

# Indice

<b>1 Esercizio 1 - Multiplexer</b>	<b>3</b>
1.1 Parte 1 . . . . .	3
1.2 Parte 2 . . . . .	8
1.3 Parte 3 . . . . .	11
<b>2 Esercizio 2 - Encoder BCD</b>	<b>12</b>
2.1 Traccia . . . . .	12
2.2 Soluzione . . . . .	12
2.3 Codice . . . . .	14
2.4 Simulazione . . . . .	20
<b>3 Esercizio 3 - Riconoscitore di Sequenze</b>	<b>22</b>
3.1 Traccia . . . . .	22
3.2 Soluzione . . . . .	22
<b>4 Esercizio 4 - Shift Register</b>	<b>31</b>
4.1 Traccia . . . . .	31
4.2 Approccio comportamentale . . . . .	31
4.3 Approccio Strutturale . . . . .	33
4.4 Simulazione . . . . .	38
<b>5 Esercizio 5 - Cronometro</b>	<b>39</b>
5.1 Traccia . . . . .	39
5.2 Sintesi su fpga . . . . .	45
<b>6 Esercizio 6 - Sistema di Testing</b>	<b>51</b>
6.1 Traccia . . . . .	51
6.2 Implementazione . . . . .	51
6.3 Simulazione e Sintesi . . . . .	58
<b>7 Esercizio 7 - Comunicazione Handshaking</b>	<b>60</b>
7.1 Introduzione all'handshaking . . . . .	60
7.2 Sistema A . . . . .	63
7.2.1 Unità Operativa A . . . . .	64
7.2.2 Unità di Controllo A . . . . .	66
7.3 Sistema B . . . . .	69
7.3.1 Unita Operativa B . . . . .	70

7.3.2 C B . . . . .	74
7.4 Considerazioni Finali . . . . .	77
<b>8 Esercizio 8 - Processor</b>	<b>78</b>
8.1 Introduzione a IJVM . . . . .	78
8.2 Unità Operativa . . . . .	81
8.3 Unità di Controllo . . . . .	84
8.4 Analisi delle istruzioni IADD e ISTORE . . . . .	89
8.4.1 Istruzione IADD . . . . .	89
8.4.2 Istruzione ISTORE . . . . .	92
8.4.3 Modifica istruzione IADD . . . . .	96
<b>9 Esercizio 9 - Interfaccia UART</b>	<b>97</b>
9.1 Introduzione all'UART . . . . .	97
9.2 Parte 1 . . . . .	100
9.3 Parte 2 . . . . .	103
9.3.1 Sistema A . . . . .	103
9.3.1.1 Unità Operativa A . . . . .	105
9.3.1.2 Unità di Controllo A . . . . .	106
9.3.2 Sistema B . . . . .	110
9.3.2.1 Unità Operativa B . . . . .	111
9.3.2.2 Unità di Controllo B . . . . .	112
<b>10 Esercizio 10 - Switch Multistadio</b>	<b>116</b>
10.1 Approccio utilizzato . . . . .	116
10.2 Parte Operativa . . . . .	117
10.3 Parte di Controllo . . . . .	120
10.4 Simulazione . . . . .	126
10.5 Considerazioni finali . . . . .	128
<b>11 Esercizio 11 - Divisore Restoring</b>	<b>129</b>
11.1 Introduzione . . . . .	129
11.2 Unità Operativa . . . . .	131
11.3 Unità di Controllo . . . . .	137
11.4 Simulazione e Sintesi . . . . .	143
<b>12 Esercizio 12 - Interfaccia VGA</b>	<b>146</b>

# 1 Esercizio 1 - Multiplexer

## 1.1 Parte 1

L'esercizio 1.1 richiede la rappresentazione di un multiplexer 16:1 tramite la composizione di multiplexer 4:1, quindi definiamo il module del componente mux\_4\_1 definito in modo Dataflow, e poi per le proprietà della modularità definiamo il module mux\_16\_1 come composizione dei precedenti, tramite il costrutto *for..generate*.

Il multiplexer 4:1 ha come ingressi quattro bit, identificati col vettore  $a(0$  to  $3)$ , e  $\lceil \log_2(n) \rceil$  segnali di abilitazione, dove  $n$  è il numero di segnali di ingresso. In questo caso, quindi, due segnali di abilitazione ed uno di uscita.

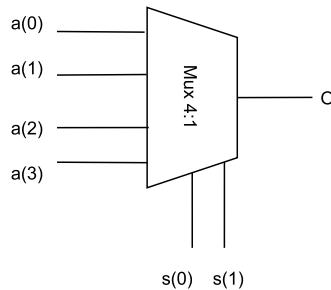


Figure 1: Mux 4:1

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

```
architecture DataFlow of mux_4_1 is

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

Figure 2: Mux 4:1 Dataflow

Composto da un costrutto *when...else* che suddivide i diversi casi e gestisce anche tutti i casi non definiti con l'ultima clausola else senza alcuna condizione.

Definito l'elemento base del progetto, si passa a comporre il multiplexer 16:1 tramite un approccio strutturale, nel quale generiamo cinque mux\_4\_1: i primi quattro avranno gli ingressi interfacciati con l'esterno e, tramite segnali interni, le loro uscite sono collegate come ingressi dell'ultimo multiplexer che costituisce l'uscita del sistema. La macchina completa presenta quindi 16 segnali di ingresso, 4 segnali di selezione ed un unico segnale di uscita:

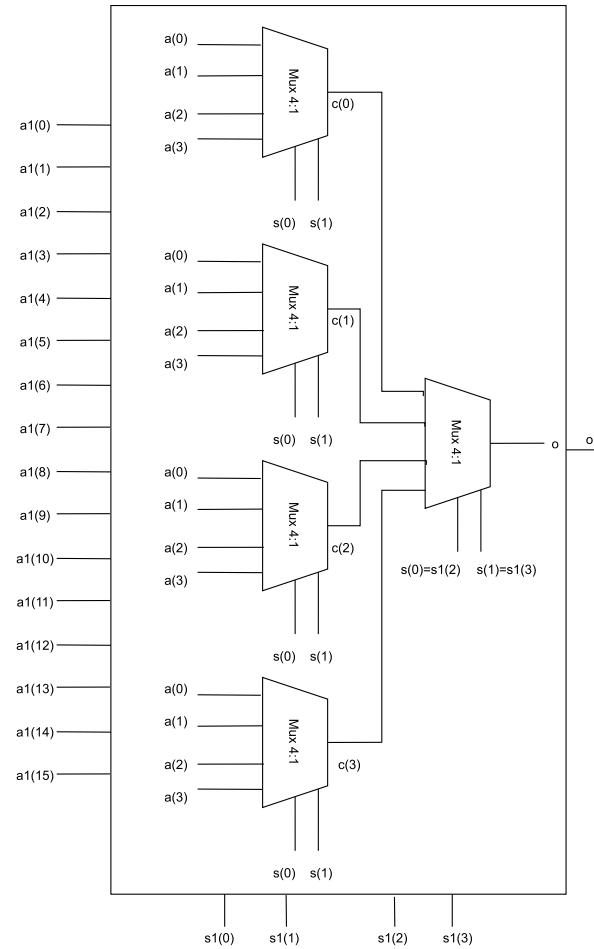


Figure 3: Mux 16:1

Lo schema sopra rappresentato è tradotto in linguaggio VHDL dal seguente codice, dove l'entità mux\_16\_1 è rappresentata con approccio strutturale e, tramite i segnali interni c(0 to 3), collegiamo le uscite parziali dei primi quattro mux\_4\_1, identificati con la label mux\_4\_1\_in, con i quattro ingressi dell'ultimo mux\_4\_1, identificato con la label mux\_4\_1\_fin.

