
en => en,
rst => r,
set => set,
I => I(3),
Y => usc(3)
);

t(3) <= clk and usc(0) and usc(1) and usc(2) and usc(3);

F5 : FFT
port map(
clk => t(3),
en => en,
rst => r,
set => set,
I => I(4),
Y => usc(4)
);

DIV <= usc(4) and not usc(3) and usc(2) and usc(1) and usc(0); --23

Y <= usc(4) & usc(3) & usc(2) & usc(1) & usc(0);

end Structural;

```

Simulazione

In figura 37 è mostrata l'evoluzione di un contatore modulo 24 mediante simulazione.

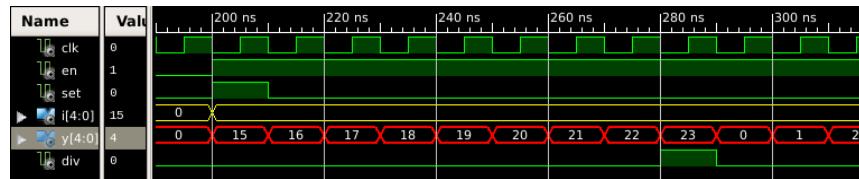


Figura 37: Simulazione contatore modulo 24

Entity e architecture - Contatore modulo 60

```
entity Contatore_60 is
port(
    clk : in std_logic;
    en : in std_logic;
    rst : in std_logic;
    set : in std_logic;
    I : in std_logic_vector(5 downto 0);
    Y : out std_logic_vector(5 downto 0);
    DIV : out std_logic
);
end Contatore_60;

architecture Structural of Contatore_60 is

signal usc : std_logic_vector(5 downto 0) := (others => '0');
signal t : std_logic_vector(4 downto 0) := (others => '0');
signal r : std_logic := '0';

component FFT is
port(
    clk : in std_logic;
    en : in std_logic;
    rst : in std_logic;
```

```

    set : in std_logic;
    I : in std_logic;
    Y : out std_logic
);
end component;

begin

r <= rst or
    (usc(5) and usc(4) and usc(3) and usc(2) and
        not usc(1) and not usc(0)); --60

F1 : FFT
port map(
    clk => clk,
    en => en,
    rst => r,
    set => set,
    I => I(0),
    Y => usc(0)
);

t(0) <= clk and usc(0);

F2 : FFT
port map(
    clk => t(0),
    en => en,
    rst => r,
    set => set,
    I => I(1),
    Y => usc(1)
);

t(1) <= clk and usc(0) and usc(1);

F3 : FFT
port map(

```

```

clk => t(1),
en => en,
rst => r,
set => set,
I => I(2),
Y => usc(2)
);

t(2) <= clk and usc(0) and usc(1) and usc(2);

F4 : FFT
port map(
clk => t(2),
en => en,
rst => r,
set => set,
I => I(3),
Y => usc(3)
);

t(3) <= clk and usc(0) and usc(1) and usc(2) and usc(3);

F5 : FFT
port map(
clk => t(3),
en => en,
rst => r,
set => set,
I => I(4),
Y => usc(4)
);

t(4) <= clk and usc(0) and usc(1) and usc(2) and usc(3) and usc(4);

F6 : FFT
port map(
clk => t(4),
en => en,

```

```

    rst => r,
    set => set,
    I => I(5),
    Y => usc(5)
);

DIV <= usc(5) and usc(4) and usc(3) and not usc(2) and usc(1) and usc(0); --59

Y <= usc(5) & usc(4) & usc(3) & usc(2) & usc(1) & usc(0);

end Structural;

```

Simulazione

In figura 38 è mostrata l'evoluzione di un contatore modulo 60 mediante simulazione.



Figura 38: Simulazione contatore modulo 60

Cronometro

Il cronometro si compone di due contatori modulo 60 e un contatore modulo 24, disposti anche in questo caso in configurazione parallela, come raffigurato in figura 39.

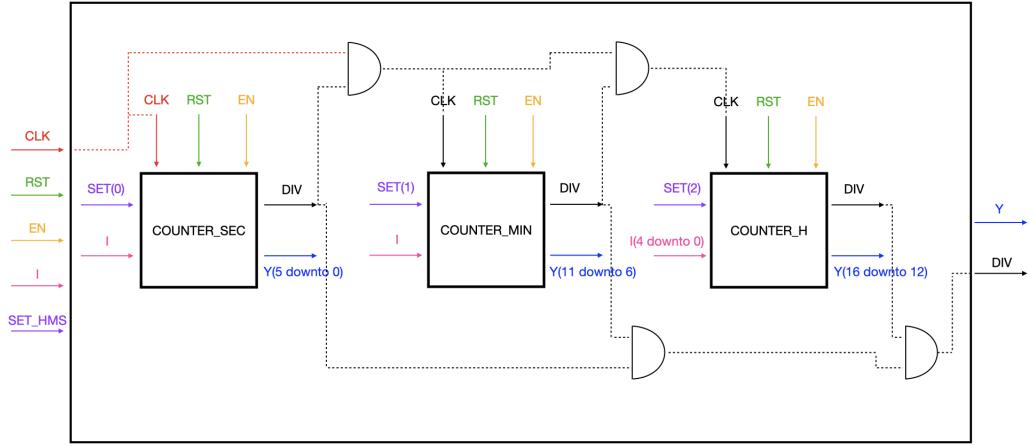


Figura 39: Schema cronometro

Entity e architecture - Cronometro

```

entity Cronometro_hh_mm_ss is
port(
    clk : in std_logic;
    rst : in std_logic;
    en : in std_logic;
    set_hms : in std_logic_vector(2 downto 0);
    I : in std_logic_vector(5 downto 0);
    Y : out std_logic_vector(16 downto 0);
    DIV : out std_logic
);
end Cronometro_hh_mm_ss;

architecture Structural of Cronometro_hh_mm_ss is

component Contatore_60 is
port(
    clk : in std_logic;
    en : in std_logic;
    rst : in std_logic;
    set : in std_logic;

```

```

I : in std_logic_vector(5 downto 0);
Y : out std_logic_vector(5 downto 0);
DIV : out std_logic
);
end component;

component Contatore_24 is
port(
    clk : in std_logic;
    en : in std_logic;
    rst : in std_logic;
    set : in std_logic;
    I : in std_logic_vector(4 downto 0);
    Y : out std_logic_vector(4 downto 0);
    DIV : out std_logic
);
end component;

signal usc : std_logic_vector(2 downto 0) := (others => '0');
signal t : std_logic_vector(1 downto 0) := (others => '0');

begin

Secondi : Contatore_60
port map(
    clk => clk,
    rst => rst,
    en => en,
    set => set_hms(0),
    I => I,
    Y => Y(5 downto 0),
    DIV => usc(0)
);

t(0) <= clk and usc(0);

Minuti : Contatore_60

```

```

port map(
  clk => t(0),
  rst => rst,
  en => en,
  set => set_hms(1),
  I => I,
  Y => Y(11 downto 6),
  DIV => usc(1)
);

t(1) <= clk and usc(0) and usc(1);

Ore : Contatore_24
port map(
  clk => t(1),
  rst => rst,
  en => en,
  set => set_hms(2),
  I => I(4 downto 0),
  Y => Y(16 downto 12),
  DIV => usc(2)
);

DIV <= usc(2) and usc(1) and usc(0);

end Structural;

```

Simulazione

In figura 40 è possibile osservare il comportamento del cronometro progettato.

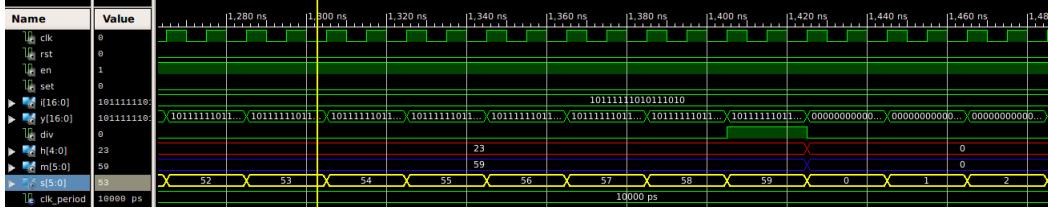


Figura 40: Simulazione cronometro

Parte II

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 pulsanti, uno per il set dell'orario e uno per il reset. Si utilizzi una codifica a scelta dello studente per la visualizzazione dell'orario sui display (esadecimale o decimale).

Per ovviare alla mancanza di switch nell'FPGA, per dare la possibilità all'utente esterno di impostare un orario desiderato, abbiamo messo a disposizione 6 switch, che impostano, tramite 2 switch di selezione, le ore, i minuti o i secondi. Una volta settati gli switch, alla pressione di un pulsante il segnale di *SET* si propaga al contatore selezionato. Per evitare fluttuazioni abbiamo inserito un debouncer in modo tale che il segnale di *SET* sia alto esattamente per un colpo di clock. Avendo a disposizione un display con solo 4 cifre, 12 bit dei minuti e dei secondi dell'uscita del cronometro sono utilizzati per pilotare il display mentre i restanti 5 bit pilotano 5 led esterni.

Divisore di frequenza

Il divisore permette di tradurre la frequenza del clock della board in quella desiderata (1 Hz).

Entity e architecture - Divisore di frequenza

```
entity Divisore_freq is
generic(
```

```

freq_in : integer := 50000000;
freq_out : integer := 1
);

port(
    clk : in std_logic;
    en : in std_logic;
    rst : in std_logic;
    clk_out : out std_logic
);
end Divisore_freq;

architecture Behavioral of Divisore_freq is
    constant cont_max : integer := freq_in/(freq_out)-1;
begin

    proc : process(clk)
        variable cont : integer range 0 to cont_max := 0;
    begin
        if (rising_edge(clk)) then
            if (rst = '1') then
                cont := 0;
                clk_out <= '0';
            elsif (en = '1') then
                if (cont = cont_max) then
                    cont := 0;
                    clk_out <= '1';
                else
                    cont := cont + 1;
                    clk_out <= '0';
                end if;
            end if;
        end if;
    end process;
end Behavioral;

```

Gestore display a sette segmenti

Il gestore del display prende in ingresso 16 bit, provenienti dal codificatore, e li mappa sulle 4 cifre a sette segmenti. Poichè è possibile illuminare una sola cifra alla volta bisogna scegliere un refresh rate che permetta all'osservatore umano di non percepire l'alternanza tra le cifre. Il range consigliato è tra 1 kHz e 60 Hz, cioè un refresh period tra 1 e 16 ms. La nostra scelta è ricaduta su 500 Hz; di conseguenza il display subisce un refresh ogni 2 ms e ciascuna cifra ogni 500 μs .

Entity e architecture - Seven segment display

```
entity Seven_segments_display is
  port(
    clk : in std_logic;
    en : in std_logic;
    rst : in std_logic;
    enable_digit : in std_logic_vector(3 downto 0);
    value : in std_logic_vector(15 downto 0);
    dots : in std_logic_vector(3 downto 0);
    anodes : out std_logic_vector(3 downto 0);
    cathodes : out std_logic_vector(7 downto 0)
  );
end Seven_segments_display;

architecture Structural of Seven_segments_display is
  component Divisore_freq is
    generic(
      freq_in : integer := 50000000;
      freq_out : integer := 500
    );
    port(
      clk : in std_logic;
      en : in std_logic;
      rst : in std_logic;
      clk_out : out std_logic
    );
  end component;
```

```

component counter_mod4 is
    Port(
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        enable : in STD_LOGIC;
        counter : out STD_LOGIC_VECTOR (1 downto 0)
    );
end component;
component Anodes_manager is
    port(
        cont : in std_logic_vector(1 downto 0);
        enable_digit : in std_logic_vector(3 downto 0);
        anodes : out std_logic_vector(3 downto 0)
    );
end component;
component Cathodes_manager is
    port(
        cont : in std_logic_vector(1 downto 0);
        value : in std_logic_vector(15 downto 0);
        dots : in std_logic_vector(3 downto 0);
        cathodes : out std_logic_vector(7 downto 0)
    );
end component;
signal d : std_logic := '0';
signal e : std_logic := '0';
signal c : std_logic_vector(1 downto 0) := (others => '0');
begin
    cnt: counter_mod4
        port map(
            clock => clk,
            reset => rst,
            enable => e,
            counter => c
        );
    Divis_f : Divisore_freq
        generic map(
            freq_in => 50000000,
            freq_out => 500

```

```

)
port map(
    clk => clk,
    en => en,
    rst => rst,
    clk_out => e
);
A_m : Anodes_manager
port map(
    cont => c,
    enable_digit => enable_digit,
    anodes => anodes
);
C_m : Cathodes_manager
port map(
    cont => c,
    value => value,
    dots => dots,
    cathodes => cathodes
);
end Structural;

```

Anodes manager

Il gestore degli anodi riceve il segnale di conteggio dal contatore modulo 4, andando ad illuminare la relativa cifra secondo il refresh rate stabilito con il divisore di frequenza.

Entity e architecture - Anodes manager

```

entity Anodes_manager is
port(
    cont : in std_logic_vector(1 downto 0);
    enable_digit : in std_logic_vector(3 downto 0);
    anodes : out std_logic_vector(3 downto 0)
);
end Anodes_manager;

```

```

architecture Behavioral of Anodes_manager is

    signal anodes_switching : std_logic_vector(3 downto 0);

begin
    anodes_process : process(cont)
    begin
        case cont is
            when "00" =>
                anodes_switching <= "0001";
            when "01" =>
                anodes_switching <= "0010";
            when "10" =>
                anodes_switching <= "0100";
            when "11" =>
                anodes_switching <= "1000";
            when others=>
                anodes_switching <= (others=>'0');

        end case;
    end process;

    anodes <= anodes_switching nand enable_digit;

end Behavioral;

```

Cathodes manager

Entity e architecture - Cathodes manager

```

entity Cathodes_manager is
    port(
        cont : in std_logic_vector(1 downto 0);
        value : in std_logic_vector(15 downto 0);
        dots : in std_logic_vector(3 downto 0);
        cathodes : out std_logic_vector(7 downto 0)
    );

```

```

end Cathodes_manager;

architecture Behavioral of Cathodes_manager is

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

alias digit_0 is value (15 downto 12);
alias digit_1 is value (11 downto 8);
alias digit_2 is value (7 downto 4);
alias digit_3 is value (3 downto 0);

signal cathodes_for_digit : std_logic_Vector(6 downto 0) := (others => '0');
signal nibble :std_logic_vector(3 downto 0) := (others => '0');
signal dot :std_logic := '0';
begin

-- questo processo multiplexa le cifre da mostrare
digit_switching: process(cont)
begin

case cont is
when "00" =>

```

```

        nibble <= digit_3;
        dot <= dots(0);
when "01" =>
        nibble <= digit_2;
        dot <= dots(1);
when "10" =>
        nibble <= digit_1;
        dot <= dots(2);
when "11" =>
        nibble <= digit_0;
        dot <= dots(2);

when others =>
        nibble <= (others => '0');
        dot <= '0';
end case;
end process;

seven_segment_decoder_process: process(nibble)
begin
    case nibble is
        when "0000" => cathodes_for_digit <= zero;
        when "0001" => cathodes_for_digit <= one;
        when "0010" => cathodes_for_digit <= two;
        when "0011" => cathodes_for_digit <= three;
        when "0100" => cathodes_for_digit <= four;
        when "0101" => cathodes_for_digit <= five;
        when "0110" => cathodes_for_digit <= six;
        when "0111" => cathodes_for_digit <= seven;
        when "1000" => cathodes_for_digit <= eight;
        when "1001" => cathodes_for_digit <= nine;
        when "1010" => cathodes_for_digit <= a;
        when "1011" => cathodes_for_digit <= b;
        when "1100" => cathodes_for_digit <= c;
        when "1101" => cathodes_for_digit <= d;
        when "1110" => cathodes_for_digit <= e;
        when "1111" => cathodes_for_digit <= f;
when others => cathodes_for_digit <= (others => '0');

```

```

    end case;
end process seven_segment_decoder_process;

cathodes <= (not dot)&cathodes_for_digit;

end Behavioral;

```

Codificatore

Poichè il display utilizzato ha un'interfaccia di 16 bit, 4 per ogni cifra, abbiamo utilizzato un circuito combinatorio (Codificatore) che mappa opportunamente questi 12 bit del cronometro sulle 4 cifre del display. Si tratta di una macchina combinatoria che riceve 12 bit provenienti dal contatore dei minuti e dei secondi e restituisce in uscita la codifica su 16 bit delle quattro cifre da mostrare sul display.

Entity e architecture - Codificatore

```

entity codificatore is
  port(
    I : in std_logic_vector(0 to 11);
    Z : out std_logic_vector(0 to 15)
  );
end codificatore;

architecture Behavioral of codificatore is

  ALIAS minuti : std_logic_vector(0 to 5) is I(0 to 5);
  ALIAS secondi: std_logic_vector(0 to 5) is I(6 to 11);
  ALIAS cifra1 : std_logic_vector(0 to 3) is Z(0 to 3);
  ALIAS cifra2 : std_logic_vector(0 to 3) is Z(4 to 7);
  ALIAS cifra3 : std_logic_vector(0 to 3) is Z(8 to 11);
  ALIAS cifra4 : std_logic_vector(0 to 3) is Z(12 to 15);

begin
  ...

```

```
end Behavioral;
```

La codifica viene effettuata esplicitamente per tutte le cifre, come di seguito:

```
cifra1 <= "0000" when(minuti = "000000" or minuti = "000001" or minuti =
    "000010" or minuti = "000011" or minuti = "000100" or minuti =
    "000101"or minuti = "000110"or minuti = "000111"or minuti =
    "001000" or minuti = "001001")
else
    "0001" when(minuti = "001010" or minuti = "001011" or minuti =
    "001100" or minuti = "001101" or minuti = "001110" or minuti =
    "001111"or minuti = "010000"or minuti = "010001"or minuti =
    "010010" or minuti = "010011")
else
    "0010" when(minuti = "010100" or minuti = "010101" or minuti =
    "010110" or minuti = "010111" or minuti = "011000" or minuti =
    "011001"or minuti = "011010"or minuti = "011011"or minuti =
    "011100" or minuti = "011101")
else
    "0011" when(minuti = "011110" or minuti = "011111" or minuti =
    "100000" or minuti = "100001" or minuti = "100010" or minuti =
    "100011"or minuti = "100100"or minuti = "100101"or minuti =
    "100110" or minuti = "100111")
else
    "0100" when(minuti = "101000" or minuti = "101001" or minuti =
    "101010" or minuti = "101011" or minuti = "101100" or minuti =
    "101101"or minuti = "101110"or minuti = "101111"or minuti =
    "110000" or minuti = "110001")
else
    "0101" when(minuti = "110010" or minuti = "110011" or minuti =
    "110100" or minuti = "110101" or minuti = "110110" or minuti =
    "110111"or minuti = "111000"or minuti = "111001"or minuti =
    "111010" or minuti = "111011")
else
    "----";
```

Controllore

Per evitare che l'utente possa inserire valori non coerenti per ore, minuti e secondi, il controllore filtra l'input, imponendolo a zero nel caso di valori inammissibili. Al variare del *SET*, l'input passa inalterato in uscita se i vincoli sono rispettati, altrimenti viene azzerato.

Entity e architecture - Controllore

```
entity ControlloreI is
    Port(
        I : in STD_LOGIC_VECTOR(5 downto 0);
        SET : in STD_LOGIC;
        HMS : in STD_LOGIC_VECTOR(1 downto 0);
        Output : out STD_LOGIC_VECTOR(5 downto 0)
    );
end ControlloreI;

architecture Behavioral of ControlloreI is

begin
control_process: process(SET)
begin
    if (HMS="10") then
        if I>"010111" then
            Output<="000000";
        else
            Output<=I;
        end if;
    else
        if (I>"111011") then
            Output<="000000";
        else
            Output<=I;
        end if;
    end if;
    end process;
end Behavioral;
```

Cronometro su board

Lo schema complessivo è in figura 41.

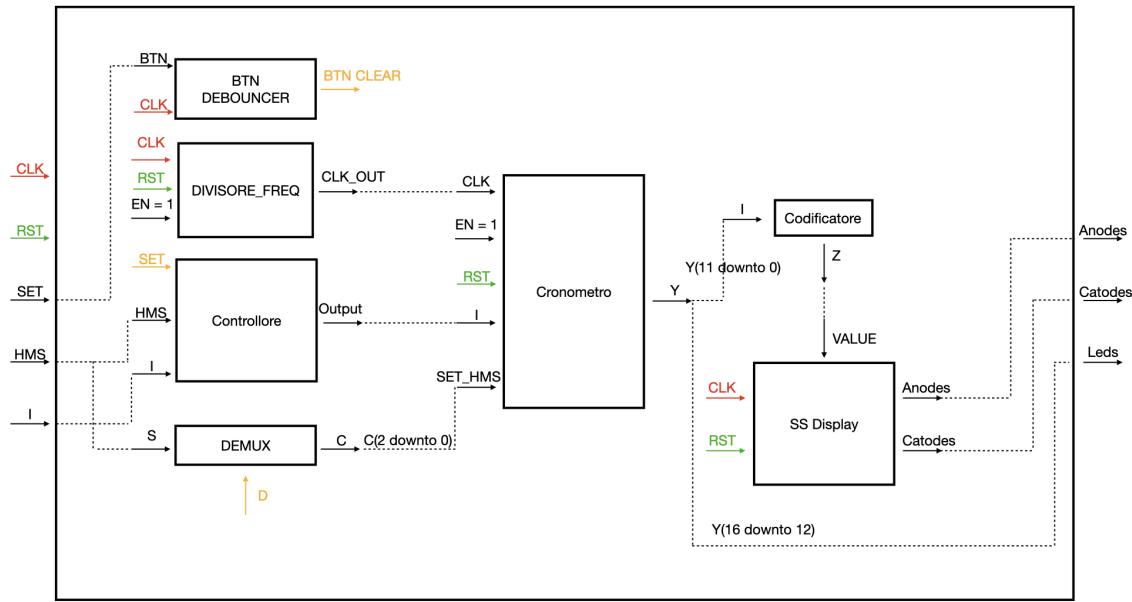


Figura 41: Schema cronometro su board

Sintesi

Terminata la progettazione, il sistema finale è stato sintetizzato sulla board di sviluppo a disposizione, come visibile dalla figura 42.

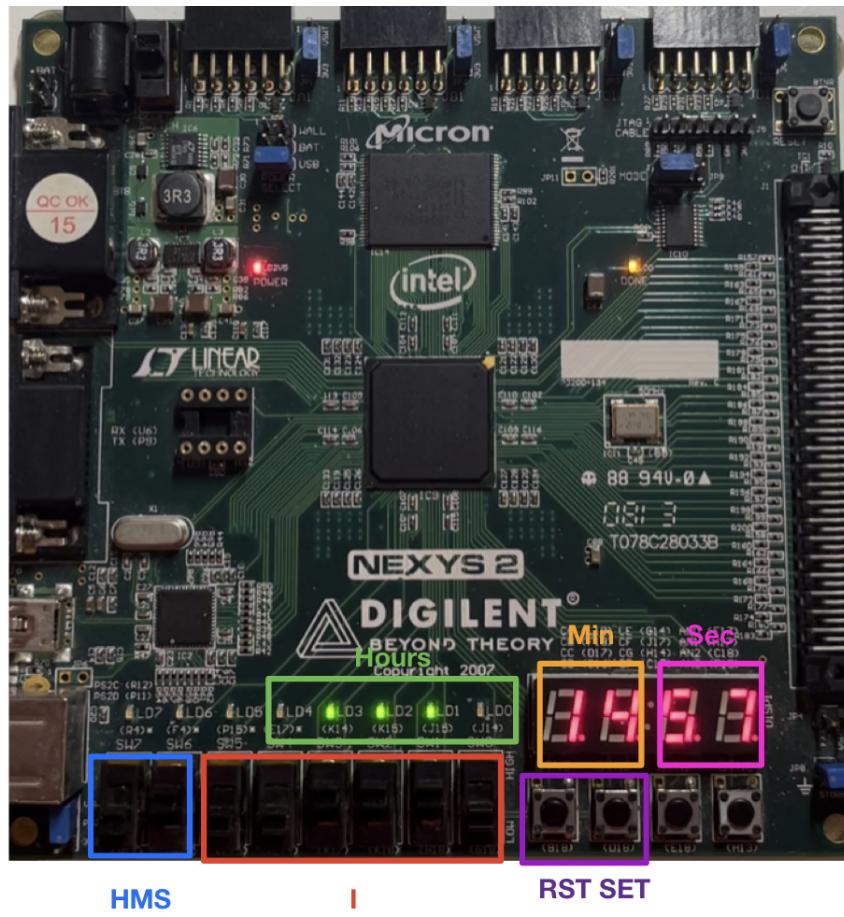


Figura 42: Sintesi cronometro

Parte III

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

L'utilizzatore può memorizzare a ogni pressione del pulsante un intertempo in una apposita RAM fino ad un totale di $N = 4$. Solo al termine della memorizzazione degli N intertempi è possibile iniziare la visualizzazione di quest'ultimi. Tale scelta deriva dal fatto di poter in questo modo fare a meno di un secondo bottone e di un secondo contatore.

Contatore

Il contatore riceve in ingresso l'uscita del debouncer, ossia un impulso di durata pari al clock, e fornisce in uscita un segnale di conteggio che va a scandire gli indirizzi della RAM per la scrittura. In aggiunta presenta un segnale e che resta alto fintanto che non sono terminate le N scritture degli intertempi.

Entity e architecture - Contatore