```

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

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

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

```

Figure 4: Mux 16:1 Dataflow

Progettato il mux\_16\_1, è possibile testarlo attraverso un testbench. La prima cosa che bisogna specificare è che il corpo dell'entity è vuoto, questo perché non si tratta di oggetto che realizziamo, ma serve solo per effettuare la simulazione e verificare se il sistema realizzato funziona correttamente. Il testbench effettivamente non ha né segnali d'ingresso né d'uscita, ma sfrutta per i test i segnali interni definiti nel codice. Per testare il mux\_16\_1 definito precedentemente, abbiamo istanziato una uut (Unit Under Test) in cui collegiamo le varie porte ai segnali (input, selection, output).

```

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

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

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

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

Figure 5: Mux 16:1 Testbench

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

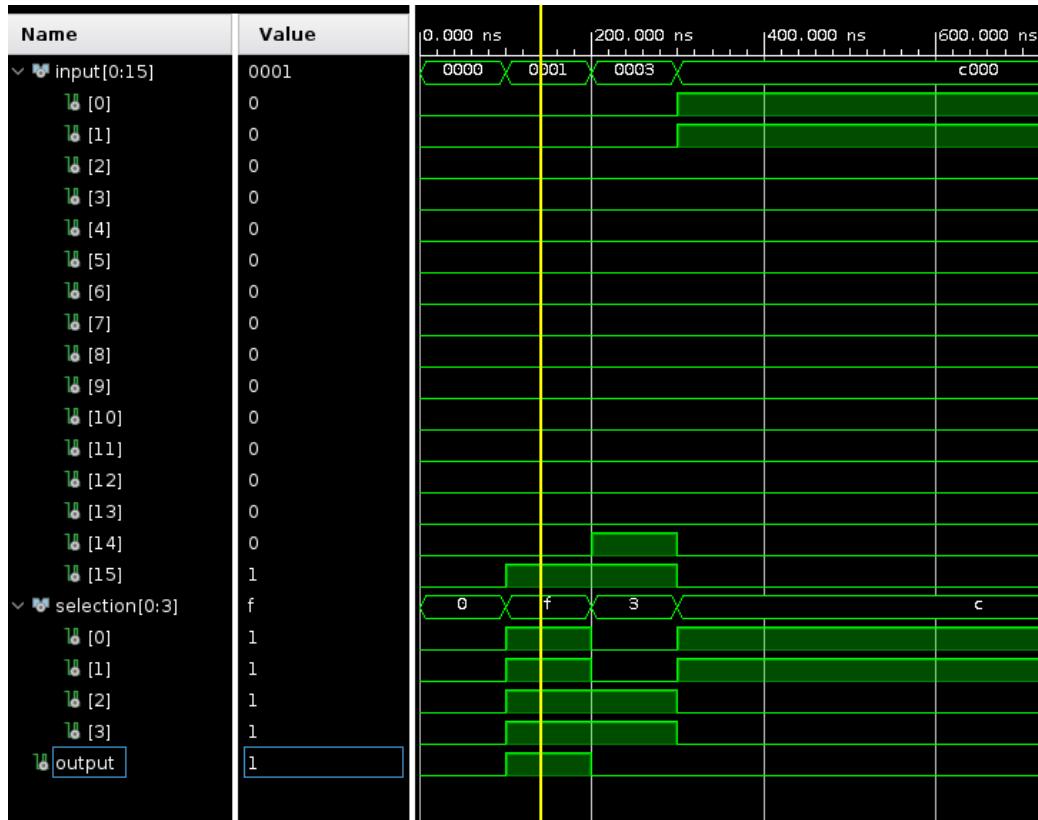


Figure 6: Mux 16:1 Simulazione

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

## 1.2 Parte 2

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

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

Figure 7: Demux 1:4 Dataflow

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

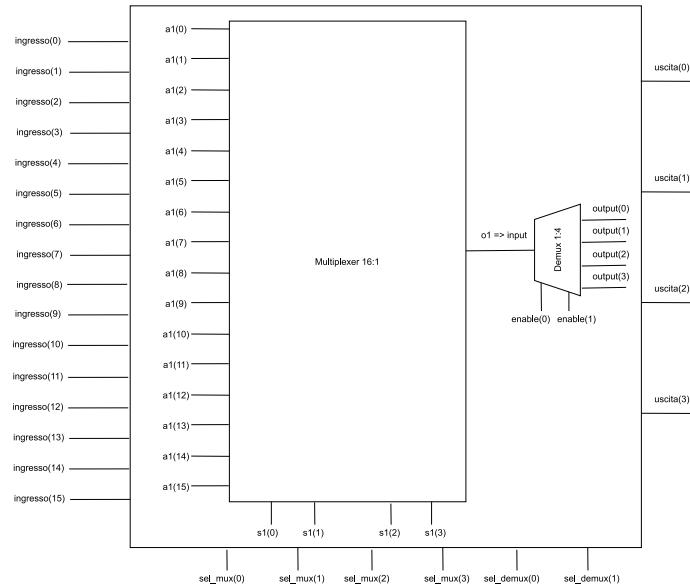


Figure 8: Rete 16:1

Una volta eseguite le interconnessioni tramite un unico segnale interno, utilizzato per collegare l'uscita del mux\_16\_1 con l'ingresso del demux\_1\_4, si ottiene la rete 16:4: tale rete presenta 16 segnali di ingressi totali, 6 di selezione (di cui 2 utilizzati per il demux) e 4 segnali di uscita.

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

end Structural;
```

Figure 9: Rete 16:4 Structural

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

```

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

rete_16_4_2: i <= "0000001000000000",
"00000000000000" after 100ns,
"10000000000000" after 200ns,
"1110111111111111" after 300ns;

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

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

```

Figure 10: Rete 16:4 Testbench

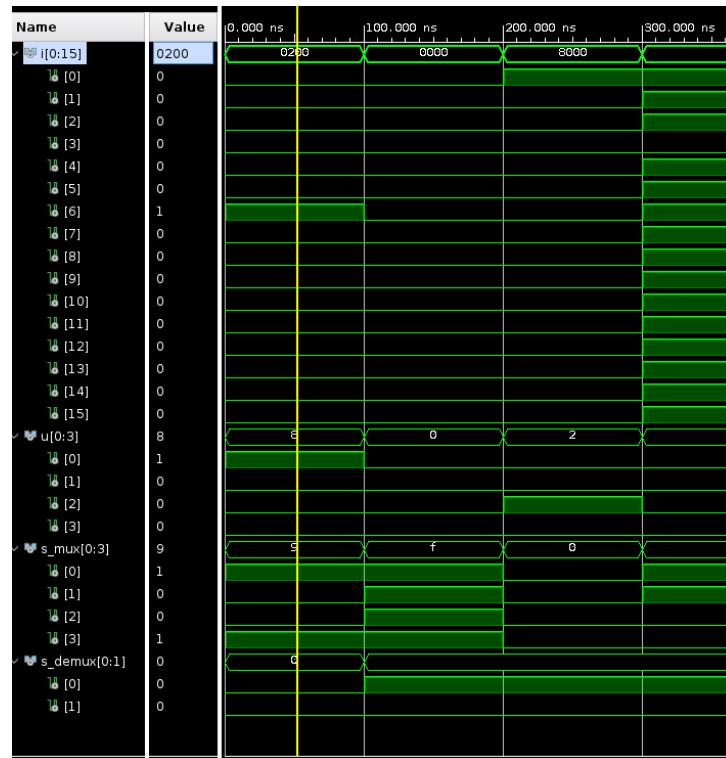


Figure 11: Rete 16:4 Simulazione

### 1.3 Parte 3

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