```
entity Counter_address is
  Port(
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    enable : in STD_LOGIC;
    counter : out STD_LOGIC_VECTOR (3 downto 0);
    e : out std_logic := '1'
  );
end Counter_address;

architecture Behavioral of Counter_address is
  signal c : std_logic_vector (3 downto 0) := (others => '0');
begin
```

```

counter <= c;
counter_process: process(clock)
begin
  if(rising_edge(clock)) then
    if reset = '1' then
      e <= '1';
      c <= (others => '0');
    elsif enable = '1' then
      if(c = "0011") then
        e <= '0';
        c <= (others => '0');
      else
        c <= std_logic_vector(unsigned(c) + 1);
      end if;
    end if;
    end if;
  end process;

end Behavioral;

```

RAM

Entity e architecture - RAM

La RAM Inizialmente a ogni colpo di clock consente di scrivere alla locazione fornita dal contatore il dato in ingresso (il write enable è alto). Successivamente, quando il write enable si abbassa, in uscita è disponibile il dato presente ad una determinata locazione.

```

entity RAM is
  port ( clk  : in std_logic;
         rst  : in std_logic;
         we   : in std_logic;
         en   : in std_logic;
         addr : in std_logic_vector(3 downto 0);
         di   : in std_logic_vector(16 downto 0);
         do   : out std_logic_vector(16 downto 0));

```

```

end RAM;
architecture Behavioral of RAM is
  type ram_type is array (0 to 15) of std_logic_vector (16 downto 0);
  signal RAM : ram_type;

begin
  process (clk)
    begin
      if clk'event and clk = '1' then
        if rst = '1' then
          RAM <= (others => (others => '0'));
        elsif en = '1' then
          if we = '1' then
            RAM(conv_integer(addr)) <= di;
          else
            do <= RAM(conv_integer(addr));
          end if;
        end if;
      end if;
    end process;
  end Behavioral;

```

Cronometro con intertempi

La AND tra e e il segnale in uscita dal debouncer è collegata al write enable della RAM. In tal modo a ogni pressione è possibile salvare un intertempo e incrementare il contatore. Quando poi sono terminate le N scritture e si abbassa il segnale e in uscita al contatore, si abbassa di conseguenza il write enable, si azzera il conteggio e la RAM si predisponde a leggere i valori memorizzati. Inoltre, il segnale e pilota un multiplexer che consente di visualizzare al termine delle scritture l'uscita della RAM sul display e sui led, piuttosto che quella del cronometro. Continuando a premere il pulsante è possibile visualizzare ognuno degli intertempi a rotazione. Per riprendere il normale funzionamento del cronometro è necessario un reset.

In figura 43 è descritta l'architettura interna del sistema complessivo.

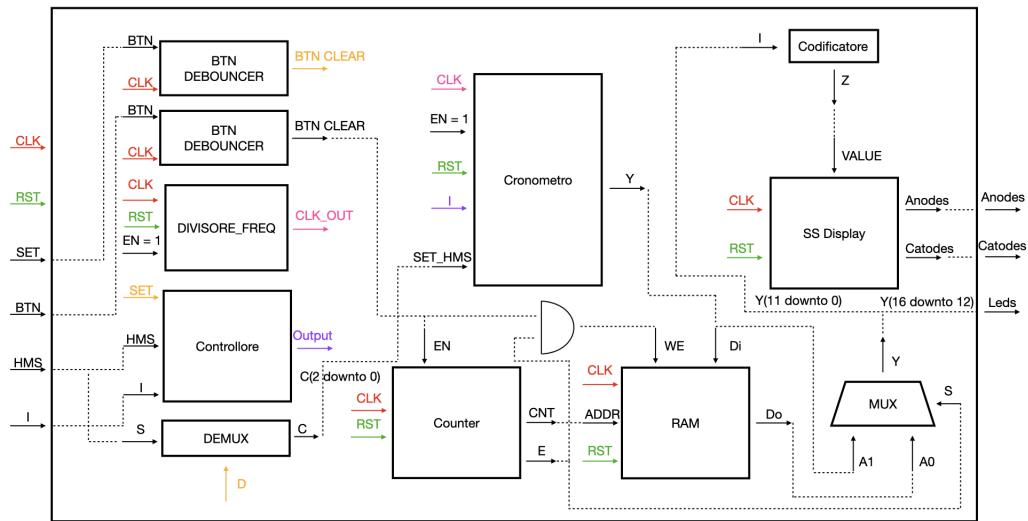


Figura 43: Schema cronometro con intertempi

Sistema di testing

Parte I

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).

ROM

Gli N valori di input per il test vengono letti da una ROM 16×4 .

Entity and architecture - ROM

```
entity ROM is
    port ( CLK : in std_logic;
           RST : in std_logic;
           EN : in std_logic;
           ADDR : in std_logic_vector(3 downto 0);
           DATA : out std_logic_vector(3 downto 0));
end ROM;

architecture behavioral of ROM is
    type rom_type is array (15 downto 0) of std_logic_vector(3 downto 0);
    signal ROM : rom_type := (
        X"0",
        X"1",
        X"2",
        X"3",
```

```

        X"4",
        X"5",
        X"6",
        X"7",
        X"8",
        X"9",
        X"A",
        X"B",
        X"C",
        X"D",
        X"E",
        X"F");
attribute rom_style : string;
attribute rom_style of ROM : signal is "block";

begin
process(CLK)
begin
    if rising_edge(CLK) then
        if (RST = '1') then
            DATA <= ROM(conv_integer("0000"));
        elsif (EN = '1') then
            DATA <= ROM(conv_integer(ADDR));
        end if;
    end if;
end process;
end behavioral;

```

Macchina combinatoria

La macchina combinatoria scelta riceve $N = 4$ bit di ingresso e restituisce in uscita su 3 bit il numero di bit in ingresso alti. La tabella di verità della nostra macchina combinatoria è stata implementata interamente, consci del fatto che un'eventuale minimizzazione, data la tecnologia target, non avrebbe portato benefici.

Entity and architecture - macchina combinatoria

```
entity Macchina_combinatoria is
    port ( X : in std_logic_vector(3 downto 0);
           Y : out std_logic_vector(2 downto 0));
end Macchina_combinatoria;

architecture Dataflow of Macchina_combinatoria is
begin
    Y <=
        "000" when X = "0000" else
        "001" when (X = "0001") or (X = "0010") or (X = "0100") or (X = "1000") else
        "010" when (X = "0011") or (X = "0101") or (X = "1001") or (X = "1010") or
        (X = "0110") or (X = "1100") else
        "011" when (X = "0111") or (X = "1011") or (X = "1101") or (X = "1110") else
        "100" when X = "1111" else
        "---";
end Dataflow;
```

Simulazione

In figura 44 è riportata la simulazione della macchina combinatoria.

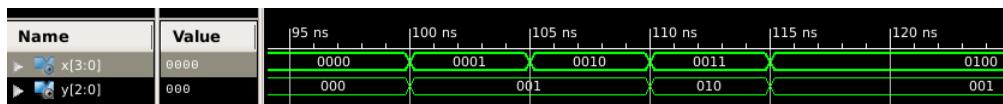


Figura 44: Simulazione macchina combinatoria

RAM

Il valore letto dalla ROM va in ingresso alla macchina combinatoria, il cui output viene salvato in una memoria RAM di dimensione 16×3 . Anche per la RAM l'indirizzo è gestito dal contatore.

Entity e architecture - RAM

```
entity RAM is
    port ( clk  : in std_logic;
```

```

rst  : in std_logic;
we   : in std_logic;
en   : in std_logic;
addr : in std_logic_vector(3 downto 0);
di   : in std_logic_vector(2 downto 0);
do   : out std_logic_vector(2 downto 0));
end RAM;

architecture Behavioral of RAM is
    type ram_type is array (15 downto 0) of std_logic_vector (2 downto 0);
    signal RAM : ram_type;
begin
    process (clk)
    begin
        if (clk'event and clk = '1') then
            if rst = '1' then
                RAM <= (others => (others => '0'));
            elsif en = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                else
                    do <= RAM(conv_integer(addr));
                end if;
            end if;
        end if;
    end process;
end Behavioral;

```

Simulazione

In figura 45 è mostrata la simulazione della RAM utilizzata. In particolare, alzato il segnale di *WE*, a ogni colpo di clock la RAM memorizza il dato in ingresso all'indirizzo specificato.

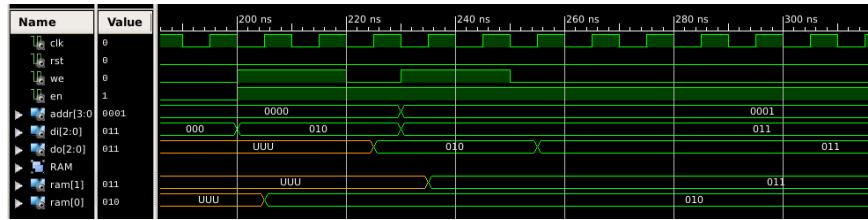


Figura 45: Simulazione RAM

Contatore

Per scandire automaticamente la sequenza da testare è stato progettato un contatore modulo 16, che fornisce gli indirizzi in ingresso alla ROM e alla RAM.

Entity e architecture - contatore

```
entity Counter_address is
    Port ( clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        enable : in STD_LOGIC;
        counter : out STD_LOGIC_VECTOR (3 downto 0);
        e : out std_logic := '1');
end Counter_address;

architecture Behavioral of Counter_address is
    signal c : std_logic_vector (3 downto 0) := (others => '0');
begin
    counter <= c;
    counter_process: process(clock)
    begin
        if(rising_edge(clock)) then
            if reset = '1' then
                e <= '1';
                c <= (others => '0');
            elsif enable = '1' then
                if(c = "1111") then
                    e <= '0';
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

```

        c <= std_logic_vector(unsigned(c) + 1);
    end if;
end if;
end process;
end Behavioral;
```

Per tempificare opportunamente le operazioni di lettura dalla ROM, scrittura nella RAM e successiva lettura da essa, è stata utilizzata una base dei tempi in grado di fornire segnali di clock a due frequenze differenti.

Clock filter

Per fare in modo che il valore letto dalla ROM all'indirizzo i sia fornito alla rete combinatoria e l'output sia scritto in RAM al medesimo indirizzo i si fa in modo che gli indirizzi siano generati dal counter con una frequenza minore di quella del clock. Il valore è scelto in modo tale che la rete combinatoria abbia il tempo per calcolare la propria uscita e far trascorrere il tempo di setup e di hold. Inoltre, facciamo in modo che la RAM riesca a scrivere il valore generato al medesimo indirizzo.

Entity e architecture - clock filter

```

entity clock_filter is
    generic ( CLKIN_freq : integer := 50000000;
              CLKOUT_freq1 : integer := 10000000;
              CLKOUT_freq2 : integer := 1);

    Port ( clock_in : in STD_LOGIC;
           reset : in STD_LOGIC;
           clock_out1 : out STD_LOGIC;
           clock_out2 : out STD_LOGIC);
end clock_filter;

architecture Behavioral of clock_filter is
    signal clockfx1 : std_logic := '0';
    signal clockfx2 : std_logic := '0';
    constant count_max_value1 : integer := CLKIN_freq/(CLKOUT_freq1) - 1;
    constant count_max_value2 : integer := CLKIN_freq/(CLKOUT_freq2) - 1;
```

```

begin
    clock_out1 <= clockfx1;
    clock_out2 <= clockfx2;
    count_for_division1: process(clock_in)
        variable counter1 : integer range 0 to count_max_value1 := 0;

        begin
            if (clock_in'event and clock_in = '1') then
                if( reset = '1') then
                    counter1 := 0;
                    clockfx1 <= '0';
                else
                    if counter1 = count_max_value1 then
                        clockfx1 <= '1';
                        counter1 := 0;
                    else
                        clockfx1 <= '0';
                        counter1 := counter1 + 1;
                    end if;
                end if;
            end if;
        end process;

    count_for_division2: process(clock_in)
        variable counter2 : integer range 0 to count_max_value2 := 0;

        begin
            if (clock_in'event and clock_in = '1') then
                if( reset = '1') then
                    counter2 := 0;
                    clockfx2 <= '0';
                else
                    if counter2 = count_max_value2 then
                        clockfx2 <= '1';
                        counter2 := 0;
                    else
                        clockfx2 <= '0';
                    end if;
                end if;
            end if;
        end process;

```

```

        counter2 := counter2 + 1;
    end if;
end if;
end if;
end process;
end Behavioral;

```

Sistema di testing

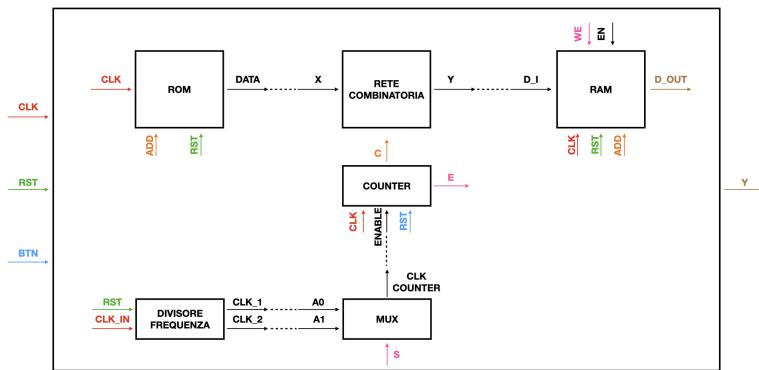


Figura 46: Schema Testing System

La composizione interna dell'intero sistema di testing è riportata in figura 46.

Entity e architecture - sistema di testing

```

entity Test_system is
    port ( CLK : in std_logic;
           RST : in std_logic;
           BTN_READ : in std_logic;
           Y : out std_logic_vector(2 downto 0));
end Test_system;

architecture Behavioral of Test_system is

```

```

signal address : std_logic_vector(3 downto 0) := (others => '0');
signal u : std_logic_vector(3 downto 0) := (others => '0');
signal t : std_logic_vector(2 downto 0) := (others => '0');
signal sel : std_logic := '1';
signal clk_counter : std_logic;
signal clocks : std_logic_vector(0 to 1);

component mux_2_1 is
    port ( s : in std_logic;
           a0, a1 : in std_logic;
           y : out std_logic);
end component;

component ROM is
    port ( CLK : in std_logic;
           RST : in std_logic;
           EN : in std_logic;
           ADDR : in std_logic_vector(3 downto 0);
           DATA : out std_logic_vector(3 downto 0));
end component;

component RAM is
    port ( clk : in std_logic;
           rst : in std_logic;
           we : in std_logic;
           en : in std_logic;
           addr : in std_logic_vector(3 downto 0);
           di : in std_logic_vector(2 downto 0);
           do : out std_logic_vector(2 downto 0));
end component;

component clock_filter is
    port ( clock_in : in STD_LOGIC;
           reset : in STD_LOGIC;
           clock_out1 : out STD_LOGIC;
           clock_out2 : out STD_LOGIC);
end component;

```

```

component Macchina_combinatoria is
    port ( X : in std_logic_vector(3 downto 0);
           Y : out std_logic_vector(2 downto 0));
end component;

component Counter_address is
    port ( clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           enable : in STD_LOGIC;
           counter : out STD_LOGIC_VECTOR (3 downto 0);
           e : out std_logic);
    end component;

begin
    ROM_0: ROM
        port map ( CLK => CLK,
                   RST => RST,
                   EN => '1',
                   ADDR => address,
                   DATA => u);

    Macchina_combinatoria_0 : Macchina_combinatoria
        port map ( X => u,
                   Y => t);

    mux_0 : mux_2_1
        port map ( s => sel,
                   a0 => clocks(1),
                   a1 => clocks(0),
                   y => clk_counter);

    clock_filter_0 : clock_filter
        port map ( clock_in => CLK,
                   reset => RST,
                   clock_out1 => clocks(0),
                   clock_out2 => clocks(1));

    RAM_0 : RAM
        port map ( clk => CLK,

```

```

        rst => RST,
        we => sel,
        en => '1',
        addr => address,
        di => t,
        do => Y);

Counter_address_0 : Counter_address
    port map ( clock => CLK,
    reset => BTN_READ,
    enable => clk_counter,
    counter => address,
    e => sel);
end Behavioral;

```

Simulazione

Allo scopo di visualizzare tramite simulazione, mostrata in figura 47, i valori in lettura, è stata modificata la frequenza del secondo clock, alzandola da 1 Hz a 25 MHz.

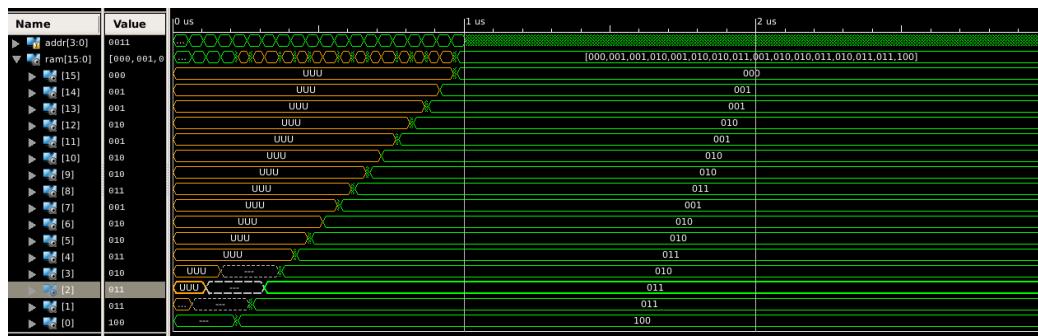


Figura 47: Simulazione sistema di testing

Parte II

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

Sintesi

Il contatore è dotato di un'opportuna uscita, che serve ad abilitare la RAM alla lettura, fermando così le scritture. Lo stesso segnale seleziona con un multiplexer uno dei due clock in ingresso al contatore. Quando questo raggiunge il valore massimo, cioè quando sono stati generati e scritti in RAM gli output di tutte le sequenze, il segnale permette di cambiare frequenza al clock (1 Hz), riuscendo così a visualizzare opportunamente l'output sui led della board. Tutto ciò è già riportato nello schema in figura 46. L'intera procedura di testing è avviata da un opportuno pulsante. Il risultato è osservabile dalla sintesi su board riportata in figura 48.

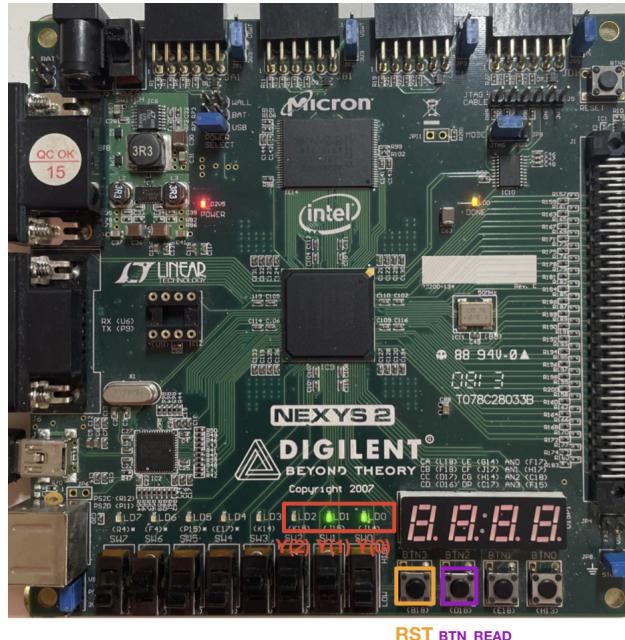


Figura 48: Sintesi sistema testing su board

Comunicazione con handshaking

Parte I

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

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

Handshaking

Per la realizzazione del sistema adottiamo un approccio strutturale. Iniziamo con l'individuare l'entità trasmettitore e ricevitore. Per ciascuna di esse separiamo la parte operativa da quella di controllo.