```
##Switches
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMS33 } [get_ports { sel_mux[0] }]; #IO_L24N_T3_R5B_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMS33 } [get_ports { sel_mux[1] }]; #IO_L3W_T3_D05_EMCLK_15 Sch=sw[1]
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMS33 } [get_ports { sel_mux[2] }]; #IO_L24P_T3_D05_EMCLK_15 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMS33 } [get_ports { sel_mux[3] }]; #IO_L3W_T2_MRC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMS33 } [get_ports { sel_mux[4] }]; #IO_L12N_T1_MRC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMS33 } [get_ports { sel_mux[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
#set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMS33 } [get_ports { input[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
#set_property -dict { PACKAGE_PIN V13 IOSTANDARD LVCMS33 } [get_ports { input[7] }]; #IO_L5W_T0_D07_14 Sch=sw[7]
#set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMS33 } [get_ports { SW[8] }]; #IO_L24P_T3_34 Sch=sw[8]
#set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMS33 } [get_ports { SW[9] }]; #IO_L24P_T3_34 Sch=sw[9]
#set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMS33 } [get_ports { SW[10] }]; #IO_L24P_T3_34 D05_FDR_B_14 Sch=sw[10]
#set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMS33 } [get_ports { SW[11] }]; #IO_L24P_T3_D10_D19_14 Sch=sw[11]
#set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
#set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
#set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMS33 } [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
#set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMS33 } [get_ports { SW[15] }]; #IO_L21P_T3_D05_14 Sch=sw[15]

## LEDs
set_property -dict { PACKAGE_PIN H7 IOSTANDARD LVCMS33 } [get_ports { uscita[0] }]; #IO_L10P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K5 IOSTANDARD LVCMS33 } [get_ports { uscita[1] }]; #IO_L24P_T2_R01_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J3 IOSTANDARD LVCMS33 } [get_ports { uscita[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N13 IOSTANDARD LVCMS33 } [get_ports { uscita[3] }]; #IO_L10P_T1_D11_14 Sch=led[3]
#set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMS33 } [get_ports { output[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMS33 } [get_ports { output[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
```

Figure 12: Constraints

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

Il sistema finale è quello raffigurato nella seguente figura

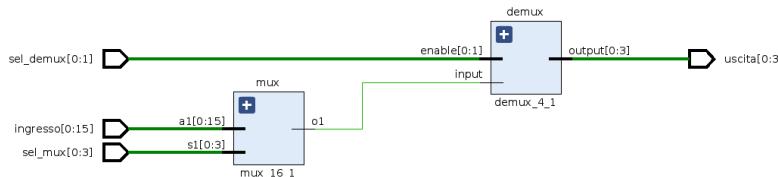


Figure 13: Rete 16:4 Analysis

## 2 Esercizio 2 - Encoder BCD

### 2.1 Traccia

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

Input: 0000000001 ⇒ Output: 0000 (cifra 0)

Input: 0000000010 ⇒ Output: 0001 (cifra 1)

Input: 0000000100 ⇒ Output: 0010 (cifra 2)

....

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

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

### 2.2 Soluzione

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

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

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

### **Arbitro a priorità:**

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

### **Encoder 10:4:**

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

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

### **Display Manager:**

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

La rete complessiva dispone dell'encoder e del display manager. Il vettore di ingresso di 10 bit andrà nell'encoder e la sua uscita su 4 bit, mediante un segnale di interconnessione, andrà in ingresso al display manager che mostra il valore di uscita sul display.

## 2.3 Codice

### Arbitro

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

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

architecture dataflow of arbitro is

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

Figure 14: Codice Arbitro

## Encoder 10:4

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

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

architecture dataflow of encoder10_4 is

begin
    with x select
        y <= "0000" when "0000000001",
                    "0001" when "0000000010",
                    "0010" when "00000000100",
                    "0011" when "00000001000",
                    "0100" when "00000010000",
                    "0101" when "00000100000",
                    "0110" when "00001000000",
                    "0111" when "00100000000",
                    "1000" when "01000000000",
                    "1001" when "10000000000",
                    "----" when others;
end dataflow;
```

Figure 15: Codice Encoder 10:4

## Encoder complessivo

L'encoder complessivo è descritto a livello strutturale utilizzando i componenti “arbitro” e “encoder10\_4”. All'interno del sistema è definito un segnale  $t$  di tipo std\_logic\_vector(9 downto 0), che fa da interconnessione e viene utilizzato come uscita dell'arbitro e come ingresso dell'encoder 10:4.

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

architecture structural of encoder is

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

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

  signal t: std_logic_vector(9 downto 0);

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

Figure 16: Codice Encoder Complessivo

## Display Manager

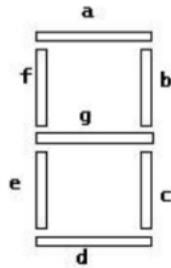


Figure 17: Display Segment

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

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

```

architecture behavioral of display_manager is

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


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

end behavioral;