Il trasmettitore contiene i seguenti elementi:

- Unità di controllo: serve a effettuare l'handshaking e incrementare il contatore
- Contatore: serve a scandire la ROM e a indicare la fine delle operazioni

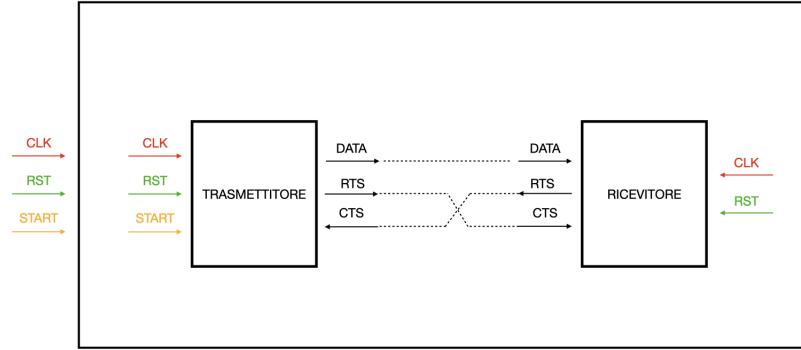


Figura 49: Architettura complessiva

- ROM: contiene le stringhe da trasmettere

Il nodo ricevitore contiene i seguenti elementi:

- Unità di controllo: serve a effettuare l'handshaking, incrementare il contatore e abilitare le scritture in RAM
- Contatore: fornisce l'indirizzo alla ROM e alla RAM
- ROM: contiene le stringhe da sommare
- RAM: per memorizzare la somma delle stringhe

Abbiamo utilizzato un protocollo di handshaking tra le due entità. In particolare il trasmettitore alza il segnale RTS e fornisce il dato. Il ricevitore effettua la somma, memorizza il risultato nella RAM e sblocca il trasmettitore per la successiva trasmissione.

Schema complessivo

In figura 49 è rappresentata l'architettura complessiva del sistema. Le entità trasmettitore e ricevitore sono mostrate rispettivamente in figura 50 e 51.

UC trasmettitore

Entity e architecture - UC trasmettitore

Riportiamo la descrizione VHDL dell'unità di controllo del trasmettitore e del relativo automa in figura 52.

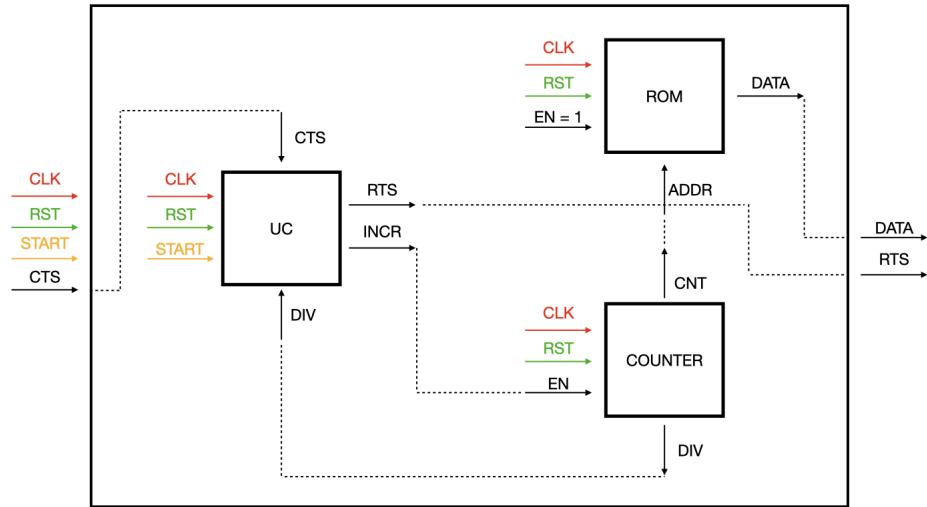


Figura 50: Trasmettitore

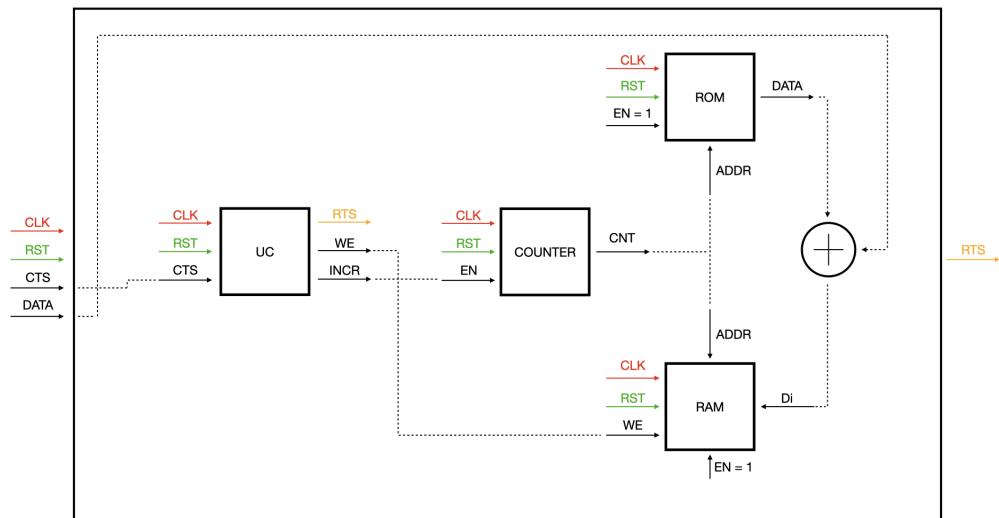


Figura 51: Ricevitore

- Idle: il trasmettitore attende che sia fornito il segnale di start per avviare la trasmissione
- Q0: il trasmettitore alza il segnale RTS, fornisce il dato e attende che il ricevitore termini l'elaborazione. Appena CTS si alza il trasmettitore cambia stato e incrementa il contatore per scandire la ROM
- Q1: viene abbassato il segnale di conteggio e RTS per poter ristabilire l'handshaking. Nel caso in cui il contatore indichi la fine delle trasmissioni si ritorna nello stato di idle, altrimenti si ripartirà con la prossima trasmissione

```

entity ucA is
    port ( CLK : in std_logic;
           start : in std_logic;
           RST : in std_logic;
           DIV : in std_logic;
           incr : out std_logic;
           RTS : out std_logic;
           CTS : in std_logic);
end ucA;

architecture Behavioral of ucA is
    type stato is (idle, q0, q1);
    signal stato_corr : stato := idle;
begin
    proc : process(CLK)
    begin
        if (rising_edge(CLK)) then
            case stato_corr is
                when idle =>
                    incr <= '0';
                    if (start = '1') then
                        stato_corr <= q0;
                    end if;
                when q0 =>
                    RTS <= '1';
                    if (CTS = '1') then
                        incr <= '1';
                    end if;
            end case;
        end if;
    end process;
end Behavioral;

```

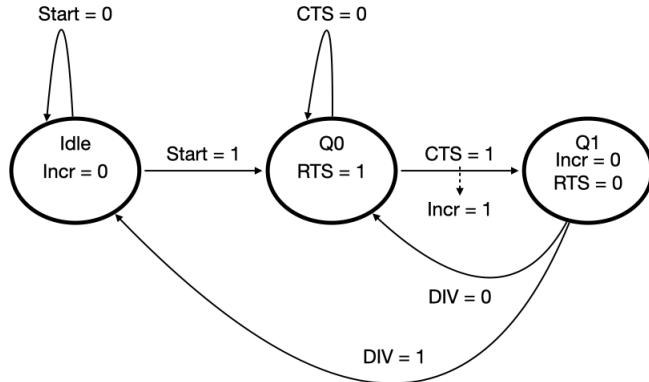


Figura 52: Automa trasmettitore

```

        stato_corr <= q1;
    end if;
when q1 =>
    incr <= '0';
    RTS <= '0';
    if (DIV = '1') then
        stato_corr <= idle;
    else
        stato_corr <= q0;
    end if;
when others =>
    stato_corr <= idle;
end case;
end if;
end process;
end Behavioral;
```

UC ricevitore

Entity e architecture - UC ricevitore

Riportiamo la descrizione VHDL dell'unità di controllo del ricevitore e del relativo automa in figura 53.

- Q0: il ricevitore attende che il trasmettitore dia inizio al protocollo di handshaking, ossia che CTS si alzi
- Q_wait: bisogna attendere un ciclo di clock per effettuare la somma tra la stringa trasmessa e quella nella ROM del ricevitore
- Q1: inizia la scrittura in RAM, alzando il segnale di write enable
- Q2: terminata la scrittura in RAM, si abbassa il segnale di write enable, si incrementa il contatore e si comunica al trasmettitore la fine dell'elaborazione, alzando il segnale RTS

```

entity ucB is
    port ( CLK : in std_logic;
           we : out std_logic;
           RST : in std_logic;
           incr : out std_logic;
           RTS : out std_logic;
           CTS : in std_logic);
end ucB;

architecture Behavioral of ucB is
    type stato is (q0, q1, q_wait, q2);
    signal stato_corr : stato := q0;
begin
    proc : process(CLK)
    begin
        if (rising_edge(CLK)) then
            case stato_corr is
                when q0 =>
                    RTS <= '0';
                    incr <= '0';
                    if (CTS = '1') then
                        stato_corr <= q_wait;
                    end if;
                when q_wait =>
                    stato_corr <= q1;
                when q1 =>
                    we <= '1';
            end case;
        end if;
    end process;
end;

```

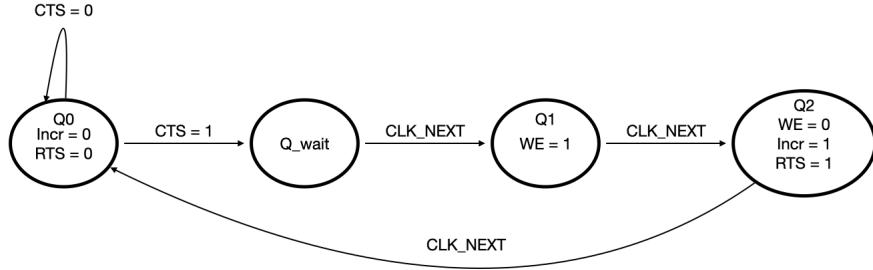


Figura 53: Automa ricevitore

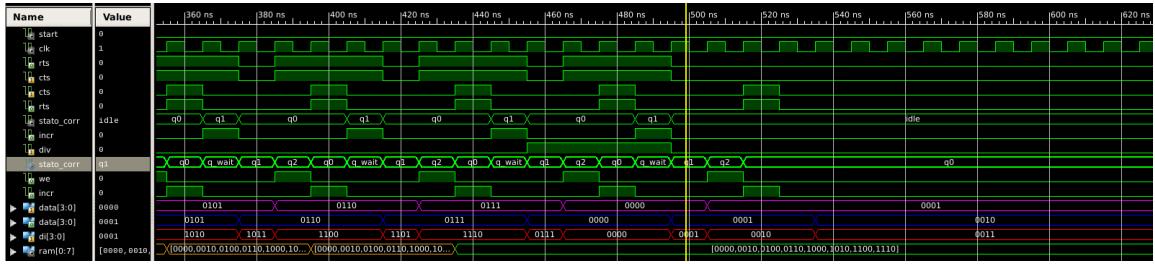


Figura 54: Simulazione handshaking

```

      stato_corr <= q2;
when q2 =>
  incr <= '1';
  we <= '0';
  RTS <= '1';
  stato_corr <= q0;
when others =>
  stato_corr <= q0;
end case;
end if;
end process;
end Behavioral;
  
```

Simulazione - Schema complessivo

In figura 54 è riportata la simulazione della comunicazione tra le due entità.

Processore

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.*

Analisi architettura

Introduzione

Il livello microarchitetturale di un calcolatore ha il compito di implementare il livello ISA sopra di esso. La progettazione di tale livello è strettamente legata all'ISA considerata. In questa trattazione si fa riferimento all'ISA IJVM. La nostra microarchitettura conterrà un microprogramma in una ROM, il cui compito è quello di prelevare, decodificare ed eseguire istruzioni IJVM. Si tenga a mente come il processore che andremo a realizzare è a stack, ossia non ha registri generali, ma le istruzioni avranno degli operandi impliciti. A seguire viene mostrata la microarchitettura e come è controllata dalle microistruzioni, ciascuna delle quali controlla il Data Path per un ciclo. La sequenza delle microistruzioni che implementano un'istruzione ISA forma una microprocedura. L'insieme delle sequenze costituisce il microprogramma.

Data Path

In figura 55 è mostrato il Data Path della nostra microarchitettura. Essa contiene registri a 32 bit, alcuni dei quali sono cablati, ossia hanno una

specifica funzione, come MAR, MDR, PC, MBR e H. Gli altri hanno un nome in base all'utilizzo che se ne fa nel microprogramma, ma non sono dedicati a uno scopo.

Si tenga a mente come questi registri siano visibili a livello microarchitetture, ma non fanno parte del modello di programmazione. La maggior parte dei registri può mandare il proprio contenuto sul bus B, che alimenta l'operando di destra dell'ALU. Sono esclusi il MAR, che può essere scritto solo attraverso il bus C e il registro H, che è dedicato a essere l'operando sinistro dell'ALU. Un solo registro alla volta può essere abilitato a scrivere sul bus B. Più registri possono leggere contemporaneamente il contenuto del bus C, ossia l'uscita dell'ALU.

In uscita all'ALU è presente uno shifter, che con due segnali di controllo può effettuare uno shift del dato.

L'ALU riceve in ingresso 6 segnali di controllo, che ne determinano l'operazione. In figura 56 sono mostrate le combinazioni utili.

È possibile in maniera esplicita leggere e scrivere lo stesso registro in un solo ciclo. Per esempio è permesso mettere SP sul bus B, disabilitare l'input sinistro dell'ALU, abilitare INC e memorizzare il risultato in SP, cioè incrementarlo di 1. Affinchè ciò sia possibile senza produrre risultati errati, bisogna lavorare in momenti differenti del clock. In particolare, quando un registro è selezionato come input destro dell'ALU, il suo valore è posto sul bus B all'inizio del ciclo. L'ALU fa il suo lavoro, producendo un risultato che tramite lo shifter arriva sul bus C. Quasi alla fine del ciclo, quando l'uscita dell'ALU e dello shifter è stabile, il segnale di clock *triggera* la memorizzazione del contenuto del bus C in uno o più registri, tra cui potrebbe esserci anche quello che ha alimentato il bus B con il proprio input. Con un'opportuna tempificazione del Data Path e sfruttando entrambi i fronti del clock è possibile quindi leggere e scrivere lo stesso registro in un ciclo.

Tempificazione del Data Path

In figura 57 è mostrato il timing degli eventi descritti sopra. Sul fronte di discesa del clock sono impostati i bit che dovranno pilotare tutte le porte. Questo richiede un tempo finito Δw . A questo punto il registro di cui si ha bisogno sul bus B è selezionato e posto su di esso. Prima che il valore sia stabile è richiesto un tempo Δx . A questo punto l'ALU e lo shifter, che sono circuiti combinatori, che hanno finora lavorato su dati errati, hanno input validi. Dopo un tempo Δy l'uscita dell'ALU e dello shifter è stabile. Dopo un

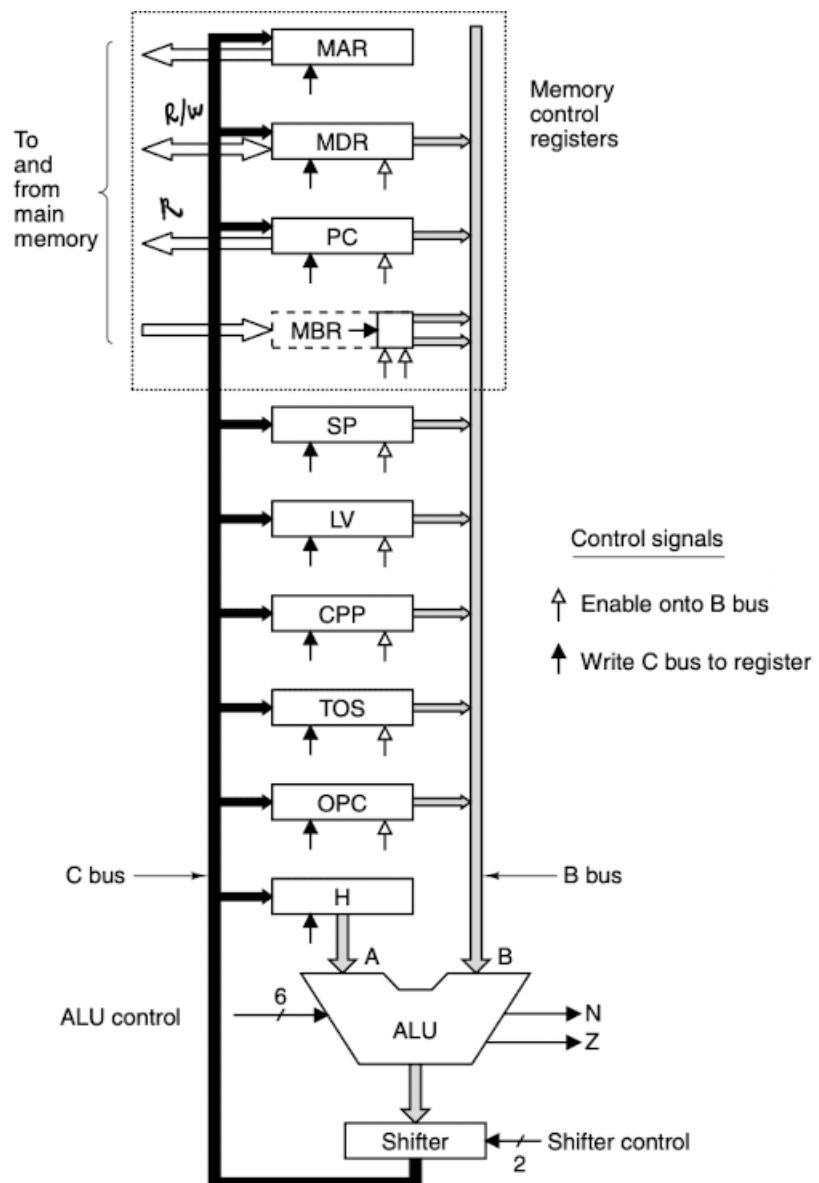


Figura 55: Data Path

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	\bar{A}
1	0	1	1	0	0	\bar{B}
1	1	1	1	0	0	$A + B$
1	1	1	1	0	1	$A + B + 1$
1	1	1	0	0	1	$A + 1$
1	1	0	1	0	1	$B + 1$
1	1	1	1	1	1	$B - A$
1	1	0	1	1	0	$B - 1$
1	1	1	0	1	1	$-A$
0	0	1	1	0	0	$A \text{ AND } B$
0	1	1	1	0	0	$A \text{ OR } B$
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	-1

Figura 56: Operazioni ALU

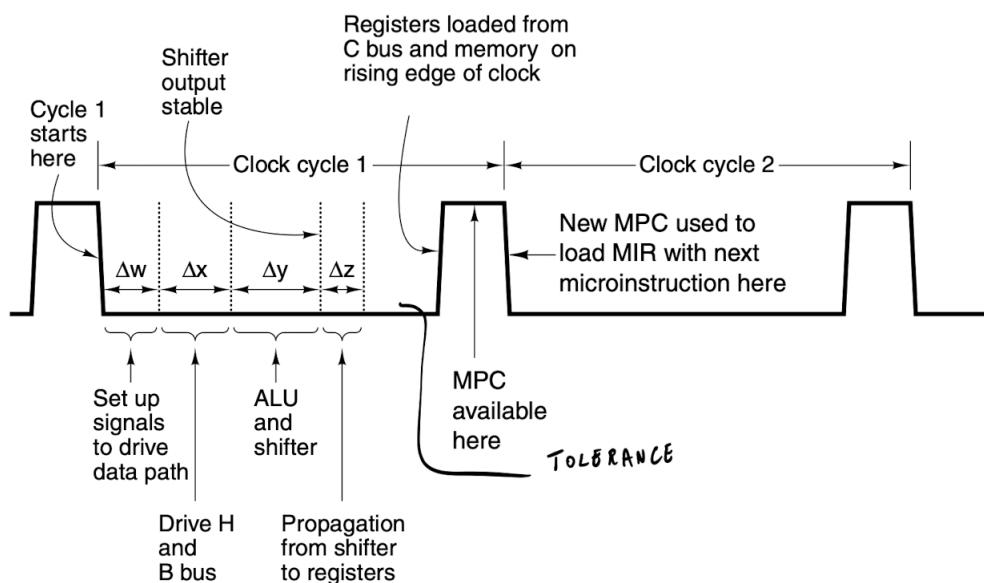


Figura 57: Timing di un ciclo del Data Path

altro tempo Δz il risultato si è propagato sul bus C e ha raggiunto i registri, che possono essere caricati sul fronte di salita del clock. È importante che il caricamento dei registri avvenga velocemente in corrispondenza del fronte. Sempre sullo stesso fronte di salita il registro che alimentava il bus B smette di farlo in preparazione per il prossimo ciclo.

Si noti come anche se non ci siano elementi di memorizzazione nel Data Path è presente un tempo di propagazione finito in esso. Quindi anche se dovessimo modificare con una store il valore di uno dei registri di input, non c'è rischio che tale valore possa raggiungere l'ALU.

Affinchè tutto funzioni bisogna progettare attentamente la temporizzazione, la durata del ciclo di clock, conoscere i tempi di propagazione nel Data Path e caricare rapidamente i registri del bus C.

Possiamo quindi pensare al ciclo del Data Path come una sequenza di sottocicli, dove l'inizio del sottociclo 1 avviene in corrispondenza del fronte di discesa del clock.

1. I segnali di controllo sono configurati (set up) $\rightarrow \Delta w$
2. Il registro è caricato sul bus B $\rightarrow \Delta x$
3. L'ALU e lo shifter operano $\rightarrow \Delta y$
4. Il risultato si propaga sul bus C fino ai registri $\rightarrow \Delta z$

Il tempo dopo Δz fornisce una soglia di tolleranza. Sul fronte di salita del prossimo ciclo di clock il risultato sarà memorizzato nei registri.

I sottocicli possono essere pensati come impliciti. Questo perchè in realtà non ci sono segnali di clock o segnali esplicativi che dicono all'ALU quando operare o quando scrivere il risultato sul bus C. In realtà l'ALU e lo shifter lavorano sempre e producono valori *garbage* fino a un tempo $\Delta w + \Delta x$ dopo il fronte di discesa del clock. L'output di essi sarà invalido fino a dopo un tempo $\Delta w + \Delta x + \Delta y$. L'unico segnale esplicito che guida il Data Path è il clock e in particolare il fronte di discesa, che ne dà inizio, e il fronte di salita, che triggerà il caricamento nei registri dal bus C. È responsabilità del progettista fare in modo che il tempo $\Delta w + \Delta x + \Delta y + \Delta z$ termini in anticipo rispetto al fronte di salita del clock in modo da caricare i registri sempre con valori validi.

Memoria

La nostra macchina ha due modi per comunicare con la memoria: un'interfaccia in lettura e in scrittura a 32 bit word-addressable e un'interfaccia a 8 bit byte-addressable in sola lettura. L'interfaccia a 32 bit è controllata da due registri: MAR e MDR. L'interfaccia a 8 bit è gestita da un registro: PC, che legge 1 byte negli 8 bit meno significativi di MBR. Si sottolinea come tale interfaccia possa solamente leggere dati dalla memoria e non scrivere verso di essa. L'interfaccia a 32 bit è responsabile del prelievo degli operandi mentre quella a 8 bit gestisce il prelievo delle istruzioni. Ognuno di questi registri è pilotato da due segnali di controllo (anche gli altri registri in figura 55). La freccia grigia sotto il registro indica un segnale di controllo che abilita l'output del registro sul bus B. Poichè MAR non si connette con tale bus non presenta questo segnale di abilitazione. H allo stesso modo non possiede una connessione con il bus B, essendo dedicato a essere l'unico possibile input di sinistra dell'ALU ed è per questo sempre abilitato. La freccia nera sotto il registro indica un segnale di controllo che scrive nel registro il contenuto del bus C. Poichè MBR non può essere caricato dal bus C non possiede un segnale di write. Per iniziare una lettura o scrittura con la memoria bisogna caricare opportunamente i registri e poi fornirle un segnale di lettura o scrittura.

MAR contiene indirizzi di word, in modo che i valori 0, 1, 2 fanno riferimento a word consecutive in memoria. PC contiene invece indirizzi di byte, in modo che 0, 1, 2 fanno riferimento a byte consecutivi. Quindi mettere 2 in PC e poi iniziare una lettura in memoria porterà a leggere il byte 2 dalla memoria, che sarà messo negli 8 bit meno significativi di MBR. Mettendo 2 in MAR e avviando una lettura dalla memoria porterà a leggere i byte 8-11 (la word 2), che saranno messi in MDR. Tale differenza deriva dal fatto che MAR e PC sono usati per far riferimento a due diverse parti della memoria. Per ora si tenga a mente solo che la coppia MAR/MDR viene usata per leggere e scrivere word di dati di livello ISA mentre la coppia PC/MBR è usata per leggere un programma eseguibile di livello ISA (istruzioni).

Nell'implementazione fisica reale c'è solo una memoria ed è byte oriented. Permettere al MAR di contare word mentre la memoria fisica conta in byte sfrutta un semplice trucco. Quando il valore in MAR è posto sull'address bus, i suoi 32 bit non si mappano direttamente con le 32 linee di indirizzo 0-31. Il bit 0 di MAR viene collegato alla linea 2 del bus, il bit 1 alla linea 3 e così via. I due bit più significativi di MAR sono scartati visto che servono

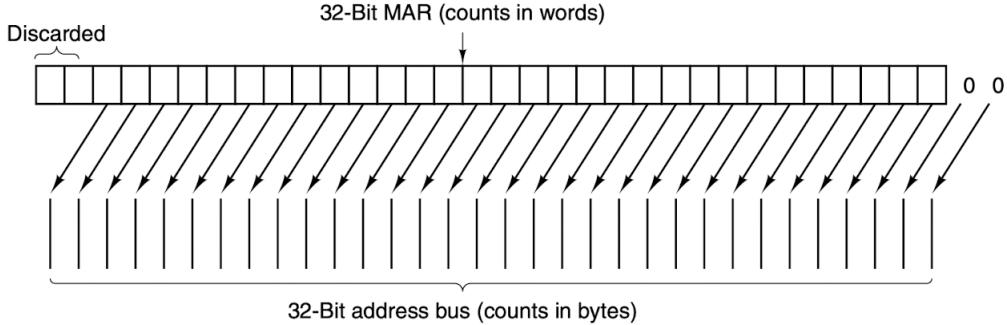


Figura 58: Mapping MAR e address

solo per indirizzi di word che superano 2^{32} , non legali per la nostra macchina a 4 GB. Con questo mapping, quando MAR è 1, l'indirizzo 4 viene posto sull'address bus, quando MAR è 2 viene posto l'indirizzo 8 sul bus e così via. Questo meccanismo è illustrato in figura 58.

I dati letti dalla memoria tramite l'interfaccia a 8 bit sono posti nel registro a 8 bit MBR, che può essere copiato sul bus B in due modi possibili: unsigned o signed. Quando serve il valore unsigned, la word posta sul bus B contiene negli 8 bit meno significativi MBR e tutti 0 nelle restanti posizioni. L'altra opzione è trattare gli 8 bit di MBR come un valore con segno tra -128 e 127 e usare tale valore per generare una word a 32 bit con lo stesso valore numerico. Questo è fatto duplicando il bit più significativo di MBR nei restanti 24 bit del bus. Si parla in tal caso di sign extension e i 24 bit più significativi saranno o tutti 0 o tutti 1 a seconda del bit più significativo degli 8 di MBR. La scelta tra signed e unsigned avviene con due differenti segnali di controllo ed ecco perchè in figura 55 sono presenti due segnali di controllo per la scrittura di MBR verso B.

Microistruzioni

Per controllare il Data Path in figura 55 abbiamo bisogno di 29 segnali divisi in 5 gruppi funzionali:

- 9 segnali per controllare la scrittura dal bus C verso i registri
- 9 segnali per controllare l'abilitazione dei registri verso il bus B per l'ingresso destro dell'ALU

- 8 segnali per controllare l'ALU e lo shifter
- 2 segnali (non mostrati) per indicare lettura o scrittura con MAR/MDR
- 1 segnale (non mostrato) per indicare il prelievo dalla memoria con PC/MBR

Questi 29 segnali di controllo specificano le operazioni da compiere per un ciclo del Data Path. Un ciclo consiste nel portare il valore di uno dei registri sul bus B, propagare il contenuto attraverso l'ALU e lo shifter, portare l'uscita sul bus C e infine scriverla nei registri appropriati. In più se è alto il segnale di memory read data, l'operazione in memoria inizia alla fine alla fine del ciclo del Data Path, dopo che MAR è stato caricato. I dati dalla memoria saranno disponibili soltanto alla fine del ciclo seguente in MBR o MDR e possono essere usati soltanto nel ciclo successivo. In altre parole una lettura in memoria iniziata alla fine del ciclo k restituisce un dato che non può essere usato nel ciclo $k+1$, ma soltanto da quello $k+2$ in poi. Questo comportamento controintuitivo è spiegato in figura 57. I segnali di controllo della memoria non sono generati nel ciclo di clock 1 fino a dopo che MAR e PC sono caricati sul fronte di salita del clock, ossia alla fine del ciclo 1. Assumiamo che la memoria restituisce il risultato sul bus dati dopo un ciclo in modo che MBR e/o MDR possono essere caricati al prossimo fronte di salita, insieme agli altri registri. In altre parole noi carichiamo MAR alla fine del ciclo del Data Path e iniziamo l'operazione verso la memoria poco dopo. Dunque non possiamo aspettarci il risultato dell'operazione di lettura nel registro MDR all'inizio del ciclo successivo, specialmente se l'impulso del clock è stretto. Non c'è abbastanza tempo se la memoria impiega un colpo di clock. Deve quindi per forza esserci un ciclo del Data Path tra l'avvio dell'operazione verso la memoria e l'utilizzo del risultato. Ovviamente possiamo compiere altre operazioni nel frattempo, che però non usino il dato atteso dalla memoria. L'assunzione poi che la memoria restituisca il dato a ogni colpo di clock significa che abbiamo un hit rate del 100%. Ovviamente tale assunzione non è mai vera, ma semplifica notevolmente il modello.

Poichè MBR e MDR sono caricati sul fronte di salita del clock insieme agli altri registri, potrebbero essere letti durante i cicli quando sta avvenendo una nuova lettura in memoria. Tali registri ritorneranno i vecchi valori visto che la lettura non ha avuto ancora tempo di sovrascriverli. Non abbiamo ambiguità. Finché non saranno caricati i nuovi valori in MBR e MDR sul fronte di clock successivo i valori precedenti sono ancora presenti e utilizzabili.

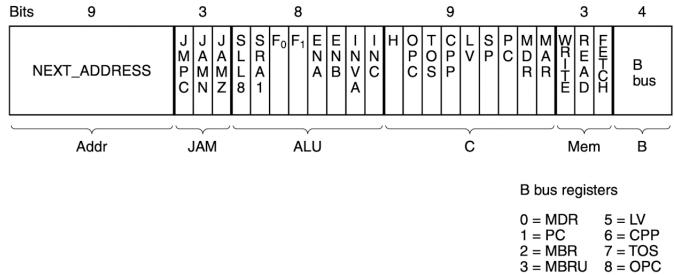


Figura 59: Formato microistruzione Mic-1

È possibile effettuare due letture in due cicli consecutivi, visto che la lettura impiega proprio un ciclo. Inoltre, entrambe le memorie operano allo stesso tempo. Però provare a leggere e scrivere contemporaneamente lo stesso byte porta a risultati indefiniti.

Mentre è desiderabile scrivere il valore sul bus C su più registri contemporaneamente, lo stesso non vale per il bus B. Soltanto uno dei registri può essere abilitato a scrivere il valore sul bus B. Possiamo quindi aumentare leggermente la circuiteria, andando a ridurre il numero di bit richiesti per scegliere tra le possibili sorgenti quella che andrà sul bus B, dove la versione signed e unsigned di MBR sono separate. Possiamo codificare questi 9 segnali di controllo con 4 bit e usiamo un decoder per generare 16 segnali di controllo, di cui 7 non servono.

Possiamo controllare il Data Path con $9 + 4 + 8 + 2 + 1 = 24$ segnali, quindi 24 bit. Questi segnali controllano il Data Path soltanto per un ciclo. La seconda parte del controllo è determinare cosa fare il ciclo successivo. Per includere questo aspetto utilizziamo un formato di microistruzione contenente i 24 segnali di controllo per un ciclo più due campi addizionali: *NEXT_ADDRESS* e *JAM*.

In figura 59 è mostrato il possibile formato della microistruzione, divisa in 6 gruppi e contenente i seguenti 36 segnali:

- **Addr:** contiene l'indirizzo di una potenziale microistruzione successiva
- **JAM:** determina come sarà scelta la prossima microistruzione
- **ALU:** operazioni ALU e shifter
- **MEM:** funzioni di memoria

- B: seleziona la sorgente del bus B, codificata come in figura 59

L'ordine dei campi della microistruzione è arbitrario, ma con la scelta fatta minimizza l'intreccio dei fili.

Unità di controllo

Abbiamo visto come il Data Path è controllato, ma non abbiamo ancora spiegato come stabilire quali segnali di controllo abilitare a ogni ciclo. Questo è determinato da un **sequencer** che ha la responsabilità di generare la sequenza di microistruzioni per l'esecuzione di una singola istruzione ISA. In particolare il sequencer produce due tipi di informazioni a ogni ciclo:

1. Lo stato di ogni segnale di controllo nel sistema
2. L'indirizzo della prossima microistruzione da eseguire

In figura 60 è mostrata la microarchitettura completa della nostra macchina, che chiamiamo **Mic-1**. Sulla sinistra è presente il Data Path mentre sulla destra è presente la parte che si occupa del controllo. L'oggetto più grande e importante nella parte di controllo è una memoria chiamata **control store**. Possiamo pensarla come una memoria che contiene l'intero microprogramma, anche se a volte è implementata come delle porte logiche.

Dal punto di vista funzionale il control store è una memoria, che invece di contenere istruzioni ISA, contiene microistruzioni. Nel nostro caso contiene 512 word, ognuna delle quali è una microistruzione di 36 bit. In realtà non tutte le word sono necessarie, ma per alcuni motivi spiegati a breve abbiamo bisogno di 512 word differenti.

Rispetto a una memoria classica in cui le istruzioni sono eseguite sequenzialmente a meno di salti, questo non vale per le microistruzioni. I microprogrammi necessitano di maggiore flessibilità (poichè le sequenze di microistruzioni tendono a essere brevi) e ogni microistruzione specifica il suo successore. Poichè il control store è funzionalmente una memoria (ROM), ha bisogno del suo memory address register e memory data register. Non necessita di segnali di lettura e scrittura perchè a ogni ciclo di clock viene letta. Il memory address register del control store è detto **MPC (MicroProgram Counter** mentre il memory data register è chiamato **MIR (MemoryInstruction Register**. La sua funzione è quella di contenere la corrente microistruzione, i cui bit pilotano i segnali di controllo del Data Path.

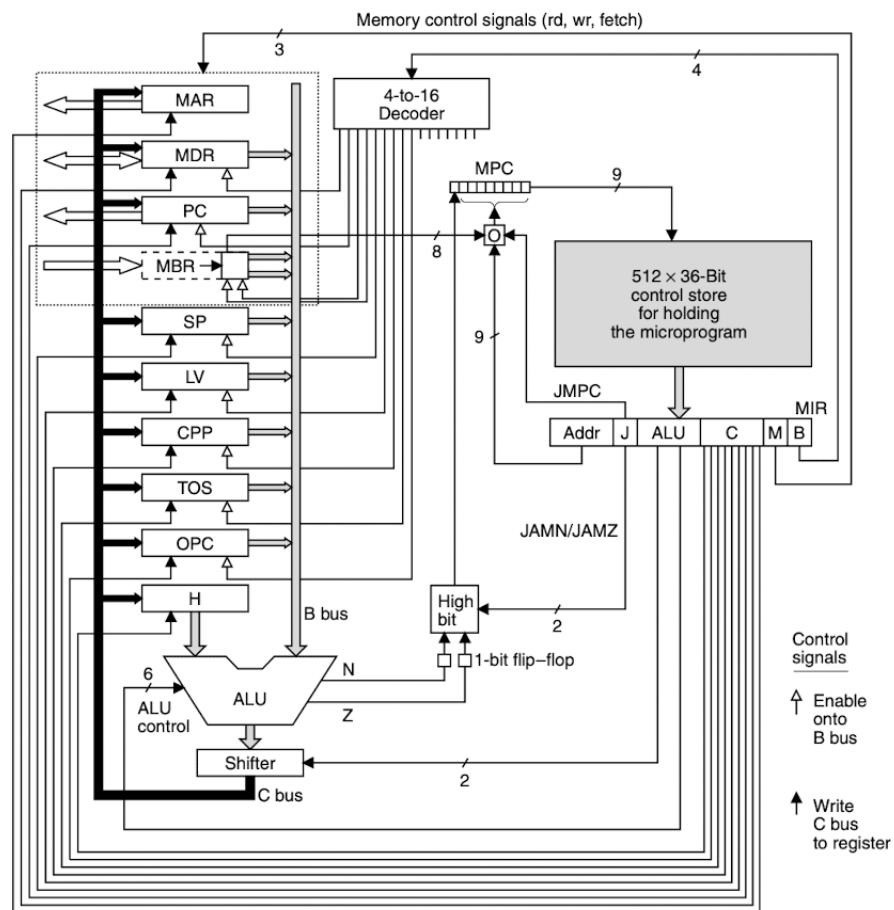


Figura 60: Schema completo microarchitettura

All'inizio di ogni ciclo di clock (fronte di discesa) il MIR è caricato con la microistruzione del control store puntata dal MPC. In figura 57 il tempo di caricamento del MIR è indicato da Δw . In termini di sottociclo il MIR è caricato nel primo. Una volta che la microistruzione è posta nel MIR, i vari segnali si propagano verso il Data Path. Un registro sarà posto sul bus B, l'ALU sa quale operazione compiere. Questo è il secondo sottociclo. Dopo un intervallo $\Delta w + \Delta x$ dall'inizio del ciclo gli ingressi dell'ALU sono stabili. Un altro Δy dopo gli output dell'ALU: N e Z e dello shifter sono stabili. I valori di N e Z sono salvati in una coppia di flip-flop da un bit. Questi bit, come tutti gli altri registri che sono caricati dal bus C e dalla memoria, sono salvati sul fronte di salita del clock, verso la fine del ciclo del Data Path. L'attività dell'ALU e dello shifter avviene durante il sottociclo 3. Dopo un ulteriore intervallo Δz l'output dello shifter si è propagato sul bus C e ha raggiunto i registri, che saranno eventualmente caricati sul fronte di salita. Il sottociclo 4 consiste nel caricare i registri dal bus C, dalla memoria e caricare i flip-flop. Esso termina poco dopo il fronte di salita del clock e adesso siamo sicuri che tutti i risultati sono stati salvati e i risultati delle precedenti operazioni in memoria sono disponibili e il MPC viene caricato. Questo processo va avanti fino a quando la macchina viene spenta.

Oltre a pilotare il Data Path, parallelamente, il micropogramma deve anche determinare quale istruzione eseguire successivamente. Il calcolo di questo indirizzo comincia dopo che il MIR è stato caricato ed è stabile. Come prima cosa il campo NEXT_ADDRESS viene copiato in MPC. Mentre avviene questa copia, viene ispezionato il campo JAM. Se esso ha valore 000 non viene fatto niente. Quando termina la copia di NEXT_ADDRESS, MPC punterà alla prossima microistruzione. Se il bit JAMN è pari a 1 viene effettuata una OR tra il bit più significativo di MPC e il flip-flop di 1 bit contentente N. Allo stesso modo se JAMZ è alto viene fatta la OR con il flip-flop di Z. Se entrambi sono settati allora si fa la OR con entrambi i flip-flop. Abbiamo bisogno dei flip-flop perché dopo il fronte di salita del clock il bus B non è pilotato e le uscite dell'ALU non è detto siano valide. Salvando questi flag in N e Z siamo sicuri di avere i valori corretti e stabili per il calcolo di MPC, indipendentemente dall'ALU. La funzione che realizza questo calcolo in figura 60 è chiamata "High bit" e realizza:

$$F = (JAMZ \text{ AND } Z) \text{ OR } (JAMN \text{ AND } N) \text{ OR } \text{NEXT_ADDRESS}[8]$$

In tutti i casi MPC può assumere solo uno di due possibili valori:

1. Il valore di NEXT_ADDRESS
2. Il valore di NEXT_ADDRESS con il bit più significativo a 1 (se già era alto non ha senso usare JAMN o JAMZ)

Il terzo bit del campo JAM è JMPC. Se pari a 1, gli 8 bit di MBR sono posti in OR bit a bit con gli 8 bit meno significativi del campo NEXT_ADDRESS della corrente microistruzione. In figura 60 il blocco "O" realizza una OR tra MBR e NEXT_ADDRESS se JMPC è pari a 1, ma lascia passare NEXT_ADDRESS in MPC se JMPC è 0.

La OR tra MBR e NEXT_ADDRESS con la memorizzazione del risultato in MPC ci consente di avere un'efficiente implementazione di un salto. Si noti che tutti i 256 indirizzi possono essere specificati e determinati soltanto dai bit presenti in MBR. In un uso tipico MBR contiene un codice operativo, così l'uso di JMPC risulterà nella selezione unica della prossima microistruzione da essere eseguita per ognuno dei possibili codici operativi. Questo metodo è utile per effettuare velocemente un salto alla funzione corrispondente al codice operativo appena fetchato.

La temporizzazione della macchina è fondamentale. La spieghiamo di nuovo in termini di sottocicli per visualizzarla più facilmente, ma gli unici eventi reali di clock sono il fronte di discesa, che inizia il ciclo, e il fronte di salita, che carica i registri e i flip-flop di N e Z.

Durante il sottociclo 1, iniziato dal fronte di discesa del clock, MIR è caricato dall'indirizzo attualmente in MPC. Durante il sottociclo 2 i segnali dal MIR si propagano e il bus B è caricato dal registro selezionato. Durante il sottociclo 3 l'ALU e lo shifter operano e producono un risultato stabile. Durante il sottociclo 4 i valori sul bus C, sul bus di memoria e i valori dell'ALU sono stabili. Sul fronte di salita i registri sono caricati dal bus C, N e Z sono caricati e MBR/MDR ricevono i risultati dall'operazione in memoria iniziata (eventualmente) alla fine del precedente ciclo di Data Path. Non appena MBR è disponibile, MPC è caricato in preparazione per la prossima microistruzione. MPC assume il proprio valore più o meno al centro dell'intervallo di quando il clock è alto, ma comunque dopo che MBR/MDR sono pronti. L'importante è che MPC non sia caricato prima che i registri da cui dipende (MBR, N e Z) sono pronti. Non appena il clock si abbassa, MPC può indirizzare il control store e inizia un nuovo ciclo. Si noti che ogni ciclo è "self contained", nel senso che specifica quello che va sul bus B, cosa devono fare ALU e shifter, dove il valore nel bus C deve essere memorizzato e infine quale deve essere il valore del prossimo MPC.

Un’ultima riflessione su MPC, che abbiamo trattato come un registro di 9 bit, caricato quando il clock è alto. In realtà non c’è bisogno di avere tale registro, ma tutti i suoi ingressi potrebbero essere direttamente posti in input al control store. L’importante è che essi siano validi sul fronte di discesa del clock per indirizzare la prossima microistruzione. Non c’è bisogno di memorizzarli in un registro. Dunque MPC potrebbe essere implementato come un **registro virtuale**, cioè un punto di raccolta per segnali (selezionatore). In questo modo si semplifica il timing: ora gli eventi avvengono solo sul fronte di discesa e salita del clock.

Analisi istruzioni

Abbiamo deciso di descrivere le microprecedure relative alle istruzioni:

- IADD
- DUP

Si noti che prima dell’esecuzione di qualunque programma IJVM c’è una fase di inizializzazione in cui viene eseguita la routine **mic1_entry**. Quest’ultima inizializza i registri del Data Path e la memoria locale del programma.

IADD

Per IADD si fa riferimento al seguente programma IJVM:

```
.main
.var
a
.endvar
BIPUSH 0xA
BIPUSH 0xE
IADD
ISTORE a
HALT
.endmethod
```

Di seguito è riportata la sequenza delle microistruzioni che implementano l’IADD in MAL e la relativa traduzione in segnali di controllo a opera del microassemblatore:

```

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

```

1. MAR = SP = SP - 1; rd

NEXT ADDRESS: 0x66

JMP: 000

ALU: 00110110

BUS C: 000001001

MEM: 010

BUS B: 0100

2. H = TOS

NEXT ADDRESS: 0x67

JMP: 000

ALU: 00010100

BUS C: 100000000

MEM: 000

BUS B: 0111

3. MDR = TOS = MDR + H; wr; goto main

NEXT ADDRESS: 0x6

JMP: 000

ALU: 00111100

BUS C: 001000010

MEM: 100

BUS B: 0000

In figura 61 è possibile osservare la simulazione dell'istruzione IADD. In particolare lo stack pointer viene decrementato al ciclo k e contemporaneamente avviene la lettura dell'operando puntato da SP-1. Al ciclo $k+1$ l'operando che era in cima allo stack, puntato da TOS, lo scriviamo nel registro H. Al ciclo $k+2$ l'operazione di lettura è terminata e il dato è presente nel registro MDR. Possiamo quindi procedere alla somma e a salvare il risultato nel registro TOS.