```

Figure 18: Codice Display Manager

## Sistema Completo

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

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

architecture structural of sistema_completo is

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

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

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

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

Figure 19: Codice Sistema Completo

## 2.4 Simulazione

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

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

architecture Behavioral of sim_encoderfinale is

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

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

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

Figure 20: Codice Testbench

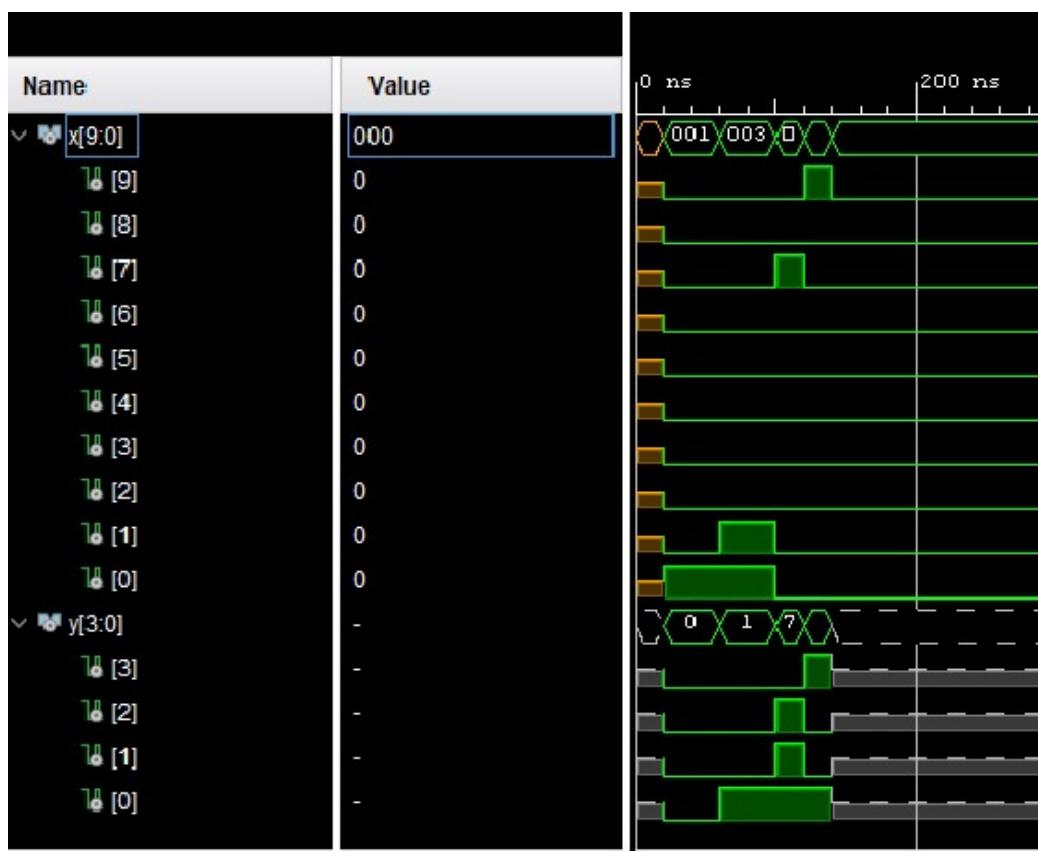


Figure 21: Simulazione Encoder

## 3 Esercizio 3 - Riconoscitore di Sequenze

### 3.1 Traccia

Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza 1001. La macchina prende in ingresso un segnale binario  $i$  che rappresenta il dato, un segnale  $A$  di 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.

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.

### 3.2 Soluzione

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

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

## Sistema per il riconoscimento

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

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

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

Figure 22: Codice Sistema Pt1

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

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

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

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

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

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

```

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

```

```

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

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

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

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

end Behavioral;

```

Figure 23: Codice Sistema Pt2

## Debouncer

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



Figure 24: Debouncer

- Stato not pressed: il sistema permane in questo stato finchè non vede il segnale in ingresso alzarsi. Passa così nello stato pressed.
- Stato pressed: il sistema permane in questo stato finchè il segnale in ingresso non si abbassa. Quando ciò accade, si utilizza una variabile di conteggio, la quale fa sì che passi un periodo di tempo pari a  $D_2$  prima di riportare il segnale come alto in uscita e tornare nello stato not pressed.

Il segnale in uscita rimane alto per un periodo di clock dato che, quando il sistema ritorna nello stato not pressed, l'uscita viene abbassata nuovamente. A differenza di un debouncer classico, l'uscita viene riportata alta solo dopo che il segnale si è abbassato e non dopo un tempo  $D_1$  dal fronte di salita; la scelta è stata fatta poiché, altrimenti, se il bottone venisse premuto troppo a lungo potrebbe essere rilevato nuovamente ed in uscita si produrrebbero 2 segnali puliti invece di 1.

## Filtro per il modo

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

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

architecture Behavioral of filtro_modo is

    signal temp: std_logic;
begin

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

Figure 25: Codice filtro modo

## Gestore per il modo

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

```

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

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

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

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

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

signal cb_temp: std_logic;

begin

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

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

end Structural;

```

Figure 26: Codice gestore modo

## Riconoscitore

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

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

architecture Structural of riconoscitore is

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

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

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

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

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

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

```

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

begin

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

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

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

end Structural;

```

Figure 27: Codice riconoscitore

## Simulazione

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

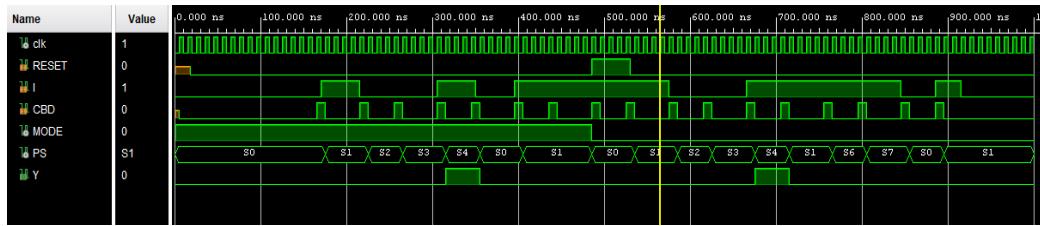


Figure 28: Simulazione riconoscitore

## 4 Esercizio 4 - Shift Register

### 4.1 Traccia

#### Shift register

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

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

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

Figure 29: Entity Shift Bidirezionale

### 4.2 Approccio comportamentale

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

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

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

```

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

output <= reg;

) end Behavioral;

```

Figure 30: Architecture Shift Register Bidirezionale

### 4.3 Approccio Strutturale

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

**Component Ffd\_bidirezionali:**

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

Figure 31: Entity Flip Flop D

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

```

architecture Behavioral of ffd_bidirezionale is
signal temp: std_logic;
begin
| delta: process(clk)
begin
    if(clk='1' and clk'event) then
        if(reset='1') then
            temp <= '0';
        elsif(a='1') then
            if(mode ='0') then
                temp <= rightInput;
            else
                temp <= leftInput;
            end if;
        end if;
    end if;
end process;
output <= temp;
end Behavioral;

```

Figure 32: Architecture Flip Flop D

### **Component mux\_2\_1:**

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

```

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

architecture Behavioral of mux_2_1 is
begin

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

end Behavioral;

```

Figure 33: Mux 2:1

Vediamo ora come tali componenti sono stati combinati:

```

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

ff_with_righth0:if i = N-1 generate
    mux_right: mux_2_1 port map( x1 => output(i+1), x2 => '0', s => shift, y => muxRight(i));
    mux_left: mux_2_1 port map( x1 => output(i-1), x2 => output(i-2), s => shift, y => muxleft(i));
    ff: ffd_bidirezionale PORT MAP(a => a, leftInput => muxleft(i) , clk => clk, reset => reset,mode => mode,
                                    output => output(i), rightInput => muxRight(i));
end generate ff_with_righth0;

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

```

Figure 34: Generate FFD

- **PrimoFF:** è il primo flip flop da sinistra, ovvero quello di posizione 1. È stato differenziato dagli altri poiché presenterà il multiplexer solo sull’ingresso destro, dato che il sinistro è legato all’input del registro.
- **UltimoFF:** è il primo flip flop da destra, ovvero quello di posizione N. È stato differenziato dagli altri poiché presenterà il multiplexer solo sull’ingresso sinistro, dato che il destro è legato all’input del registro.
- **FF\_with0:** sono i flipflop che, nel multiplexer per la selezione, presentano degli 0, in quanto non hanno abbastanza flipflop che li precedono. Es: nel flipflop di posizione 2, per l’ingresso sinistro, la scelta ricadrà tra l’uscita del flipflop di posizione 1 ed uno 0, dato che non esiste un flipflop di posizione 0.
- **FFintermedi:** sono quei flipflop che presentano multiplexer si per l’ingresso di destra che di sinistra, ed entrambi prendono gli ingressi da flipflop precedenti.

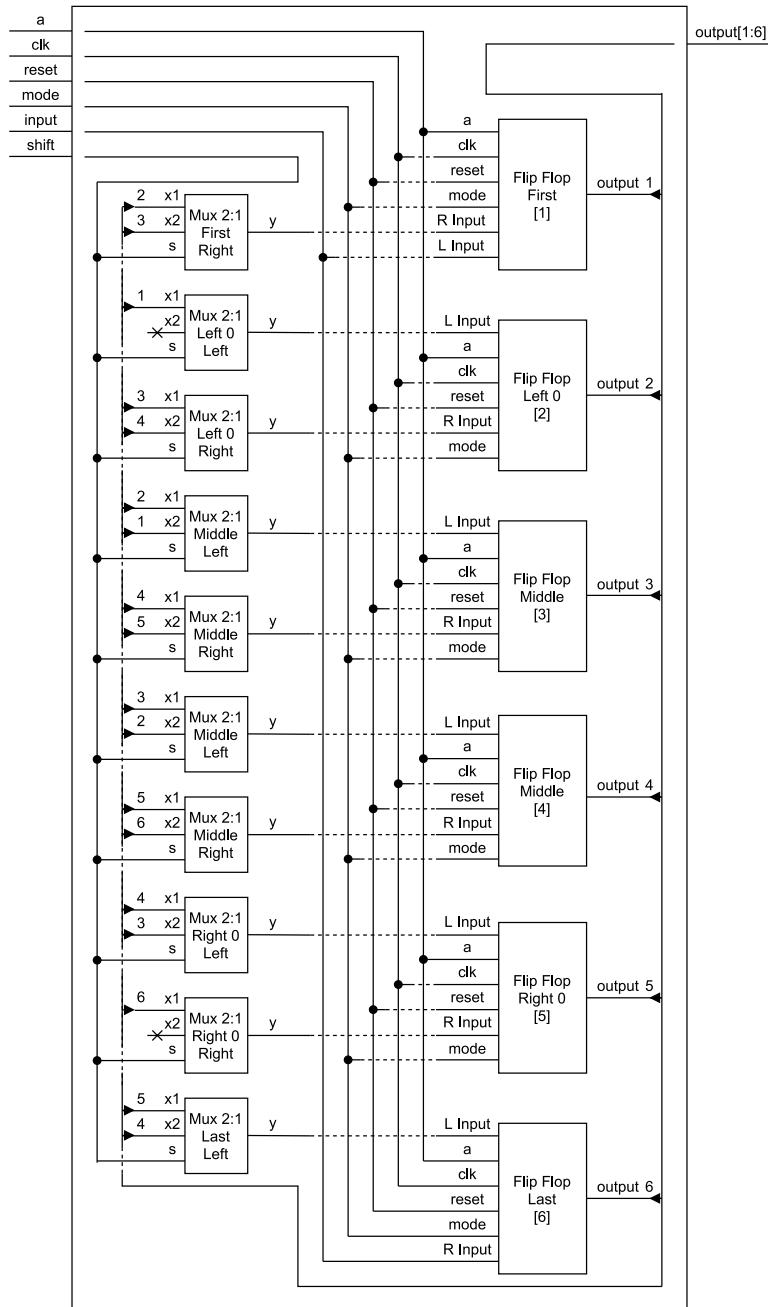


Figure 35: Progetto Completo Esercizio 1

## 4.4 Simulazione

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

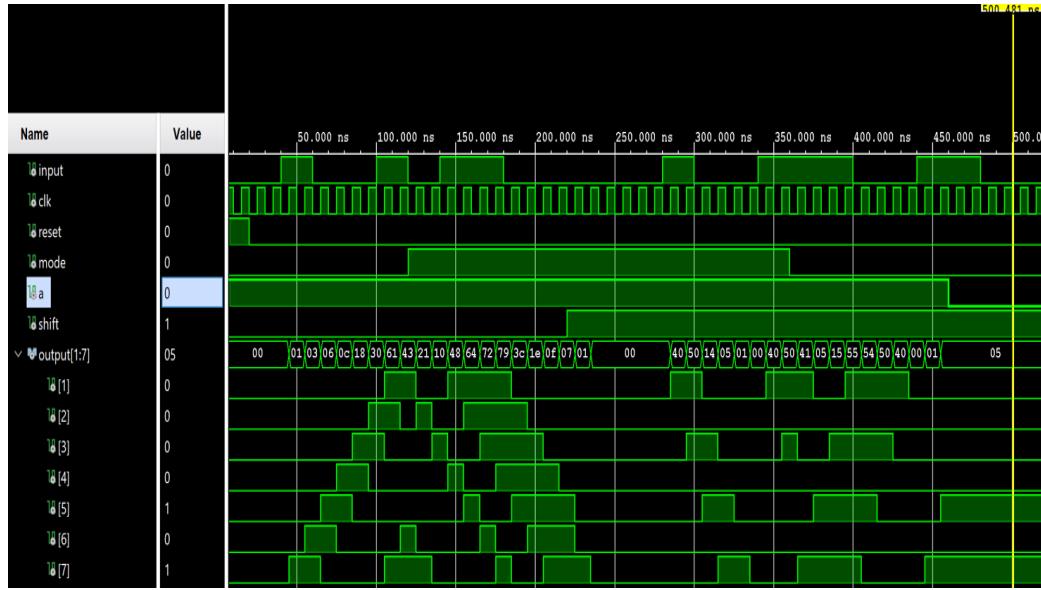


Figure 36: Simulazione Shift Register

## 5 Esercizio 5 - Cronometro

### 5.1 Traccia

#### Cronometro

L'obiettivo di questo esercizio è quello di progettare un cronometro tramite un approccio strutturale, visualizzando i valori di ore, minuti e secondi sul display presente sulla board. Il progetto prevede sicuramente l'utilizzo di una serie di contatori, i quali dovranno scandire il tempo.

#### Contatore:

```
entity contatore is
  Generic( N:positive);
  Port ( clock : in STD_LOGIC;
         reset : in STD_LOGIC;
         ab : in STD_LOGIC; -- abilitazione da contatori precedenti
         set : in STD_LOGIC; -- set esterno valore iniziale
         input: in std_logic_vector ( 1 to integer( ceil( log2(real(N)) ) ) ); -- input valore iniziale da set
         output : out STD_LOGIC_VECTOR (1 to integer( ceil( log2(real(N)) ) ) ); -- valore del conteggio
         div : out STD_LOGIC); -- ab in uscita per prossimo contatore
end contatore;
```

Figure 37: Entity Contatore

Tale contatore è stato implementato tramite un approccio comportamentale, in modo da incrementare il valore di conteggio solo in presenza di un opportuno segnale di abilitazione.

```
architecture Behavioral of contatore is
begin
process (clock)
begin
  if( clock = '1' and clock'event) then
    div <= '0';
    if(reset = '1') then
      tmp <= (others => '0');
    elsif ( set = '1') then
      tmp <= input;
    elsif ( ab = '1') then
      if ( tmp >= N-1) then
        tmp <= (others => '0');
        div <= '1';
      else
        tmp <= tmp + 1;
      end if;
    end if;
  end if;
end process;
output <= tmp;
end Behavioral;
```

Figure 38: Architecture Contatore

Data la richiesta di poter settare i valori di secondi, minuti e ore da cui il cronometro deve partire, si è deciso di introdurre due ingressi aggiuntivi: un segnale di set ed un vettore d'ingresso. Sostanzialmente, quando il segnale di set è alto, il contatore non rileva il segnale di abilitazione e carica il valore presente in ingresso; quando il segnale di set si abbassa, il contatore ritorna al suo normale funzionamento. Una volta definiti tre contatori differenti per scandire ore, minuti e secondi, vediamo come questi sono stati combinati tra loro.