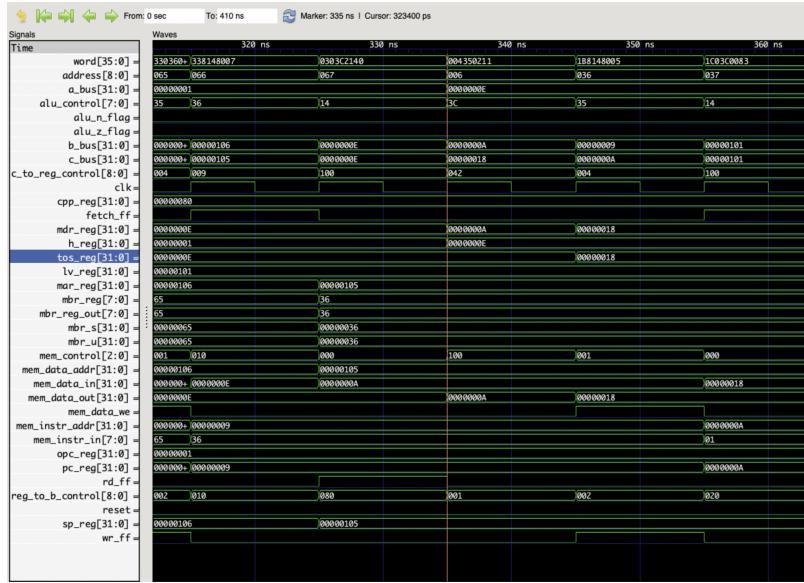


Figura 61: Simulazione IADD

DUP

Per DUP si fa riferimento al seguente programma IJVM:

```
.main
.var
a
.endvar
BIPUSH 0xA
DUP
IADD
ISTORE a
HALT
.endmethod

dup = 0x57;
MAR = SP = SP + 1;
MDR = TOS; wr; goto main
```

1. MAR = SP = SP + 1;
NEXT ADDRESS: 0x58

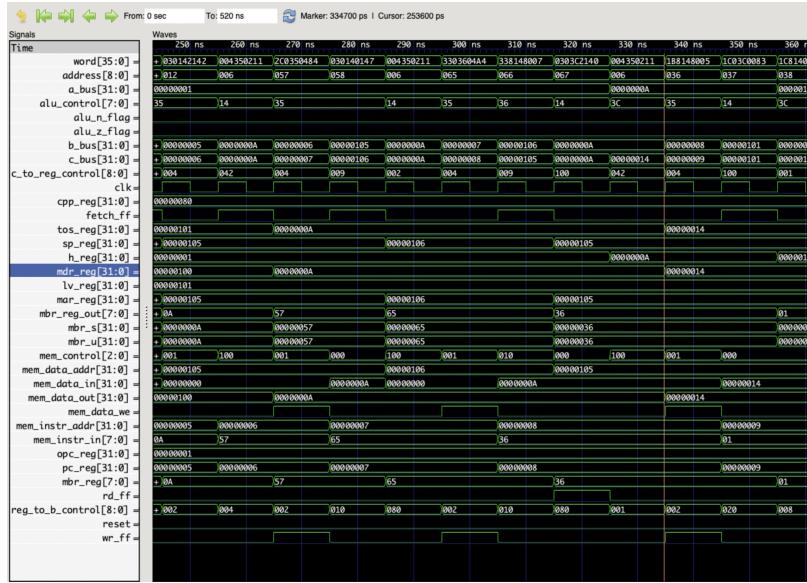


Figura 62: Simulazione DUP

JMP: 000
 ALU: 00110101
 BUS C: 000001001
 MEM: 000
 BUS B: 0100

- MDR = TOS; wr; goto main
 NEXT ADDRESS: 0x6

JMP: 000
 ALU: 00010100
 BUS C: 000000010
 MEM: 000
 BUS B: 0111

In figura 62 è possibile osservare la simulazione dell'istruzione DUP. Al ciclo k lo stack pointer viene incrementato di uno per riservare spazio al dato duplicato, memorizzando nel registro MAR il suo indirizzo. Al ciclo di clock successivo il valore in TOS viene scritto in cima allo stack. Con IADD si

effettua poi la somma dei due valori sullo stack, ottenendo il doppio del valore iniziale.

Modifica istruzione ISUB

Per ISUB si fa riferimento al seguente programma IJVM:

```
.main
.var
a
.endvar
BIPUSH 0xE
BIPUSH 0xA
ISUB
ISTORE a
HALT
.endmethod
```

Qui è riportata la sequenza delle microistruzioni che implementano l'ISUB non modificata in MAL:

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

Abbiamo modificato l'operazione di ISUB aggiungendo un'operazione di incremento unitario. La sequenza delle microistruzioni diventa:

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

1. MAR = SP = SP - 1; rd
NEXT ADDRESS: 0x5D
JMP: 000
ALU: 00110110
BUS C: 000001001
MEM: 010
BUS B: 0100

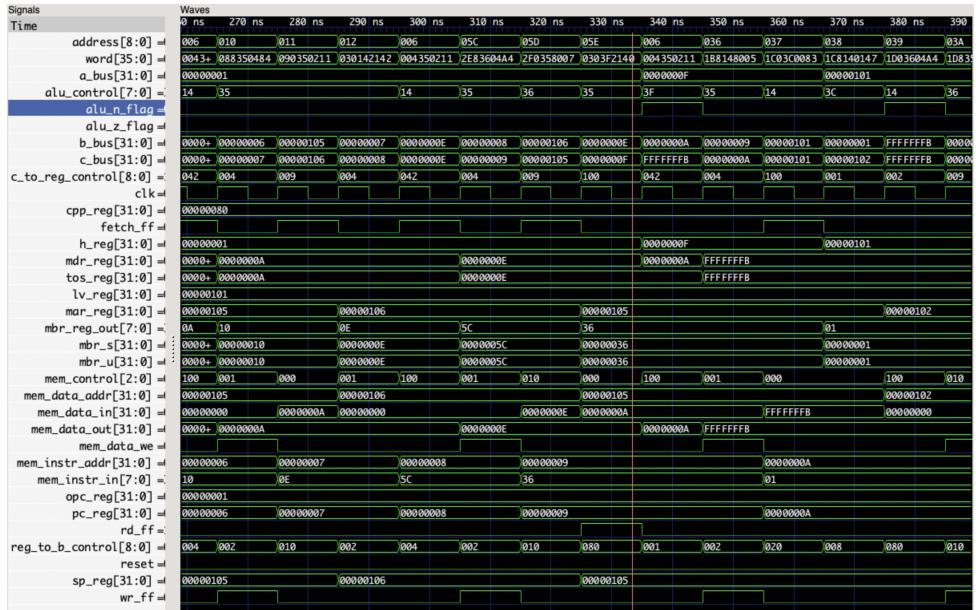


Figura 63: Simulazione ISUB modificata

2. H = TOS + 1
NEXT ADDRESS: 0x5E
JMP: 000
ALU: 00110101
BUS C: 1000000000
MEM: 000
BUS B: 0111
3. MDR = TOS = MDR - H; wr; goto main
NEXT ADDRESS: 0x6
JMP: 000
ALU: 00111111
BUS C: 001000010
MEM: 100
BUS B: 0000

UART

Parte I

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.

Sistema complessivo

Il sistema progettato si compone di due entità A e B, le quali svolgono rispettivamente il ruolo di trasmettitore e ricevitore nel contesto della comunicazione. Considerato lo scarso numero di switch della board di sviluppo il sistema A acquisisce gli 8 bit per la comunicazione seriale tramite un decodificatore 3:8. Analogamente, il sistema B deve essere dotato di un codificatore 8:3 per la corretta visualizzazione del messaggio ricevuto sui led.

Entity e Architecture - sistema

```
entity sistema is
    port ( CLK : in std_logic;
           RST : in std_logic;
           input : in std_logic_vector(0 to 2);
           start : in std_logic;
           output : out std_logic_vector(0 to 2));
end sistema;
```

```

architecture structural of sistema is
    signal tr : std_logic;
    component Nodo_tx
        port ( TXD : out std_logic := '1';
               CLK : in  std_logic;
               IN_DEC : in  std_logic_vector (2 downto 0);
               WR : in  std_logic;
               RST : in std_logic := '0');
    end component;

    component Nodo_rx
        port ( RXD : in std_logic  := '1';
               CLK : in  std_logic;
               OUT_COD : out std_logic_vector (2 downto 0);
               RST : in std_logic := '0');
    end component;

    component ButtonDebouncer is
        generic ( CLK_period : integer := 20;
                  btn_noise_time : integer := 2000000000 );
        Port ( CLK : in STD_LOGIC;
                BTN : in STD_LOGIC;
                CLEARED_BTN : out STD_LOGIC);
    end component;

    signal clear_wave: std_logic;
begin

    btn : ButtonDebouncer
        Port map ( CLK =>CLK,
                   BTN => start,
                   CLEARED_BTN => clear_wave);

    tx : Nodo_tx
        Port map ( TXD => tr,
                   CLK => CLK,
                   IN_DEC => input,

```

```

WR => clear_wave,
RST => RST);

rx : Nodo_rx
Port map ( RXD => tr,
CLK => CLK,
OUT_COD => output,
RST => RST);
end structural;

```

Sintesi

In figura 64 è mostrato un esempio di sintesi su board nel quale il sistema A comunica con il sistema B.

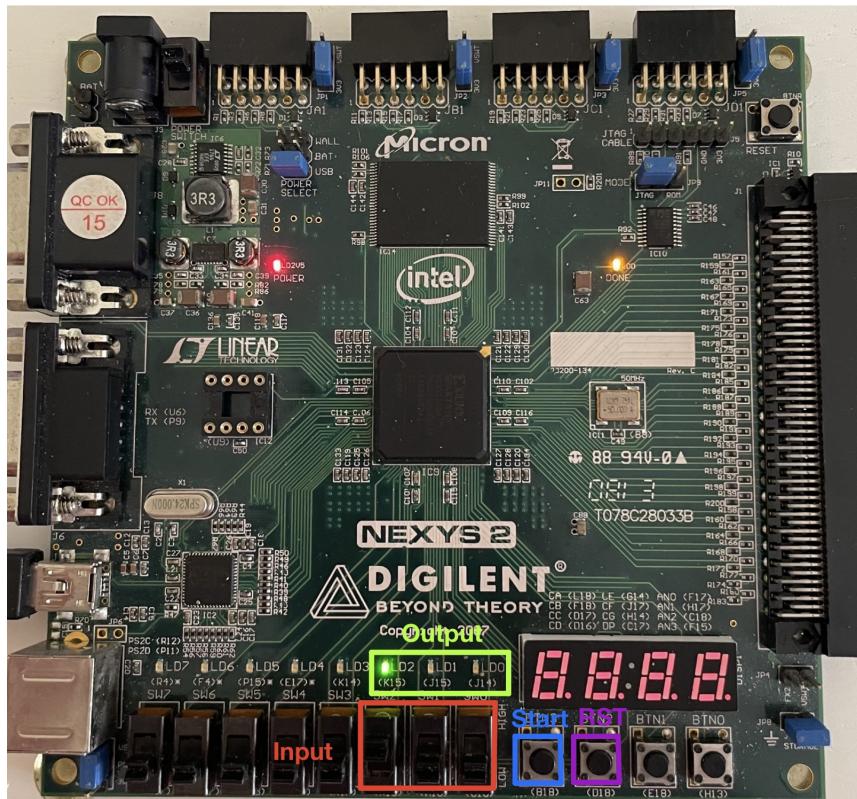


Figura 64: Sintesi su board

Parte II

Come variante dell'esercizio precedente, 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.

Per entrambe un'unità di controllo permette di scandire le fasi della comunicazione.

Sistema A

Il sistema A ha una propria unità operativa costituita da una ROM dove sono memorizzate le N stringhe da trasmettere, un contatore per l'indirizzamento e un dispositivo UART per la comunicazione.

In figura 65 viene mostrata la composizione interna di A.

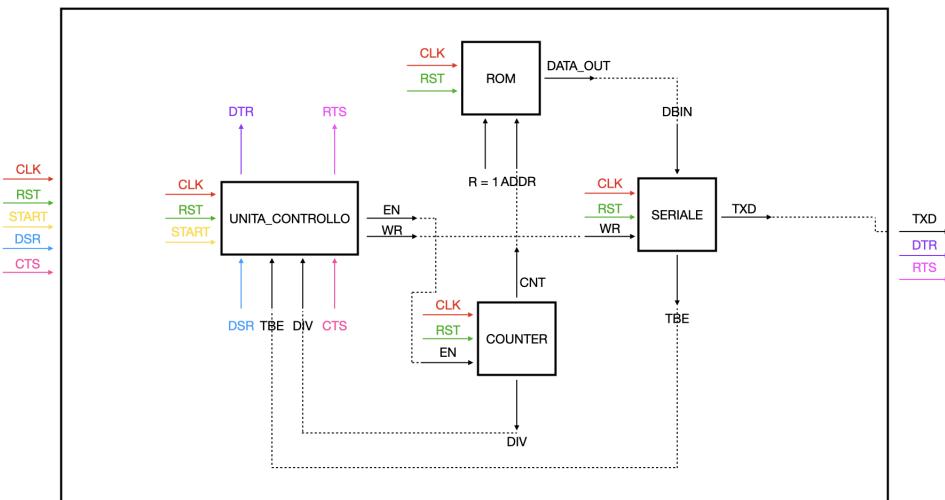


Figura 65: Architettura interna del sistema A

Automa

Il funzionamento di A, così come per B, risiede nell'unità di controllo. Ne presentiamo l'automa in figura 66.

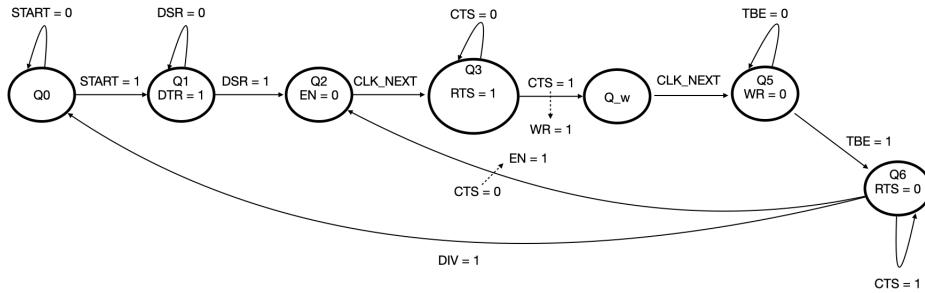


Figura 66: Automa unità di controllo A

- Q0: il sistema è in attesa dello start.
- Q1: inizia l'handshaking, si alza DTR e si attende DSR.
- Q2: attendo che il dato da trasmettere sia stabile.
- Q3: si alza RTS, si attende che CTS si alzi per abilitare la scrittura sull'UART.
- Q_w: stato di attesa della durata di un colpo di clock per la trasmissione.
- Q5: viene abbassato WR e si attende il TBE che segnala il termine della comunicazione.
- Q6: si abbassa RTS, se il segnale DIV è alto, ossia sono stati trasmessi tutti i caratteri, si torna in Q0, altrimenti si attende il segnale CTS dal ricevitore prima di riprendere la trasmissione in Q2.

Entity e architecture - unità di controllo A

```
entity fsm_trasmettitore is
  port ( CLK : in std_logic;
```

```

RST : in std_logic;
START : in std_logic;
DSR : in std_logic;
DTR : out std_logic;
EN : out std_logic;
RTS : out std_logic;
CTS : in std_logic;
WR : out std_logic := '0';
TBE : in std_logic;
DIV : in std_logic);
end fsm_trasmettitore;

architecture Behavioral of fsm_trasmettitore is
type stato is (Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q_w);
signal stato_corrente : stato := Q0;

begin
mem : process(CLK)
begin
if(CLK'event and CLK = '1') then
    if(RST = '1') then
        stato_corrente <= Q0;
    end if;
    case stato_corrente is
        when Q0 =>
            if (START = '1') then
                stato_corrente <= Q1;
            else
                stato_corrente <= Q0;
            end if;
        when Q1 =>
            DTR <= '1';
            if (DSR = '1') then
                stato_corrente <= Q2;
            else
                stato_corrente <= Q1;
            end if;
        when Q2 =>

```

```

        EN <= '0';
        stato_corrente <= Q3;
when Q3 =>
    RTS <= '1';
    if (CTS = '1') then
        WR <= '1';
        stato_corrente <= Q_w;
    else
        stato_corrente <= Q3;
    end if;
when Q_w =>
    stato_corrente <= Q4;
when Q4 =>
    WR <= '0';
    if (TBE = '1') then
        stato_corrente <= Q5;
    else
        stato_corrente <= Q4;
    end if;
when Q5 =>
    RTS <= '0';
    if (DIV = '1') then
        stato_corrente <= Q0;
    elsif (CTS = '0') then
        EN <= '1';
        stato_corrente <= Q2;
    else
        stato_corrente <= Q5;
    end if;
end case;
end if;
end process;
end Behavioral;
```

Sistema B

Dall'altra parte della comunicazione il sistema B, caratterizzato anch'esso da una propria unità operativa, si compone di una memoria RAM per l'imma-gazzinamento delle stringhe ricevute, un contatore per l'indirizzamento della memoria stessa e un dispositivo UART per la comunicazione.

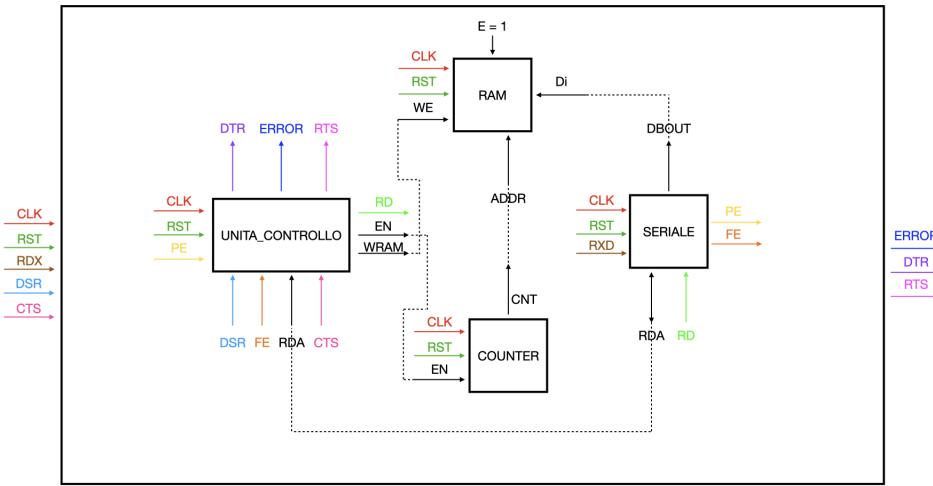


Figura 67: Architettura interna del sistema B

Automa

In figura 68 è presente l'automa che descrive il funzionamento di B.

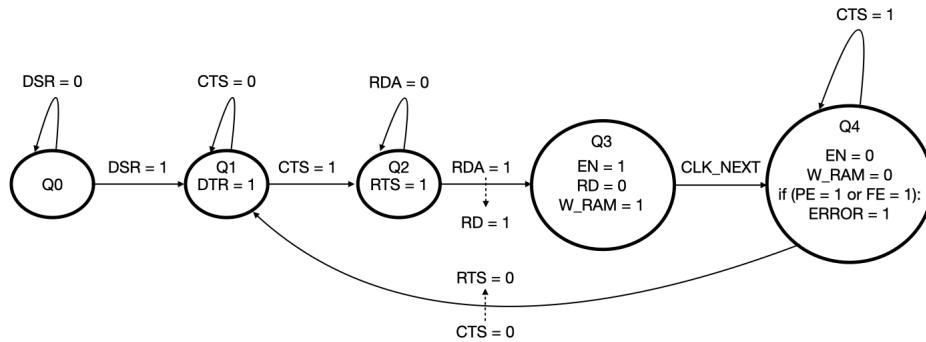


Figura 68: Automa unità di controllo B

- **Q0:** il sistema rimane in attesa di DSR.

- Q1: inizia l'handshaking, si alza DTR e si attende che il CTS segnali che è possibile iniziare a ricevere il primo carattere.
- Q2: viene alzato RTS e si attende che l'UART termini di ricevere tutti i bit. Quando RDA si alza significa che la ricezione è terminata e che il carattere è disponibile in uscita dall'UART. Di conseguenza viene alzato RD.
- Q3: si abbassa RD e si dà sia l'incremento al contatore sia l'abilitazione in scrittura alla RAM.
- Q4: si abbassano EN e W_RAM, eventuali errori vengono comunicati dal segnale in uscita ERROR. Quando CTS si azzera, viene abbassato RTS e la ricezione ricomincia.

Entity e architecture - unità di controllo B

```

entity fsm_ricevitore is
    port ( CLK : in std_logic;
    RST : in std_logic;
    DSR : in std_logic;
    DTR : out std_logic;
    EN : out std_logic;
    RTS : out std_logic;
    CTS : in std_logic;
    W_RAM : out std_logic := '0';
    RDA : in std_logic;
    RD : out std_logic;
    ERROR : out std_logic;
    PE : in std_logic;
    FE : in std_logic);
end fsm_ricevitore;

architecture Behavioral of fsm_ricevitore is
    type stato is (Q0, Q1, Q2, Q3, Q4);
    signal stato_corrente : stato := Q0;

begin
    mem : process(CLK)

```

```

begin
    if(CLK'event and CLK = '1') then
        if(RST = '1') then
            stato_corrente <= Q0;
        end if;
        case stato_corrente is
            when Q0 =>
                stato_corrente <= Q0;
                if (DSR = '1') then
                    stato_corrente <= Q1;
                end if;
            when Q1 =>
                DTR <= '1';
                if (CTS = '1') then
                    stato_corrente <= Q2;
                end if;
            when Q2 =>
                RTS <= '1';
                if (RDA = '1') then
                    RD <= '1';
                    stato_corrente <= Q3;
                else
                    stato_corrente <= Q2;
                end if;
            when Q3 =>
                W_RAM <= '1';
                RD <= '0';
                EN <= '1';
                stato_corrente <= Q4;
            when Q4 =>
                W_RAM <= '0';
                EN <= '0';
                if (PE = '1' OR FE = '1') then
                    error <= '1';
                end if;
                if (CTS = '0') then
                    RTS <= '0';
                    stato_corrente <= Q1;

```

```

        end if;
    end case;
end if;
end process;
end Behavioral;

```

Sistema Complessivo

Complessivamente il sistema si compone come in figura 69.

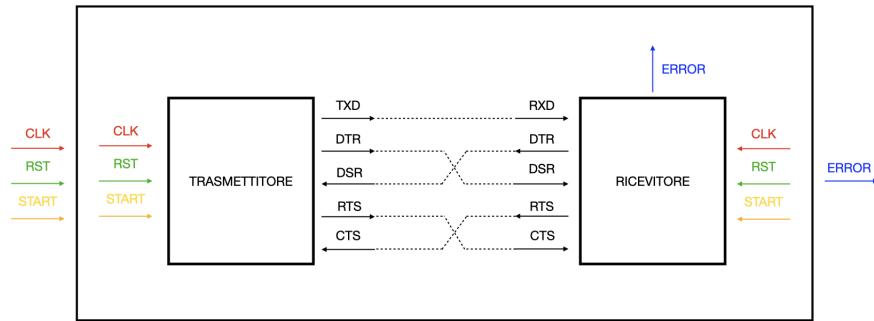


Figura 69: Architettura interna del sistema complessivo

Entity e architecure - Sistema complessivo

```

entity system is
    port ( START : IN  std_logic;
           CLK : IN  std_logic;
           RST : IN  std_logic;
           error : out std_logic;
           dato_ram : out std_logic_vector(7 downto 0));
end system;

architecture Structural of system is
COMPONENT Trasmettitore
    PORT ( START : IN  std_logic;
           CLK : IN  std_logic;
           RST : IN  std_logic;
           TXD : OUT  std_logic;

```

```

        DTR : OUT  std_logic;
        DSR : IN   std_logic;
        RTS : OUT  std_logic;
        CTS : IN   std_logic);
END COMPONENT;

component Ricevitore
    port ( CLK : in std_logic;
           RST : in std_logic;
           RXD : in std_logic;
           DTR : OUT  std_logic;
           DSR : IN   std_logic;
           RTS : OUT  std_logic;
           CTS : IN   std_logic;
           ERROR: OUT std_logic);
end component;

signal d : std_logic;
signal dtr1 : std_logic;
signal rts1: std_logic;
signal cts1 : std_logic;
signal dsr1 : std_logic;

begin
    t0 : Trasmettitore
        PORT map ( START => start,
                   CLK => CLK,
                   RST => RST,
                   TXD => d,
                   DTR => dtr1,
                   DSR => dsr1,
                   RTS =>rts1,
                   CTS =>cts1);

    r0 : Ricevitore
        port map ( CLK => CLK,
                   RST => RST,
                   RXD => d,

```

```

DTR => dsr1,
DSR => dtr1,
RTS => cts1,
CTS => rts1,
ERROR => ERROR);
end Structural;

```

Sintesi - Sistema complessivo

In figura 70 è mostrata la sintesi su board di sviluppo tramite la quale sui led è visualizzato il contenuto dell'ultima locazione della RAM del nodo ricevitore.

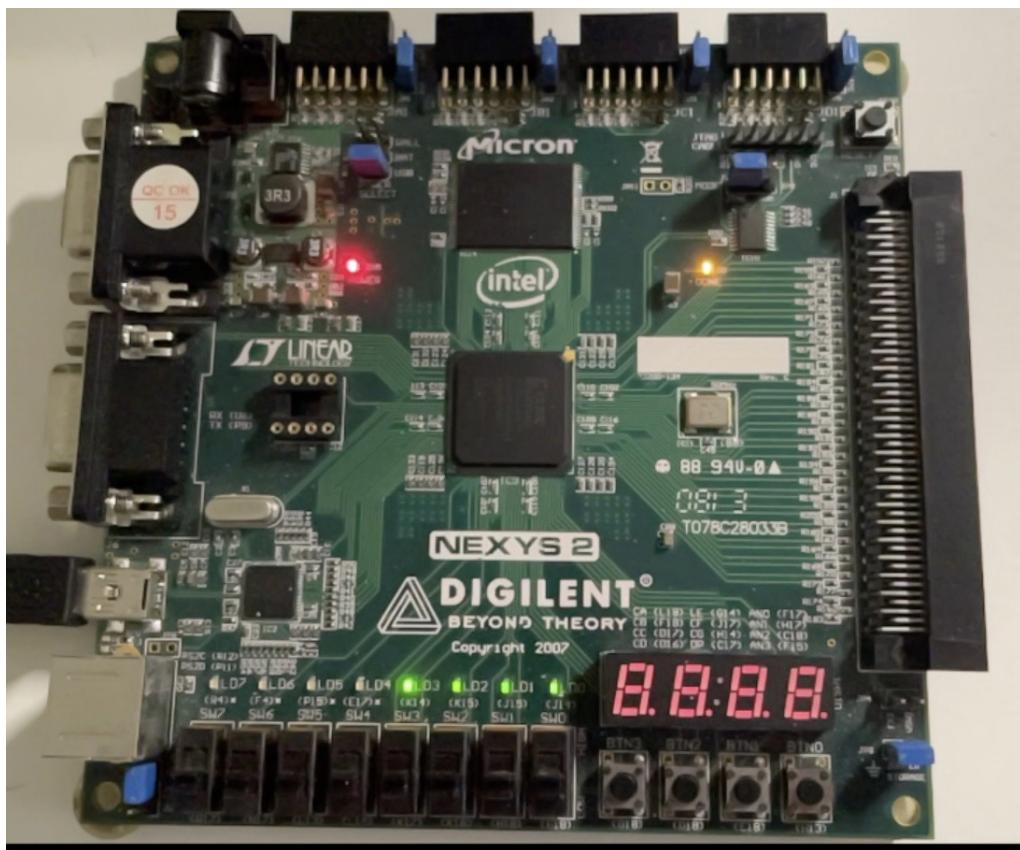


Figura 70: Sintesi su board di sviluppo

Switch multistadio

Parte I

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 un rete con 4 nodi, implementando uno schema a priorità fissa fra i nodi (ed. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).*
- (b) *(Opzionale) rimuovendo l'ipotesi di lavorare secondo uno schema a priorità fra i nodi e considerando una rete di 8 nodi, lo switch deve gestire eventuali conflitti generati da collisioni secondo un meccanismo a scelta (ad es. perdendo uno dei messaggi in conflitto).*
- (c) *(Opzionale) Si implementi un protocollo di handshaking semplice regolato da una coppia di segnali (pronto a inviare/pronto a ricevere) per l'invio di ciascun messaggio fra due nodi.*

Per la realizzazione dello switch multistadio secondo il modello Omega Network separiamo la parte operativa e da quella di controllo. In particolare, il nostro sistema dovrà consentire lo scambio di messaggi in una rete formata da 4 nodi.

Unità operativa

La parte operativa è ottenuta con un approccio strutturale, utilizzando un componente elementare di instradamento. Quest'ultimo, in figura 71, è ottenuto come composizione di un demux 1:2 e di un mux 2:1.

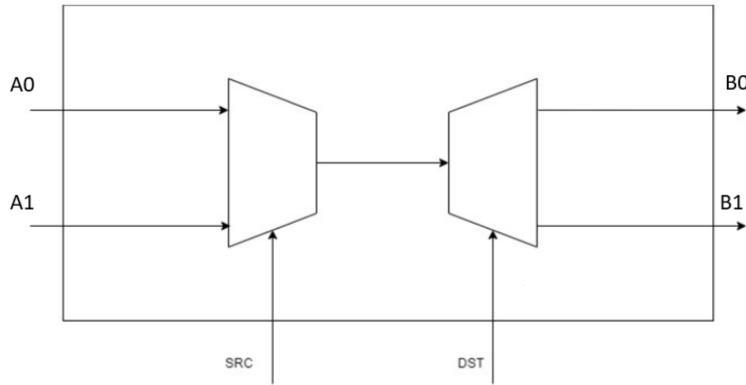


Figura 71: Routing node

Routing node

Entity e architecture - Routing Node

Il componente di instradamento, caratterizzato da 2 ingressi e 2 uscite, è realizzato secondo un approccio dataflow. L'ingresso viene selezionato con un bit proveniente dall'indirizzo sorgente. L'uscita su cui verrà pilotato l'ingresso dipende invece da un bit dell'indirizzo destinazione.

```

entity RoutingNode is
    generic ( N : integer := 2);

    port ( DATA_IN1 : in std_logic_vector(N-1 downto 0);
           DATA_IN2 : in std_logic_vector(N-1 downto 0);
           DATA_OUT1 : out std_logic_vector(N-1 downto 0);
           DATA_OUT2 : out std_logic_vector(N-1 downto 0);
           SRC : in std_logic;
           DEST : in std_logic);
end RoutingNode;

architecture Dataflow of RoutingNode is
begin
    DATA_OUT1 <= DATA_IN1 when SRC = '0' and DEST = '0' else
    DATA_IN2 when SRC = '1' and DEST = '0' else "00";
    DATA_OUT2 <= DATA_IN1 when SRC = '0' and DEST = '1' else

```

```

    DATA_IN2 when SRC = '1' and DEST = '1' else "00";
end Dataflow;

```

Entity e architecture - Unità operativa

Il collegamento tra i nodi di instradamento segue il modello di perfect shuffling. I bit dell'indirizzo sorgente e destinazione pilotano a ogni stadio quale l'ingresso che andrà in uscita verso lo stadio successivo. È possibile visualizzare lo schema dei collegamenti complessivo dell'unità operativa in figura 72.

```

entity op_unit is
generic ( N : integer := 2);

port ( IN0 : in std_logic_vector(N-1 downto 0);
IN1 : in std_logic_vector(N-1 downto 0);
IN2 : in std_logic_vector(N-1 downto 0);
IN3 : in std_logic_vector(N-1 downto 0);
SRC : in std_logic_vector(1 downto 0);
DEST : in std_logic_vector(1 downto 0);
OUT0 : out std_logic_vector(N-1 downto 0);
OUT1 : out std_logic_vector(N-1 downto 0);
OUT2 : out std_logic_vector(N-1 downto 0);
OUT3 : out std_logic_vector(N-1 downto 0));
end op_unit;

architecture Structural of op_unit is
component RoutingNode is
generic ( N : integer := 2);

port ( DATA_IN1 : in std_logic_vector(N-1 downto 0);
DATA_IN2 : in std_logic_vector(N-1 downto 0);
DATA_OUT1 : out std_logic_vector(N-1 downto 0);
DATA_OUT2 : out std_logic_vector(N-1 downto 0);
SRC : in std_logic;
DEST : in std_logic);
end component;

```

```

type u_type is array(0 to 3) of std_logic_vector(N-1 downto 0);
signal u : u_type;
begin

    nodo01 : RoutingNode
        generic map ( N => N)

            port map ( DATA_IN1 => IN0,
DATA_IN2 => IN1,
DATA_OUT1 => u(0),
DATA_OUT2 => u(1),
SRC => SRC(0),
DEST => DEST(1) );

    nodo23 : RoutingNode
        generic map ( N => N)

            port map ( DATA_IN1 => IN2,
DATA_IN2 => IN3,
DATA_OUT1 => u(2),
DATA_OUT2 => u(3),
SRC => SRC(0),
DEST => DEST(1));

    nodo02 : RoutingNode
        generic map ( N => N)

            port map ( DATA_IN1 => u(0),
DATA_IN2 => u(2),
DATA_OUT1 => OUT0,
DATA_OUT2 => OUT1,
SRC => SRC(1),
DEST => DEST(0));

    nodo13 : RoutingNode
        generic map ( N => N)

            port map ( DATA_IN1 => u(1),

```

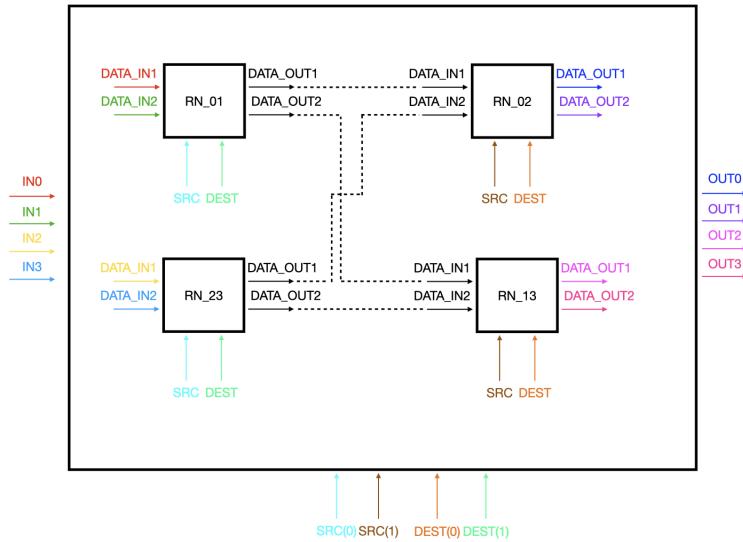


Figura 72: Unità operativa

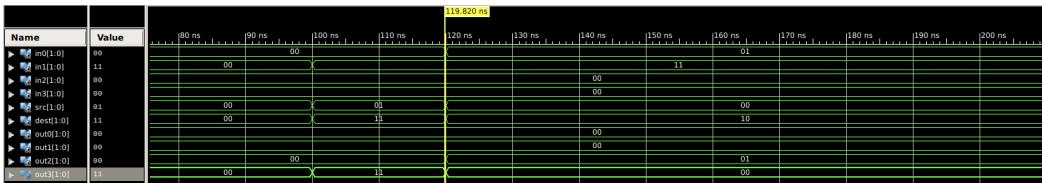


Figura 73: Simulazione Unità operativa

```

DATA_IN2 => u(3),
DATA_OUT1 => OUT2,
DATA_OUT2 => OUT3,
SRC => SRC(1),
DEST => DEST(0));
end Structural;

```

Simulazione - Unità operativa

Si riporta la simulazione dell'unità operativa in figura 73.

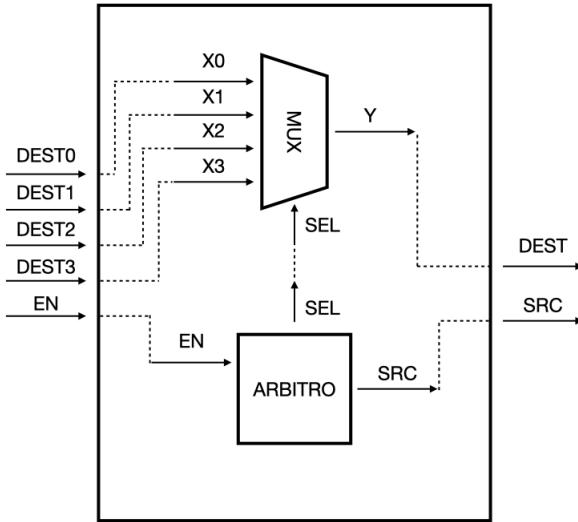


Figura 74: Unità di controllo

Unità di controllo

Ognuno dei quattro nodi fornisce all’unità di controllo una richiesta per poter trasmettere insieme all’indirizzo destinazione. Le richieste sono filtrate da un arbitro secondo uno schema a priorità fissa. Sulla base delle richieste ricevute, quest’ultimo selezionerà il nodo abilitato alla trasmissione. L’indirizzo sorgente e destinazione perverranno così all’unità operativa. L’architettura dell’unità di controllo è mostrata in figura 74.

Entity e architecture - Unità di controllo

Si riporta la descrizione VHDL dell’unità di controllo.

```
entity control_unit is
    port ( DEST0 : in std_logic_vector(1 downto 0);
    DEST1 : in std_logic_vector(1 downto 0);
    DEST2 : in std_logic_vector(1 downto 0);
    DEST3 : in std_logic_vector(1 downto 0);
    EN : in std_logic_vector(0 to 3);
    SRC : out std_logic_vector(1 downto 0);
    DEST : out std_logic_vector(1 downto 0));
```

```

end control_unit;

architecture Structural of control_unit is
component Mux_4_1
    port ( X0 : in std_logic_vector(1 downto 0);
           X1 : in std_logic_vector(1 downto 0);
           X2 : in std_logic_vector(1 downto 0);
           X3 : in std_logic_vector(1 downto 0);
           sel : in std_logic_vector(1 downto 0);
           Y : out std_logic_vector(1 downto 0));
end component;

component Arbitro
    port ( EN : in std_logic_vector(0 to 3);
           sel : out std_logic_vector(1 downto 0);
           SRC : out std_logic_vector(1 downto 0));
end component;

signal u : std_logic_vector(1 downto 0);

begin
    m : Mux_4_1
        port map ( X0 => DEST0,
                   X1 => DEST1,
                   X2 => DEST2,
                   X3 => DEST3,
                   sel => u,
                   Y => DEST);
    arb : Arbitro
        port map ( EN => EN,
                   SRC => SRC,
                   sel => u);
end Structural;

```

Mux

Il multiplexer stabilisce, a fronte del segnale di selezione dell'arbitro, quale sarà l'indirizzo destinazione a propagarsi verso l'unità operativa.

Entity e architecture - Mux

```
entity Mux_4_1 is
    port ( X0 : in std_logic_vector(1 downto 0);
           X1 : in std_logic_vector(1 downto 0);
           X2 : in std_logic_vector(1 downto 0);
           X3 : in std_logic_vector(1 downto 0);
           sel : in std_logic_vector(1 downto 0);
           Y : out std_logic_vector(1 downto 0));
end Mux_4_1;

architecture Dataflow of Mux_4_1 is
begin
    Y <=      X0 when sel = "00" else
              X1 when sel = "01" else
              X2 when sel = "10" else
              X3 when sel = "11";
end Dataflow;
```

Arbitro

Entity e architecture - Arbitro

```
entity Arbitro is
    port ( EN : in std_logic_vector(0 to 3);
           sel : out std_logic_vector(1 downto 0);
           SRC : out std_logic_vector(1 downto 0));
end Arbitro;

architecture Dataflow of Arbitro is
begin
    sel <=  "00" when EN(0) = '1' else
            "01" when EN(1) = '1' else
            "10" when EN(2) = '1' else
```

```
"11" when EN(3) = '1';

SRC <= "00" when EN(0) = '1' else
"01" when EN(1) = '1' else
"10" when EN(2) = '1' else
"11" when EN(3) = '1';
end Dataflow;
```

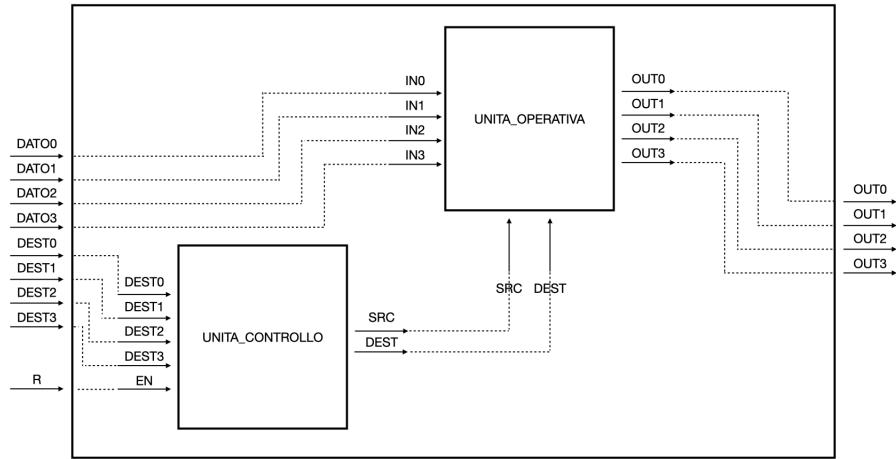


Figura 75: Schema complessivo

Omega Network

Componendo unità di controllo e unità operativa, otteniamo l'architettura complessiva della rete Omega Network, come mostrato in figura 75.

Entity e architecture - Arbitro

```
entity Omega_network is
    generic ( N : integer := 2);

    port ( R : in std_logic_vector(0 to 3);
DEST0 : in std_logic_vector(1 downto 0);
DAT00 : in std_logic_vector(N-1 downto 0);
DEST1 : in std_logic_vector(1 downto 0);
DAT01 : in std_logic_vector(N-1 downto 0);
DEST2 : in std_logic_vector(1 downto 0);
DAT02 : in std_logic_vector(N-1 downto 0);
DEST3 : in std_logic_vector(1 downto 0);
DAT03 : in std_logic_vector(N-1 downto 0);
OUT0 : out std_logic_vector(N-1 downto 0);
```

```

        OUT1 : out std_logic_vector(N-1 downto 0);
        OUT2 : out std_logic_vector(N-1 downto 0);
        OUT3 : out std_logic_vector(N-1 downto 0));
    end Omega_network;

architecture Structural of Omega_network is
    component op_unit
        generic ( N : integer := 2);

        port ( IN0 : in std_logic_vector(N-1 downto 0);
               IN1 : in std_logic_vector(N-1 downto 0);
               IN2 : in std_logic_vector(N-1 downto 0);
               IN3 : in std_logic_vector(N-1 downto 0);
               SRC : in std_logic_vector(1 downto 0);
               DEST : in std_logic_vector(1 downto 0);
               OUT0 : out std_logic_vector(N-1 downto 0);
               OUT1 : out std_logic_vector(N-1 downto 0);
               OUT2 : out std_logic_vector(N-1 downto 0);
               OUT3 : out std_logic_vector(N-1 downto 0));
    end component;

    component control_unit
        port ( DEST0 : in std_logic_vector(1 downto 0);
               DEST1 : in std_logic_vector(1 downto 0);
               DEST2 : in std_logic_vector(1 downto 0);
               DEST3 : in std_logic_vector(1 downto 0);
               EN : in std_logic_vector(0 to 3);
               SRC : out std_logic_vector(1 downto 0);
               DEST : out std_logic_vector(1 downto 0));
    end component;

    signal sorg : std_logic_vector(1 downto 0);
    signal destin : std_logic_vector(1 downto 0);

begin
    op : op_unit
        generic map ( N => N)

```

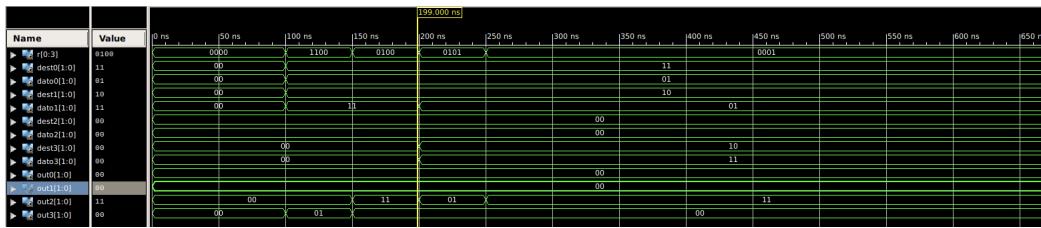


Figura 76: Simulazione Omega Network

```

port map ( IN0 => DAT00,
IN1 => DAT01,
IN2 => DAT02,
IN3 => DAT03,
SRC => sorg,
DEST => destin,
OUT0 => OUT0,
OUT1 => OUT1,
OUT2 => OUT2,
OUT3 => OUT3);

cnt : control_unit
port map ( DEST0 => DEST0,
DEST1 => DEST1,
DEST2 => DEST2,
DEST3 => DEST3,
EN => R,
SRC => sorg,
DEST => destin);
end Structural;

```

Simulazione - Omega Network

In figura 76 si riporta la simulazione della rete Omega Network.