## Sistema Contatori:

```


) entity sistema_contatori is
    Port ( clock : in STD_LOGIC;
            reset : in STD_LOGIC;
            set : in STD_LOGIC;
            input: in std_logic_vector(1 to 17 ) := (others => '0');
            output : out STD_LOGIC_VECTOR (1 to 17));
) end sistema_contatori;
```

Figure 39: Entity Sistema Contatori

E' formato da 4 contatori, 1 blocco di delay ed un blocco AND. Il primo contatore che ritroviamo è un divisore di frequenza, il quale ha il compito di passare dalla frequenza del clock della board a quella di 1HZ, la cui uscita di div funge da abilitazione per il contatore dei secondi. L'uscita di div del contatore dei secondi entra, poi, come abilitazione del contatore relativo ai minuti; entrambi i contatori sono definiti modulo 60. Leggermente più complessa è la connessione con il contatore relativo alle ore, il quale dovrà effettuare un conteggio quando sia il contatore dei secondi che quello dei minuti sono arrivati al termine. Come segnale di abilitazione, dunque, si utilizza l'uscita di una porta and, che presenta come ingresso entrambi i segnali di div dei contatori precedenti. Si presenta però un problema di temporificazione: il segnale di div del contatore relativo ai secondi, fungendo da abilitazione al contatore dei minuti e rimanendo alto per 1 solo colpo di clock, si alzerebbe appunto un colpo di clock prima di quello dei minuti; si è realizzato pertanto un blocco di delay, il quale ritarda tale segnale di 1 colpo di clock, in modo che la porta AND possa vedere entrambi i segnali alti. Se non avessimo usato tale blocco, il contatore delle ore non avrebbe mai visto il segnale di abilitazione alto.

```

architecture Structural of sistema_contatori is
component cont generic(N: positive); port (clock : in STD_LOGIC;
                                             reset : in STD_LOGIC;
                                             ab : in STD_LOGIC; -- abilitazione da contatori precedenti |
                                             set : in STD_LOGIC; -- set esterno valore iniziale
                                             input: in std_logic_vector ( 1 to integer( ceil( log2(real(N) ) ) ) );
                                             output : out STD_LOGIC_VECTOR (1 to integer( ceil( log2(real(N) ) ) ) );
                                             div : out STD_LOGIC); -- ab in uscita per prossimo contatore
end component;

component and_block port (a:in std_logic; b:in std_logic; o:out std_logic);
end component;
component wait_block port (clock: in std_logic; x:in std_logic; y:out std_logic);
end component;

constant frequency : integer := 100000000; -- 100 MHz=ls -- Per test 100Hz = 1000ns

FOR ALL: wait_block USE ENTITY work.wait_block( Behavioral);
FOR ALL: and_block USE ENTITY work.and_block(Dataflow);
FOR ALL: cont USE ENTITY work.contatore(Behavioral);

signal div_secondi,div_minuti, and_ore,div_secondi_delayed,ab_frequency : std_logic;

begin

wait_b: wait_block port map(clock=>clock,x=>div_secondi,y=>div_secondi_delayed);
and_b: and_block port map(a=>div_secondi_delayed,b=>div_minuti,o=>and_ore);
| div_frequenza: cont generic map (N=>frequency) port map (clock => clock, reset => reset, set => '0', ab=>'1',input => (others => '0'),
|                                         output => open,div => ab_frequency);
| cont_secondi: cont generic map( N=>60 ) port map (clock => clock, reset => reset, set => set, ab=>ab_frequency, input(1 to 6)=> input(1 to 6),
|                                         output(1 to 6) => output(1 to 6),div => div_secondi);
| cont_minuti: cont generic map( N=>60 ) port map (clock => clock, reset => reset, set => set, ab=>div_secondi, input(1 to 6) => input(7 to 12),
|                                         output(1 to 6) => output(7 to 12),div => div_minuti);
| cont_ore: cont generic map( N=>24 ) port map (clock => clock, reset => reset, set => set, ab=> and_ore, input(1 to 5) => input(13 to 17),
|                                         output(1 to 5) => output(13 to 17),div => open);

| end Structural;

```

Figure 40: Struct Sistema Contatori

## Component wait\_block:

E' stato progettato mediante un approccio comportamentale, con lo scopo di ritardare di 1 colpo di clock il segnale in ingresso.

```

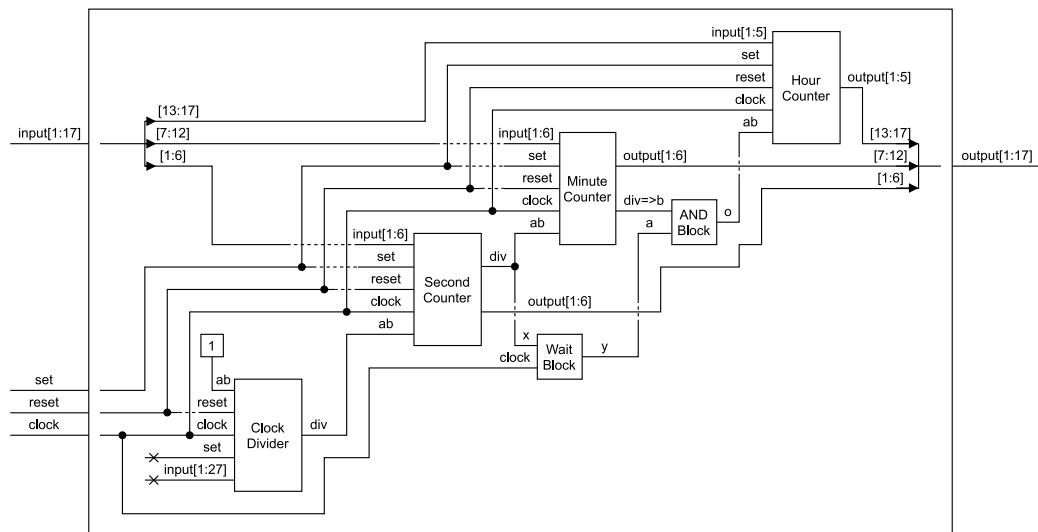
entity wait_block is
    Port ( clock: in STD_LOGIC;
            x : in STD_LOGIC;
            y : out STD_LOGIC);
end wait_block;

architecture Behavioral of wait_block is
begin
    process (clock)
    begin
        if ( clock = '1' and clock'event) then
            tmp<=x;
        end if;
    end process;
    y<=tmp;
end Behavioral;

```

Figure 41: Wait Block

## Schema sistema contatori



Definita la parte relativa al conteggio, possiamo ora vedere i componenti necessari al set di un valore iniziale dei contatori, in modo da far partire il cronometro da un orario scelto.

### Component gestore\_set:

```
entity gestore_set is
    Port ( clock, abModo : in STD_LOGIC;
            clr_button : in STD_LOGIC;
            dato : in STD_LOGIC_VECTOR (1 to 6);
            output : out STD_LOGIC_VECTOR (1 to 17):= (others=>'0');
            reset : in STD_LOGIC:= '0';
            set : out STD_LOGIC := '0';
            leds: out STD_LOGIC_VECTOR(2 downto 0));
end gestore_set;
```

Figure 42: Gestore set

Il segnale di set in ingresso ai contatori viene processato da un gestore di set, il quale ha il compito di mantenere quel segnale sempre alto finché non vengono settate ore, minuti e secondi. Il segnale di set viene prodotto dal gestore\_set in base ad un segnale fornito dall'esterno tramite un pulsante: la prima volta che si riceve tale segnale, il valore di set viene posto ad 1. A questo punto, tutti i contatori fermano il conteggio e rimangono in attesa di un vettore in ingresso e, tramite una variabile di conteggio chiamata count, si gestisce quale contatore inizializzare ad ogni passo. Il vettore in ingresso, infatti, contiene solo 6 bit e viene ogni volta aggiornato con il valore di ore, minuti e secondi che si vuole caricare; si è preferita una scelta di questo tipo rispetto ad un unico vettore contenente 17 ingressi dato che, quando si passerà alla sintesi su fpga, non saranno presenti 17 switch per gli input. Tornando alla variabile count, vediamo come il componente gestisce gli ingressi:

- Count=1 : i valori in ingresso vengono caricati nel counter dei secondi;
- Count=2 : i valori in ingresso vengono caricati nel counter dei minuti;
- Count=3 : i valori in ingresso vengono caricati nel counter delle ore;

Quando il count arriva a 3, inoltre, il suo valore viene portato a 0, indicando la fine dei dati in input da ricevere. Il valore di count viene incrementato ogni volta che arriva il segnale esterno che definisce il set; la quarta volta che il gestore riceve tale segnale, il valore di count sarà pari a 0 e, dunque, il segnale di set in uscita verrà abbassato e il cronometro riprende il suo normale funzionamento. È bene notare che, tutte le operazioni del gestore\_set vengono eseguite quando il segnale di ingresso ab\_modo è pari a 0; il motivo verrà spiegato nel seguito.

```

architecture Behavioral of gestore_set is

begin
    process ( clock)
        variable count: integer range 0 to 3:= 1;
    begin
        if(clock = '1' and clock'event) then
            if(abModo='0') then
                if(reset = '1') then
                    leds <= "000";
                    set <= '0';
                    output(l to 17) <= (others => '0');
                    count := 1;

                elsif(clr_button ='1') then
                    set <= '1';
                    if(count = 0) then
                        leds <= "000";
                        set <= '0';
                        count:= count +1;
                    elsif(count = 1) then
                        leds <= "001";
                        if(unsigned(dato) <= 59) then
                            output(l to 6) <= dato(l to 6);
                            output(7 to 17) <= (others => '0');

                        else
                            output <= (others => '0');
                        end if;
                        count := count+1;
                    elsif (count = 2) then
                        leds <= "011";
                        if(unsigned(dato) <= 59) then
                            output(7 to 12) <= dato(l to 6);
                            output(13 to 17) <= (others => '0');
                        else
                            output(7 to 17) <= (others => '0');
                        end if;
                        count := count+1;
                    elsif (count = 3) then
                        leds <= "111";
                        if(unsigned(dato) <= 23) then
                            output(13 to 17) <= dato(2 to 6);
                        else
                            output(13 to 17) <= (others => '0');
                        end if;
                        count := 0;
                    end if;
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

Figure 43: Architecture Gestore set

L'implementazione riportata è già adattata alla sintesi su fpga, introducendo dei valori di output relativi all'accensione di alcuni led, i quali indicano quale counter stiamo inizializzando.

## 5.2 Sintesi su fpga

Così come richiesto nell'esercizio 5.2, il componente cronometro realizzato in precedenza verrà sintetizzato su fpga. In particolare, verrà utilizzato il display a 7 segmenti presente sulla board, per la visualizzazione dell'orario, e 2 bottoni, per inviare i segnali di set e reset. Tuttavia, tale sintesi non può essere immediata, ma necessita di 2 componenti aggiuntivi:

- Debouncer: componente che serve a ripulire il segnale inviato dal bottone, che sarà sicuramente caratterizzato da rumore.
- Convertitore per display: l'uscita dei contatori deve essere tradotta in un formato leggibile dal display.

### **Componente debouncer:**

Tale componente ha il compito di trasformare un segnale rumoroso in un segnale pulito e la sua implementazione è analoga a quella riportata nell'esercizio 3.

### **Componente conv\_per\_display:**

Tale componente è stato progettato con un approccio dataflow: prende in ingresso i vettori in uscita dai contatori e traduce ognuno di essi in 2 vettori da 4 bit (richiesti dal display in ingresso), i quali rappresentano i valori di decine e unità.

```

) entity conv_per_display is
    Port ( input : in STD_LOGIC_VECTOR (1 to 17); -- secondi - minuti - ore
           output : out STD_LOGIC_VECTOR (1 to 32)); -- secondi - minuti - ore
) end conv_per_display;
architecture Dataflow of conv_per_display is
    signal ore,minuti,secondi : integer;

begin
    ore <=to_integer(unsigned(input(13 to 17)));
    minuti <= to_integer(unsigned(input(7 to 12)));
    secondi <= to_integer(unsigned(input(1 to 6)));

    output(29 to 32) <= std_logic_vector(to_unsigned(natural(secondi mod 10),4)); -- unita secondi
    output(25 to 28) <= std_logic_vector(to_unsigned(natural((secondi - secondi mod 10)/10),4)); -- decina secondi
    output(21 to 24) <= std_logic_vector(to_unsigned(natural(minuti mod 10),4)); -- unita minuti
    output(17 to 20) <= std_logic_vector(to_unsigned(natural((minuti - minuti mod 10)/10),4)); -- decina minuti
    output(13 to 16) <= std_logic_vector(to_unsigned(natural(ore mod 10),4)); -- unita ore
    output(9 to 12) <= std_logic_vector(to_unsigned(natural((ore - ore mod 10)/10),4)); -- decina ore
    output(1 to 8) <= (others => '0');

) end Dataflow;

```

Figure 44: Convertitore per display

Le uscite di questo componente andranno poi in ingresso al display\_manager.

### Estensione con intertempi:

Per far sì che il sistema sia in grado di memorizzare degli intertempi, sono stati aggiunti ulteriori componenti:

- Gestore modo: necessario per la visualizzazione degli intertempi salvati.
- Memoria: componente nel quale salvare gli intertempi.

### Componente Gestore\_modo:

```

entity gestore_modo is
    Port ( i_cronometro : in STD_LOGIC_VECTOR (1 to 32);
           i_intertempi : in STD_LOGIC_VECTOR (1 to 32);
           output : out STD_LOGIC_VECTOR (1 to 32);
           abModo:out std_logic;
           cleared_button : in STD_LOGIC;
           clk,reset: in STD_LOGIC);
end gestore_modo;

```

Figure 45: Entity Gestore modo

E' stato realizzato come una macchina a 2 stati

- Stato cronometro: a tale stato è associata l'uscita ab\_modo=0, la quale indica che l'uscita sul display è quella del sistema dei contatori, ovvero stiamo visualizzando il cronometro che scorre. Quando riceve in ingresso il segnale di cambio modo, inviato da un pulsante e gestito da un debouncer, si passa allo stato intertempo.
- Stato intertempo: a tale stato è associata l'uscita ab\_modo=1, la quale indica dunque che l'uscita sul display è quella della memoria, ovvero stiamo visualizzando gli intertempi salvati.

Per implementare la modalità intertempi, dunque, le uscite dei contatori e della memoria entrano nel gestore modo il quale, in base alla modalità di funzionamento del sistema, decide quale dei due ingressi riportare in uscita. Tale uscita andrà poi in ingresso al convertitore per display. L'uscita ab\_modo verrà inoltre riportata in ingresso alla memoria ed al gestore set, in quanto ne limita le funzionalità:

- Gestore set: quando siamo nella modalità intertempo, non è possibile settare un valore nei contatori.
- Memoria: quando si è nella modalità cronometro e si invia un segnale di write, allora l'intertempo viene scritto in memoria, altrimenti no. Inoltre, i dati possono essere letti dalla memoria e riportati sullo schermo, in corrispondenza di un segnale di read, solo quando il sistema è in modalità intertempo.

È bene notare che il segnale di ab\_modo non va in ingresso al sistema di contatori. Si è infatti supposto che, anche quando si è in modalità intertempo, il cronometro continua a lavorare, ma il tempo non viene mostrato sul display. Quando si ripassa alla modalità cronometro, infatti, si noterà che il valore di conteggio sarà avanzato.