Divisore non restoring

Parte I

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.

Divisore

Entity e architecture - Divisore

Per la realizzazione del sistema complessivo abbiamo individuato una parte operativa e una di controllo. In particolare, in figura 77 è mostrata l'architettura complessiva del divisore non restoring.

La parte operativa comprende:

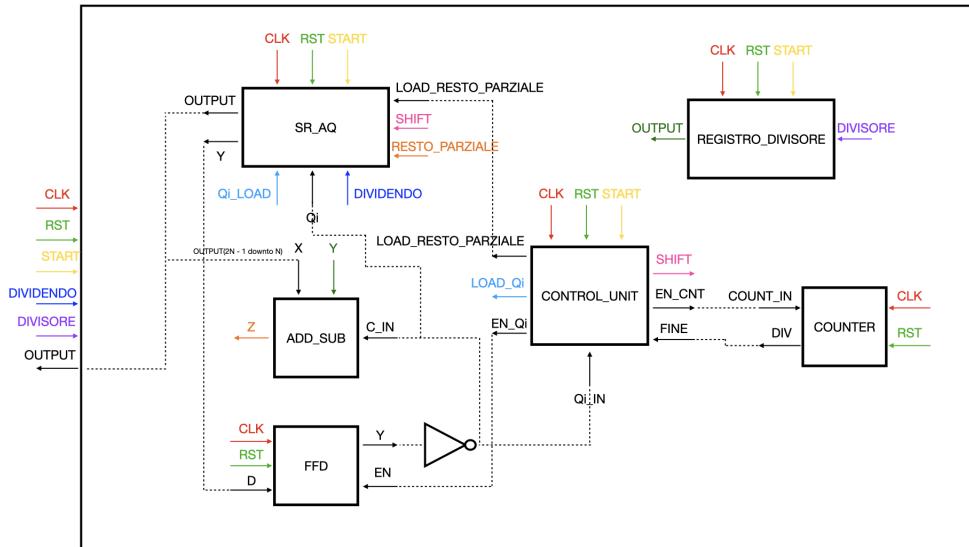


Figura 77: Architettura divisore

- Shift register AQ: registro contenente il dividendo e che al termine conterrà quoziante e resto
- Registro divisore
- Sommatore/sottrattore RCA
- Flip flop D: memorizza il segno della operazione precedente per sapere se sommare o sottrarre all'iterazione successiva
- Contatore: necessario per la terminazione delle operazioni

Shift register AQ

Entity e architecture - Shift register AQ

Per la descrizione del registro utilizziamo un approccio comportamentale, dovendo gestire differenti segnali di controllo.

```
entity shift_register_AQ is
  generic ( N :integer :=4);
```

```

port ( CLK : in std_logic;
RST : in std_logic;
dividendo : in std_logic_vector(N-1 downto 0);
resto_parziale: in std_logic_vector(N-1 downto 0);
output: out std_logic_vector(2*N-1 downto 0);
start : in std_logic;
load_resto_parziale : in std_logic;
qi : in std_logic;
qi_load : in std_logic;
shift : in std_logic;
Y : OUT std_logic := '0');
end shift_register_AQ;

architecture behavioural of shift_register_AQ is
signal temp: std_logic_vector(2*N-1 downto 0) := (others => '0');
begin
y <= temp(2*N - 1);
SR: process(CLK, RST)
begin
if(RST = '1') then
temp<=(others=>'0');
elsif(rising_edge(CLK)) then
if(start = '1') then
temp(N-1 downto 0) <= dividendo;
elsif (shift = '1') then
temp <= temp(2*N-2 downto 0) & '0';
elsif(load_resto_parziale = '1') then
temp(2*N-1 downto N) <= resto_parziale;
elsif (qi_load = '1') then
temp(0) <= qi;
end if;
end if;
end process;
output <= temp;
end behavioural;

```

Registro divisore

Entity e architecture - Registro divisore

Il registro divisore viene caricato in fase di inizializzazione con il divisore, alzando un opportuno segnale di start.

```
entity registroDivisore is
    generic(N : integer := 4);

    port ( divisore : in std_logic_vector(N-1 downto 0);
           CLK, RST, start: in std_logic;
           output: out std_logic_vector(N-1 downto 0) := (others => '0'));
end registroDivisore;

architecture behavioural of registroDivisore is
begin
    proc: process(clk, rst)
    begin
        if(RST = '1') then
            output <= (others=>'0');
        elsif(rising_edge(CLK)) then
            if (start = '1') then
                output <= divisore;
            end if;
        end if;
    end process;
end behavioural;
```

Flip flop D

Entity e architecture - FFD

È qui riportata la descrizione comportamentale del Flip flop D asincrono. Esso viene utilizzato per memorizzare il segno dell'operazione effettuata dal RCA. Tale informazione è necessaria per poter decidere quale sarà la prossima operazione da eseguire.

```
entity FFD is
    port ( CLK, RST, d, EN: in std_logic;
```

```

        y: out std_logic :='0');
end FFD;

architecture behavioural of FFD is
begin
    FF_D: process(clk, rst)
begin
    if(RST = '1') then
        y <= '0';
    elsif(falling_edge(CLK)) then
        if (EN = '1') then
            y <= d;
        end if;
    end if;
    end process;
end behavioural;

```

ADD/SUB

Entity e architecture - ADD/SUB

```

entity adder_sub is
generic ( N:integer := 4);

port ( X, Y: in std_logic_vector(N-1 downto 0);
cin: in std_logic;
Z: out std_logic_vector(N-1 downto 0);
cout: out std_logic);
end adder_sub;

architecture structural of adder_sub is
component ripple_carry is
port ( X, Y: in std_logic_vector(N-1 downto 0);
c_in: in std_logic;
c_out: out std_logic;
Z: out std_logic_vector(N-1 downto 0));
end component;

```

```

signal complementoy: std_logic_vector(N-1 downto 0);
begin
    complemento_y: FOR i IN 0 TO N-1 GENERATE
        complementoy(i) <= Y(i) xor cin;
    END GENERATE;

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

```

RCA

Entity e architecture - RCA

```

entity ripple_carry is
    generic ( N : integer := 4);

    port ( X, Y: in std_logic_vector(N-1 downto 0);
           c_in: in std_logic;
           c_out: out std_logic;
           Z: out std_logic_vector(N-1 downto 0));
end ripple_carry;

architecture structural of ripple_carry is
    component full_adder is
        port ( a,b: in std_logic;
               cin: in std_logic;
               cout, s: out std_logic);
    end component;

    signal temp: std_logic_vector(N-1 downto 0);
begin
    RA_first: full_adder
        port map ( X(0),

```

```

Y(0),
c_in,
temp(0),
Z(0));

RA_middle: FOR i IN 1 TO N-2 GENERATE
    RA: full_adder
        port map(X(i),
                  Y(i),
                  temp(i-1),
                  temp(i),
                  Z(i));
    END GENERATE;

RA_last: full_adder
    port map ( X(N-1),
               Y(N-1),
               temp(N-2),
               c_out,
               Z(N-1));
end structural;

```

FA

Entity e architecture - FA

```

entity full_adder is
    port ( a,b: in std_logic;
           cin: in std_logic;
           cout, s: out std_logic);
end full_adder;

architecture rtl of full_adder is
begin
    s <= a xor b xor cin;
    cout<= (a and b) or (cin and (a xor b));
end rtl;

```

Unità di controllo

Entity e architecture - Unità di controllo

Viene qui riportata la descrizione VHDL dell'unità di controllo e del relativo automa in figura 78.

- Idle: si attende che il segnale di reset si alzi
- Wait_start: si attende il segnale di start per avviare il caricamento degli operandi
- Q_load: si effettua il caricamento degli operandi nei registri
- Q1: si alza il segnale shift per effettuare lo shift del contenuto del registro AQ
- Q2: abbasso il segnale di shift e attendo che l'uscita dell'adder/subtracter si stabilizzi
- Q3: si alza il segnale load_resto_parziale per salvare l'output dell'adder/subtracter nel registro AQ
- Q_wait: si abbassa il segnale load_resto_parziale, memorizzo il segno dell'operazione nel Flip flop D e si incrementa il contatore
- Q4: si alza il segnale Load_qi per caricare il LSB del registro AQ con il valore memorizzato nel Flip flop D
- Q5: abbasso il segnale Load_qi. Se le operazioni sono terminate vado nello stato di correzione altrimenti torno in Q1 per il prossimo passo
- Q_corr: nel caso in cui l'ultimo risultato fosse positivo si effettua un passo di correzione prima di tornare nello stato di idle

```
entity unita_controllo is
  generic ( N : integer:= 4);

  port ( CLK, RST : IN STD_LOGIC;
         start : in std_logic;
         load_resto_parziale : out std_logic;
```

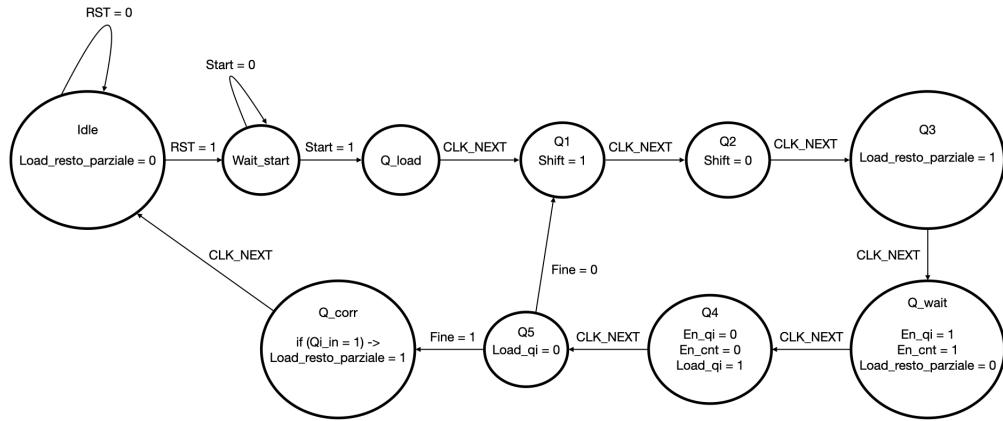


Figura 78: Automa unità di controllo

```

qi_in : in std_logic := '0';
en_qi: out std_logic;
shift : out std_logic;
en_cnt : out std_logic;
fine : in std_logic;
load_qi : out std_logic);
end unita_controllo;

architecture structural of unita_controllo is
type state is (idle, wait_start, q_load, q1 , q2, q3, q4,q_wait,q5,q_corr);
signal current_state : state := idle;

begin
    reg_stato: process(CLK)
    begin
        if(rising_edge(CLK)) then
            CASE current_state is
                WHEN idle =>
                    load_resto_parziale <= '0';
                    if(RST = '1') then
                        current_state <= wait_start;
                    else
                        current_state <= idle;
                WHEN wait_start =>
                    if(Start = '1') then
                        current_state <= q_load;
                    else
                        current_state <= wait_start;
                WHEN q_load =>
                    if(Fine = '0') then
                        current_state <= q1;
                    else
                        current_state <= q_load;
                WHEN q1 =>
                    if(Fine = '0') then
                        current_state <= q2;
                    else
                        current_state <= q1;
                WHEN q2 =>
                    if(Fine = '0') then
                        current_state <= q3;
                    else
                        current_state <= q2;
                WHEN q3 =>
                    if(Fine = '0') then
                        current_state <= q_wait;
                    else
                        current_state <= q3;
                WHEN q4 =>
                    if(Fine = '0') then
                        current_state <= q5;
                    else
                        current_state <= q4;
                WHEN q5 =>
                    if(Fine = '1') then
                        current_state <= q_corr;
                    else
                        current_state <= q5;
                WHEN q_corr =>
                    if(Fine = '1') then
                        current_state <= q5;
                    else
                        current_state <= q_corr;
                WHEN q_wait =>
                    if(Fine = '1') then
                        current_state <= q5;
                    else
                        current_state <= q_wait;
            end CASE;
        end if;
    end process;
end;

```

```

        end if;
WHEN wait_start =>
    if(start = '1') then
        current_state <= q_load;
    else
        current_state<=wait_start;
    end if;
WHEN q_load =>
    current_state <= q1;
WHEN q1 =>
    shift <= '1';
    current_state <= q2;
WHEN q2 =>
    shift <= '0';
    current_state <= q3;
WHEN q3 =>
    current_state <= q_wait;
    load_resto_parziale <= '1';
WHEN q_wait =>
    en_qi <= '1';
    load_resto_parziale <= '0';
    current_state <= q4;
    en_cnt <= '1';
WHEN q4 =>
    en_qi <= '0';
    load_qi <='1';
    en_cnt <= '0';
    current_state <= q5;
WHEN q5 =>
    load_qi <='0';
    if (fine = '1') then
        current_state <= q_corr;
    else
        current_state <= q1;
    end if;
WHEN q_corr =>
    if (qi_in = '0') then
        load_resto_parziale <= '1';

```

```

        end if;
        current_state <= idle;
    end CASE;
end if;
end process;
end structural;

```

Simulazione - Divisore

In figura 79 è mostrata la simulazione del divisore.

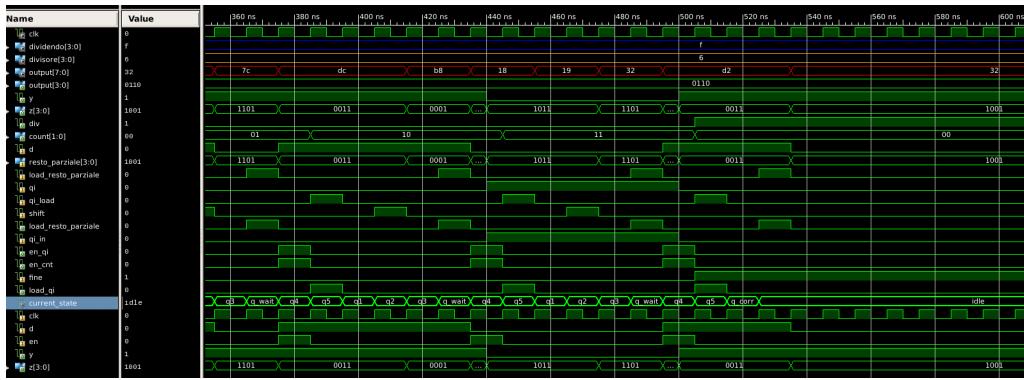


Figura 79: Simulazione divisore

Implementazione su FPGA

Per l'implementazione su FPGA, analogamente agli esercizi precedenti, abbiamo utilizzato il gestore del display e due debouncer per il segnale di reset e start. In figura 80 è riportato un esempio di divisione sulla board.

```

entity divisore_board is
generic ( N : integer := 4);

port (
    dividendo : in std_logic_vector(N-1 downto 0);
    divisor : in std_logic_vector(N-1 downto 0);
    start : in std_logic;
    RST : in std_logic;
    CLK : in std_logic;
    ...
);

```

```

anodes : out std_logic_vector(3 downto 0);
cathodes : out std_logic_vector(7 downto 0));
end divisore_board;

architecture structural of divisore_board is
    component Divisore_NR is
        port ( divisore : in std_logic_vector(N-1 downto 0);
               dividendo : in std_logic_vector(N-1 downto 0);
               start : in std_logic;
               RST : in std_logic;
               CLK : in std_logic;
               OUTPUT : out std_logic_vector(2*N - 1 downto 0));
    end component;

    component Seven_segments_display is
        port ( clk : in std_logic;
               en : in std_logic;
               rst : in std_logic;
               enable_digit : in std_logic_vector(3 downto 0);
               value : in std_logic_vector(15 downto 0);
               dots : in std_logic_vector(3 downto 0);
               anodes : out std_logic_vector(3 downto 0);
               cathodes : out std_logic_vector(7 downto 0));
    end component;

    component ButtonDebouncer is
        Port ( CLK : in STD_LOGIC;
               BTN : in STD_LOGIC;
               CLEARED_BTN : out STD_LOGIC);
    end component;

    signal clear_start : std_logic;
    signal clear_rst : std_logic;
    signal output_AQ :std_logic_vector(7 downto 0);
    signal output_board : std_logic_vector(15 downto 0) := (others =>'0');

begin
    output_board(7 downto 0)  <= output_AQ;

```

```

dv_0 : Divisore_NR
  port map ( divisore => divisore,
  dividendo => dividendo,
  start => clear_start,
  RST => clear_rst,
  CLK => CLK,
  OUTPUT => output_AQ);

btn_RST : ButtonDebouncer
  Port map( CLK => clk,
  BTN => rst,
  CLEARED_BTN => clear_rst);

btn_start : ButtonDebouncer
  Port map( CLK => clk,
  BTN => start,
  CLEARED_BTN => clear_start);

ssd : Seven_segments_display
  port map ( clk => clk,
  en => '1',
  rst => clear_rst,
  enable_digit => "1111",
  value => output_board ,
  dots  => "0000",
  anodes => anodes,
  cathodes => cathodes);
end structural;

```

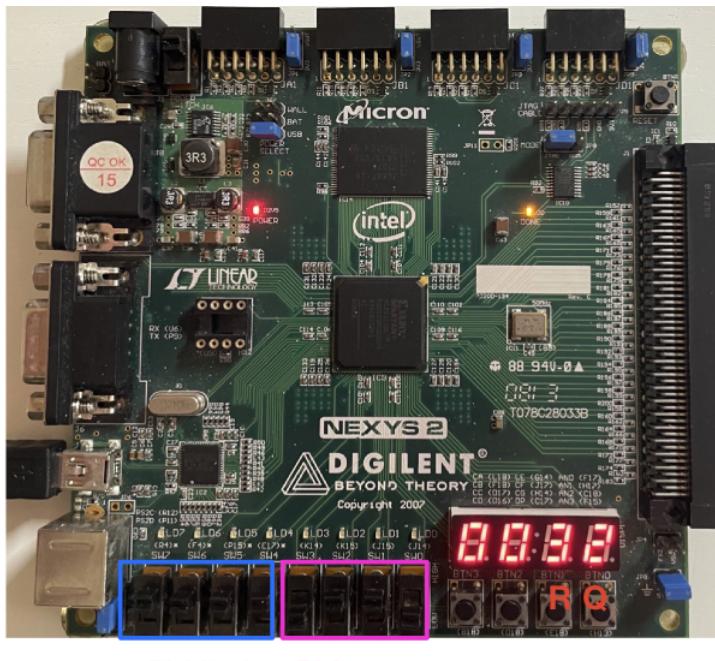


Figura 80: Divisore su FPGA

Blowfish algorithm

Parte I

Il nostro progetto affronta il problema della cifratura su sistemi embedded. In particolare, abbiamo implementato l'algoritmo di cifratura e decifratura Blowfish. Si tratta di un algoritmo a chiave simmetrica a blocchi di lunghezza fissa pari a 64 bit. Oggi sta suscitando nuovamente interesse se utilizzato con blocchi di dimensione maggiore. La lunghezza della chiave varia da 32 a 448 bit. Nella nostra implementazione essa è stata fissata a 32 bit.

Sono previste le seguenti fasi:

- Inizializzazione: si parte da un vettore P composto da 18 locazioni di memoria di 32 bit ciascuna contenente le cifre decimali del π . Ciascun elemento del vettore viene posto in xor con la chiave.
- 16 round di cifratura/decifratura, dove in figura 81 è mostrato l' i -esimo. Il blocco F contenuto all'interno realizza le operazioni mostrate in figura 82.

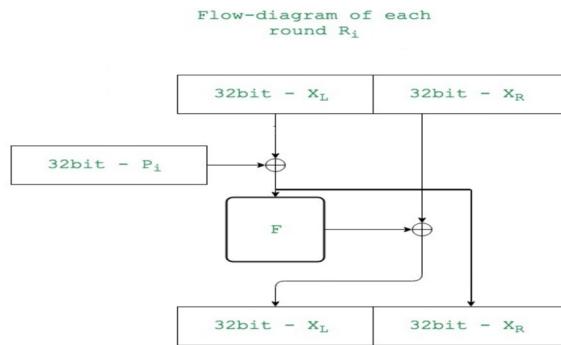


Figura 81: Round i

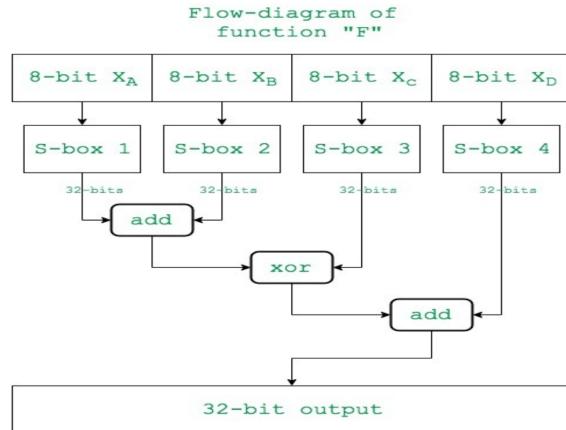


Figura 82: Funzionamento blocco F

- Post processing: dopo i 16 round l'output subisce le trasformazioni mostrate in figura 83.

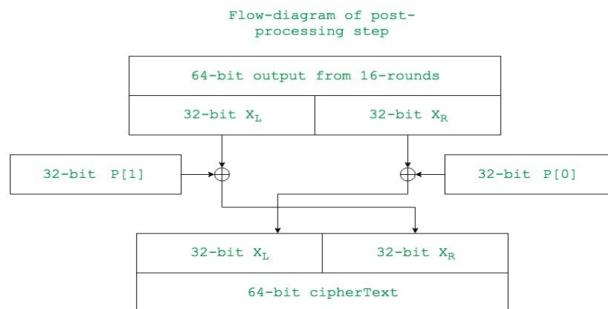


Figura 83: Post processing

Gli schemi complessivi di cifratura e decifratura, che differiscono per l'ordine di scansione del vettore P, sono illustrati rispettivamente in figura 84 e 85.

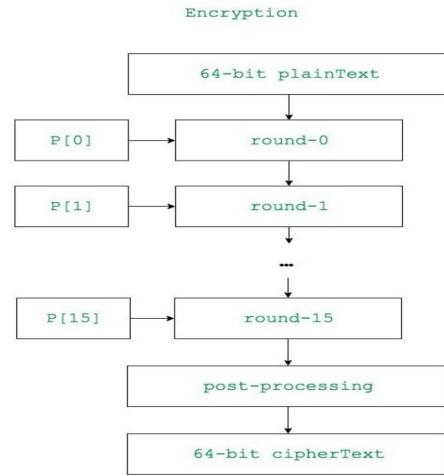


Figura 84: Schema di cifratura

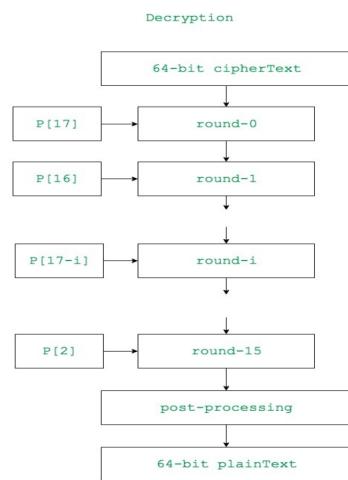


Figura 85: Schema di decifratura

Implementazione python

```
P = np.array([0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344,
0xa4093822, 0x299f31d0, 0x082efa98, 0xec4e6c89,
0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,
```

```

0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917,
0x9216d5d9, 0x8979fb1b ], np.uint32)

plainText = np.uint64(0x123456abcd132536)
key = np.uint32(0xaabb0918)
def f(x):
    bytes = np.zeros(4,np.uint8)
    i = 3
    while x > 0:
        byte = x % 0x100
        bytes[i] = byte
        i-=1
        x // = 0x100

    y = np.zeros(4,np.uint32)
    for i in range(len(y)):
        y[i] = S[i,bytes[i]]

    add1 = (y[0] + y[1])%(2**32)
    a = add1 ^ y[2]
    b = (a + y[3])%(2**32)
    return b

def Blowfish_enc(plainText,key,P):
    P_ = np.zeros(18,np.uint32)
    for i in range(18):
        P_[i] = P[i] ^ key

    i = 1
    x_l_r = np.zeros(2,np.uint32)

    while plainText > 0:
        x_l_r[i] = plainText % np.uint64(0x100000000)
        i-=1
        plainText // = 0x100000000

    xl,xr = x_l_r[0],x_l_r[1]
    for _ in range(16):

```

```

xl,xr = round(xl,xr,_)

return ((xr ^ P_[17])<<32)+ ((xl ^ P_[16]))

def Blowfish_dec(cipher_text,key,P):
    P_ = np.zeros(18,np.uint32)
    for i in range(18):
        P_[i] = P[i] ^ key

    i = 1
    x_l_r = np.zeros(2,np.uint32)

    while cipher_text > 0:
        x_l_r[i] = cipher_text % np.uint64(0x100000000)
        i-=1
        cipher_text //= np.uint64(0x100000000)

    xl,xr = x_l_r[0],x_l_r[1]

    for _ in range(17,1,-1):
        xl,xr = round(xl,xr,_)

    return ((xr ^ P_[0])<<32) + ((xl ^ P_[1]))

```

Architettura complessiva

L'architettura complessiva descritta in VHDL è mostrata in figura 86. Si procede adesso ad analizzare nel dettaglio i singoli componenti.

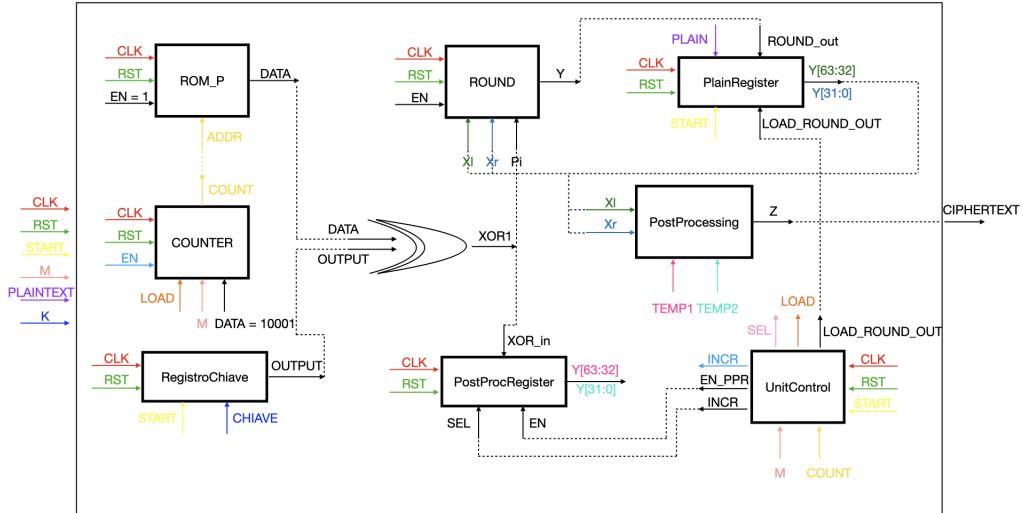


Figura 86: Architettura complessiva

Round

Entity e architecture - Round

Il componente Round è realizzato così come mostrato in figura 81.

```

entity Round is
port(
    CLK : in std_logic;
    RST : in std_logic;
    EN : in std_logic;
    Pi : in std_logic_vector(31 downto 0);
    Xl : in std_logic_vector(31 downto 0);
    Xr : in std_logic_vector(31 downto 0);
    Y : out std_logic_vector(63 downto 0)
);
end Round;

architecture Structural of Round is
component F
port(

```

```

CLK : in std_logic;
RST : in std_logic;
EN : in std_logic;
Xl : in std_logic_vector(31 downto 0);
Y : out std_logic_vector(31 downto 0));
end component;
signal xor1 : std_logic_vector(31 downto 0);
signal f_out : std_logic_vector(31 downto 0);
signal xor2 : std_logic_vector(31 downto 0);
begin
xor1 <= Pi xor Xl;
xor2 <= f_out xor Xr;
f1 : F
port map(
CLK => CLK,
EN => EN,
RST => RST,
Xl => xor1,
Y => f_out);
Y(63 downto 32) <= xor2;
Y(31 downto 0) <= xor1;
end Structural;

```

Il blocco F contiene 4 SBOX, ciascuna delle quali è un vettore di 256 locazioni da 32 bit. Ogni SBOX è implementata come una ROM che riceve come indirizzo 8 bit dei 32 in ingresso. Per realizzare le somme intermedie abbiamo fatto ricorso al sommatore ripple carry adder.

F block

Entity e architecture - F block

La descrizione VHDL del blocco F segue la figura 82.

```

entity F is
port(
CLK : in std_logic;
RST : in std_logic;
EN : in std_logic;

```

```

X1 : in std_logic_vector(31 downto 0);
Y : out std_logic_vector(31 downto 0));
end F;

architecture Structural of F is

component Sbox1
port(
CLK : in std_logic;
RST : in std_logic;
EN : in std_logic;
ADDR : in std_logic_vector(7 downto 0);
DATA : out std_logic_vector(31 downto 0));
end component;

component Sbox2
port(
CLK : in std_logic;
RST : in std_logic;
EN : in std_logic;
ADDR : in std_logic_vector(7 downto 0);
DATA : out std_logic_vector(31 downto 0)
);
end component;

component Sbox3
port(
CLK : in std_logic;
RST : in std_logic;
EN : in std_logic;
ADDR : in std_logic_vector(7 downto 0);
DATA : out std_logic_vector(31 downto 0));
end component;

component Sbox4
port(
CLK : in std_logic;
RST : in std_logic;

```

```

EN : in std_logic;
ADDR : in std_logic_vector(7 downto 0);
DATA : out std_logic_vector(31 downto 0));
end component;

component ripple_carry
port(
  X, Y: in std_logic_vector(31 downto 0);
  c_in: in std_logic;
  c_out: out std_logic;
  Z: out std_logic_vector(31 downto 0));
end component;
signal s : std_logic_vector(31 downto 0);
signal y1 : std_logic_vector(31 downto 0);
signal y2 : std_logic_vector(31 downto 0);
signal y3 : std_logic_vector(31 downto 0);
signal y4 : std_logic_vector(31 downto 0);
signal add1 : std_logic_vector(31 downto 0);
signal xor1 : std_logic_vector(31 downto 0);
begin
adder1 : ripple_carry
  port map(
    X => y1,
    Y => y2,
    c_in => '0',
    Z => add1);
xor1 <= add1 xor y3;
adder2 : ripple_carry
  port map(
    X => xor1,
    Y => y4,
    c_in => '0',
    Z => Y);
S1 : Sbox1
  port map(
    CLK => CLK,
    RST => '0',
    EN => EN,

```

```

    ADDR => X1(31 downto 24),
    DATA => y1);
S2 : Sbox2
port map(
    CLK => CLK,
    RST => '0',
    EN => EN,
    ADDR => X1(23 downto 16),
    DATA => y2);
S3 : Sbox3
port map(
    CLK => CLK,
    RST => '0',
    EN => EN,
    ADDR => X1(15 downto 8),
    DATA => y3 );
S4 : Sbox4
port map(
    CLK => CLK,
    RST => '0',
    EN => EN,
    ADDR => X1(7 downto 0),
    DATA => y4);
end Structural;

```

PlainRegister

Entity e architecture - PlainRegister

Per gestire la propagazione dei valori corretti all'interno dell'architettura abbiamo seguito il modello register transfer, ossia abbiamo inserito alcuni registri tra i circuiti combinatori per disaccoppiarli e poter memorizzare il valore corretto soltanto a fronte di opportuni segnali dell'unità di controllo. Qui è riportata la descrizione comportamentale del *PlainRegister*, che memorizza l'uscita del blocco Round. Quest'ultima dovrà poi essere retroazionata opportunamente in ingresso al blocco per effettuare le 16 iterazioni. Con il segnale di start, fornito dall'esterno, viene caricata in fase di inizializzazione la stringa da cifrare/decifrare su 64 bit. Successivamente l'output del blocco

round sarà memorizzato nel registro solo quando il segnale *load_round_out* si alzerà.

```
entity Plain_register is
  generic ( N :integer := 64);
  port(
    CLK : in std_logic;
    RST : in std_logic;
    Plain : in std_logic_vector(N-1 downto 0);
    Round_out : in std_logic_vector(N-1 downto 0);
    Y : out std_logic_vector(N-1 downto 0);
    start : in std_logic;
    load_round_out : in std_logic);
end Plain_register;

architecture behavioural of Plain_register is
  signal temp: std_logic_vector(N -1 downto 0) := (others => '0');
begin
  SR: process(CLK, RST)
  begin
    if(RST = '1') then
      temp <= (others=>'0');
    elsif(rising_edge(CLK)) then
      if(start = '1') then
        temp(N-1 downto 0) <= Plain;
      elsif(load_round_out = '1') then
        temp(N-1 downto 0) <= Round_out;
      end if;
    end if;
  end process;
  Y <= temp;
end behavioural;
```

Registro chiave

Entity e architecture - Registro chiave

Per il registro chiave si è scelta una descrizione comportamentale. Quest'ultimo memorizza il valore della chiave in fase di inizializzazione quando start è alto.

```
entity registroChiave is
  generic(N : integer := 32);
  port(
    chiave : in std_logic_vector(N-1 downto 0);
    CLK, RST, start: in std_logic;
    output: out std_logic_vector(N-1 downto 0) := (others => '0')
  );
end registroChiave;

architecture behavioural of registroChiave is
begin
  proc: process(CLK, RST)
  begin
    if(RST = '1') then
      output <= (others=>'0');
    elsif(rising_edge(CLK)) then
      if (start = '1') then
        output <= chiave;
      end if;
    end if;
  end process;
end behavioural;
```

Post processing unit

Entity e architecture - Post processing unit

In accordo alla figura 83 viene realizzata l'unità che effettua il post processing alla fine dei 16 round. I due valori del vettore P necessari per questa fase, in xor con la chiave, sono precaricati opportunamente in un registro.

Post processing register

Entity e architecture - Post processing register

Il registro PPR deve contenere i due valori del vettore P in xor con la chiave da utilizzare per la fase di post processing. Tale registro riceverà un segnale di abilitazione dall'unità di controllo per memorizzare il valore di ingresso e un opportuno segnale di selezione, dipendente dal conteggio, che indirizza la prima o la seconda parte del registro.

```
entity Post_proc_register is
  generic ( N :integer := 32);
  port(
    CLK : in std_logic;
    RST : in std_logic;
    SEL : in std_logic;
    EN : in std_logic;
    Xor_in : in std_logic_vector(N-1 downto 0);
    Y : out std_logic_vector(2*N-1 downto 0)
  );
end Post_proc_register;

architecture behavioural of Post_proc_register is
  signal temp: std_logic_vector(2*N-1 downto 0) := (others => '0');
begin
  SR: process(CLK, RST)
  begin
    if(RST = '1') then
      temp <= (others=>'0');
    elsif(rising_edge(CLK)) then
      if(EN = '1') then
        if (SEL = '1') then
          temp(2*N-1 downto N) <= Xor_in;
        else
          temp(N-1 downto 0) <= Xor_in;
        end if;
      end if;
    end if;
  end if;
end;
```

```

end process;
Y <= temp;

end behavioural;
```

Contatore

Entity e architecture - Contatore

Viene qui riportata la descrizione comportamentale del contatore utilizzato per la scansione degli indirizzi delle ROM. Quest'ultimo supporta due modalità di funzionamento, dovendo incrementarsi o decrementarsi per scandire opportunamente la ROM P nella fase di cifratura o decifratura.

```

entity Counter_address is
  Port(
    CLK : in STD_LOGIC;
    RST : in STD_LOGIC;
    EN : in STD_LOGIC;
    M : in STD_LOGIC;
    LOAD : in STD_LOGIC;
    DATA : in STD_LOGIC_VECTOR (4 downto 0);
    counter : out STD_LOGIC_VECTOR (4 downto 0);
    DIV : out std_logic);
end Counter_address;

architecture Behavioral of Counter_address is
  signal c : std_logic_vector (4 downto 0) := (others => '0');
begin
  counter <= c;
  counter_process: process(CLK)
  begin
    if(rising_edge(CLK)) then
      if RST = '1' then
        DIV <= '0';
        c <= (others => '0');
      elsif(load = '1') then
        c <= data;
      elsif (EN = '1' and M = '0') then
```

```

if(c = "10001") then
  DIV <= '1';
  c <= "00000";
else
  c <= std_logic_vector(unsigned(c) + 1);
  DIV <= '0';
end if;
elsif (EN = '1' and M = '1') then
  if(c = "00000") then
    DIV <= '1';
    c <= "10001";
  else
    c <= std_logic_vector(unsigned(c) - 1);
    DIV <= '0';
  end if;
end if;
end if;
end if;
end process;
end Behavioral;

```

Unità di controllo

Entity e architecture - Unità di controllo

Viene qui riportata la descrizione VHDL dell'unità di controllo e del relativo automa in figura 87. Attraverso un opportuno bit l'unità di controllo è in grado di fornire segnali di controllo sia per cifrare che decifrare l'input.

- Idle: si attende che il segnale di start si alzi
- Load_reg: si attende un ciclo di clock per caricare il testo in chiaro/cifrato e la chiave nei rispettivi registri
- M_wait: si attende che l'uscita della ROM P si stabilizzi in caso di decifratura
- Round1: inizia la fase di round e si alza il segnale incr
- Round_wait: si abbassa il segnale incr e si abilita il registro PlainRegister a memorizzare l'output della fase di round i

- Round2: si abbassa l'abilitazione di PlainRegister e si valuta se la fase di cifratura o decifratura è terminata, andando nuovamente nello stato Round1 o nella fase di post processing
- Pp1: viene memorizzato il primo valore nel registro PPR
- Pp2: viene memorizzato il secondo valore nel registro PPR e viene ora calcolato l'output finale corretto

```

entity control_unit is
port(
    CLK : in std_logic;
    RST : in std_logic;
    start : in std_logic;
    m : in std_logic;
    load : out std_logic;
    incr : out std_logic;
    count : in std_logic_vector(4 downto 0);
    En_ppr : out std_logic;
    sel : out std_logic;
    load_round_out : out std_logic
);
end control_unit;

architecture Behavioral of control_unit is

type stato is (idle, load_reg, M_wait1, round1, round_wait, round2, pp1, pp2);
signal stato_corr : stato := idle;
begin
    proc : process(CLK)
    begin
        if (rising_edge(CLK)) then
            if (RST = '1') then
                stato_corr <= idle;
            else
                case stato_corr is
                    when idle =>
                        En_ppr <= '0';

```

```

if (start = '1') then
  stato_corr <= load_reg;
end if;

when load_reg =>
  if(M ='1') then
    LOAD <= '1';
    stato_corr <= M_wait1;
  else
    stato_corr <= round1;
  end if;

when M_wait1 =>
  LOAD <= '0';
  stato_corr <= round1;

when round1 =>
  incr <= '1';
  stato_corr <= round_wait;

when round_wait =>
  incr <= '0';
  load_round_out <= '1';
  stato_corr <= round2;

when round2 =>
  load_round_out <= '0';
  if (count = "10000" and M = '0') then
    stato_corr <= pp1;
    En_ppr <= '1';
    incr <= '1';
  elsif (count = "00001" and M = '1') then
    stato_corr <= pp1;
    En_ppr <= '1';
    incr <= '1';
  else
    stato_corr <= round1;
  end if;

```

```

when pp1 =>
  sel <= '0';
  stato_corr <= pp2;

when pp2 =>
  incr <= '0';
  sel <= '1';
  stato_corr <= idle;
end case;
end if;
end if;
end process;
end Behavioral;

```

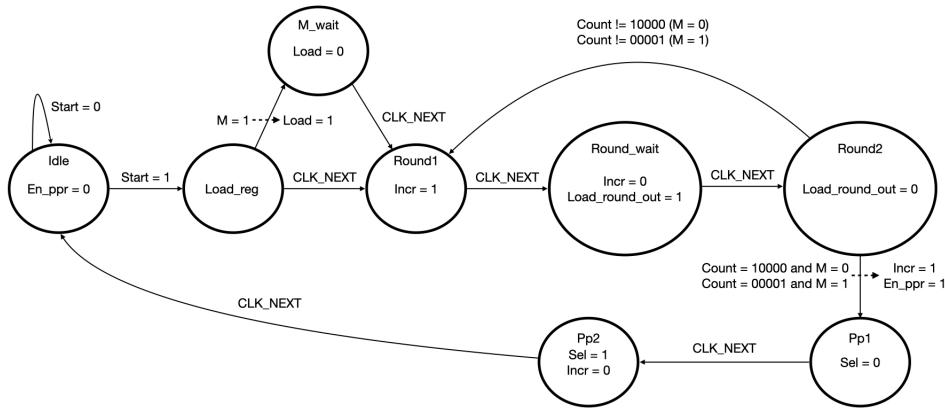


Figura 87: Unità di controllo Blowfish

Simulazione - Blowfish

Per testare la nostra macchina finale abbiamo confrontato i risultati intermedi prodotti in simulazione con quelli della nostra implementazione in python. Essi sono riportati in figure 88, 89, 90 e 91.

```

Plain text: 0x123456abcd132536
Key: 0xaabb0918
Round 0: 0x77b3ba639cb0353b
Round 1: 0x11bfe5e558abbbba8
Round 2: 0x6e21e78a81d66d3
Round 3: 0x165bb466af296424
Round 4: 0x19ca40d218e9855c
Round 5: 0x4765fb7e9aee781a
Round 6: 0x61c8b1a5e5f008fe
Round 7: 0x5ca47136273dd434
Round 8: 0xab26efbb33759c8
Round 9: 0x66f01445394df5d4
Round 10: 0xe66c2371721f7b92
Round 11: 0xc8b790a783e2605
Round 12: 0x494d16b0669c59a5
Round 13: 0x9e18d1002a8a4f75
Round 14: 0x4be2e8c7b270dad
Round 15: 0xa2f7ac70541ee8c8
Cipher text: 0x77dc1acb9a5a70b1

```

Figura 88: Cifratura

```

Cipher text: 0x77dc1acb9a5a70b1
Key: 0xaabb0918
Round 17: 0x338ad16c541ee8c8
Round 16: 0x35764f7ab270dad
Round 15: 0xf3a385082a8a4f75
Round 14: 0x1bf97fc0669c59a5
Round 13: 0x18085b3d783e2605
Round 12: 0xa7fff0a0721f7b92
Round 11: 0xa7d8361f394df5d4
Round 10: 0xb556ce5bb33759c8
Round 9: 0xa632000273dd434
Round 8: 0xdc1b1d8e5f008fe
Round 7: 0xba7c76dc9aee781a
Round 6: 0x2c0d5cec18e9855c
Round 5: 0xa6af57e9af296424
Round 4: 0xf160c1fa481d66d3
Round 3: 0x2512b60d58abbbba8
Round 2: 0xe20b24fd9cb0353b
Plain text: 0x123456abcd132536

```

Figura 89: Decifratura

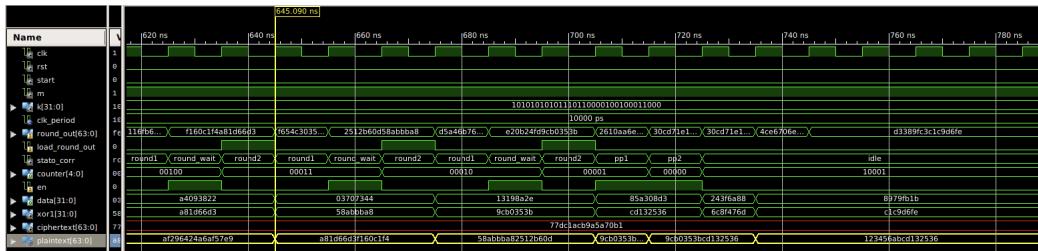


Figura 90: Simulazione decifratura

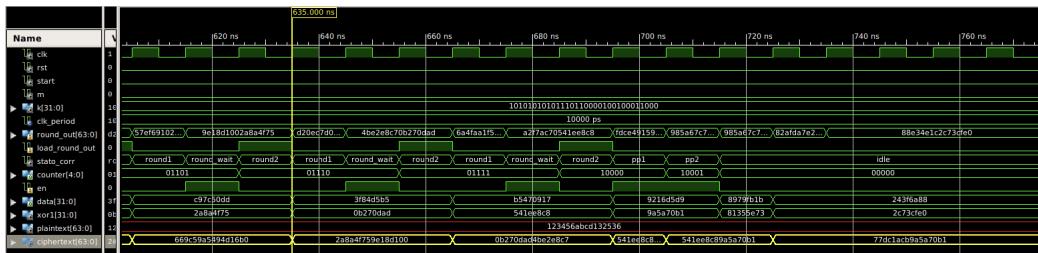


Figura 91: Simulazione cifratura