```

architecture Behavioral of gestore_modo is
signal temp : std_logic_vector(1 to 32);
signal tempModo: std_logic;
begin
process(clk)
variable modo: bit :='0';
begin
    if(clk='1' and clk'event) then
        if( reset = '1') then
            temp<= (others => '0');
            tempModo <='0';
            modo := '0';
        elsif( modo ='1') then
            tempModo<= '1';
            temp <= i_intertempi;
            if( cleared_button ='1') then
                modo := '0';
            end if;
        else
            tempModo<='0';
            temp <= i_cronometro;
            if( cleared_button ='1') then
                modo := '1';
            end if;
        end if;
    end if;
end process;

output<= temp;
abModo <= tempModo;
end Behavioral;

```

Figure 46: Architecture Gestore modo

## Componente memoria:

```
) entity memoria is
    Generic( N:integer);
    Port ( input : in STD_LOGIC_VECTOR (1 to 32);
           output : out STD_LOGIC_VECTOR (1 to 32);
           reset : in STD_LOGIC;
           abModo: in STD_LOGIC; -- |(1 intertempo, 0 cronometro)
           clk : in STD_LOGIC;
           read,write : in STD_LOGIC);
) end memoria;
```

Figure 47: Entity Memoria

Tale componente è stato progettato tramite un approccio comportamentale e varia le sua funzionalità in base alla modalità di lavoro del sistema:

- Modalità intertempo: permette, in corrispondenza di un segnale di read, di scorrere gli intertempi salvati. Non si possono salvare nuovi intertempi in tale modalità.
- Modalità cronometro: permette, in corrispondenza di un segnale di write, di salvare l'intertempo in memoria. Non si possono visualizzare gli intertempi salvati in tale modalità.

Per come è stata progettata, se si salvano più di N intertempi, vengono sovrascritti i primi che sono stati salvati.

```

) architecture Behavioral of memoria is
type memoria_type is array (0 to N-1) of std_logic_vector(1 to 32);
signal mem: memoria_type;
begin
) memo: process(clk)
variable countInput,countOutput: integer range 0 to N-1:=0;
begin
    if(reset='1') then
        countInput:=0;
        countOutput:=0;
    for i in 0 to N-1 loop
        mem(i)<=(others => '0');
    end loop;
    output <= (others => '0');
    elsif(write='1' and abModo = '0') then
        mem(countInput)<= input;
        countInput:=(countInput+1) mod N;
    elsif(abModo = '1') then
        output <= mem(countOutput);
    if(read='1') then      --solo se sono in modalita intertempo devo scorrere i valori in memoria
        countOutput:=(countOutput +1) mod N;
    end if;
    else
        countOutput:=0;
    end if;
end if;
end process;

```

Figure 48: Architecture Memoria

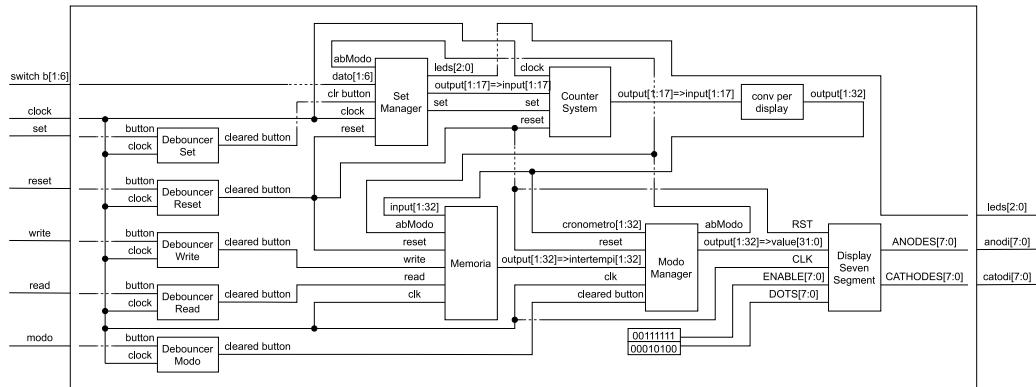


Figure 49: Schema Completo Esercizio 5

# 6 Esercizio 6 - Sistema di Testing

## 6.1 Traccia

**Esercizio 6.1:** Progettare, implementare in VHDL e verificare mediante simulazione un sistema in grado di testare in maniera automatica una macchina combinatoria M avente 4 ingressi e 3 uscite binarie sottponendole N ingressi diversi (si considerino una macchina M e un numero di input N a scelta dello studente). Gli N valori di input per il test devono essere letti da una ROM, in cui essi sono precaricati, in corrispondenza di un segnale read. Le N uscite fornite della macchina in corrispondenza di ciascuno degli input devono essere memorizzati in una memoria interna, che deve poter essere svuotata in qualsiasi momento in presenza di un segnale di reset.

**Esercizio 6.2:** Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando due bottoni per i segnali di read e reset rispettivamente e i led per la visualizzazione delle uscite della macchina istante per istante.

## 6.2 Implementazione

### Scelte progettuali:

L'esercizio richiedeva di progettare un sistema di testing per una macchina combinatoria, a partire da dei valori di input per la macchina precaricati in una ROM, si visualizza l'output della macchina combinatoria a tali valori tramite dei LED. Gli output vengono visualizzati a intervalli di 1s. Per poter iniziare il testing della macchina occorre un segnale di RESET, necessario per svuotare la memoria, ed un segnale di READ, per presentare in ingresso alla macchina gli input presenti nella ROM.

SISTEMA ESTERNO:



Il sistema è stato progettato mediante un approccio strutturale:

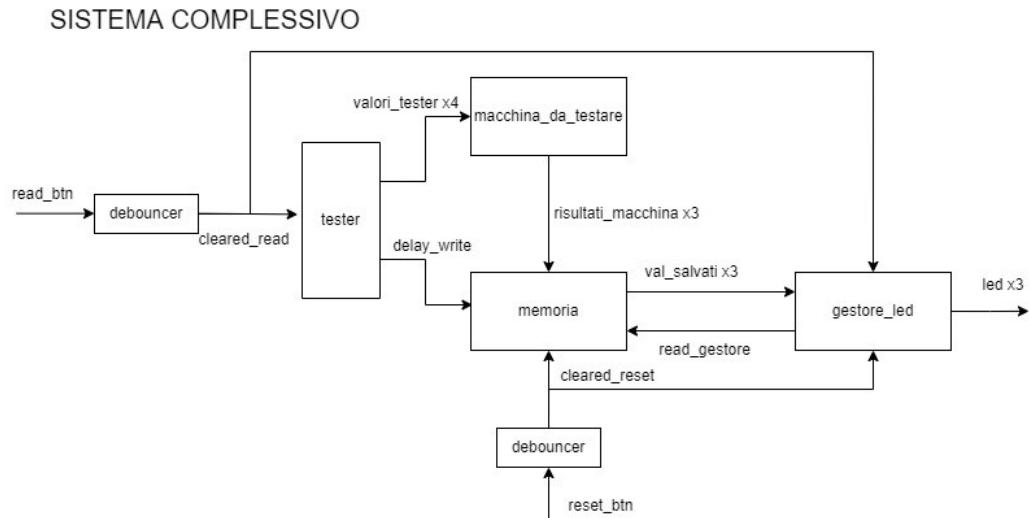


Figure 50: Sistema Complessivo

### Tester:

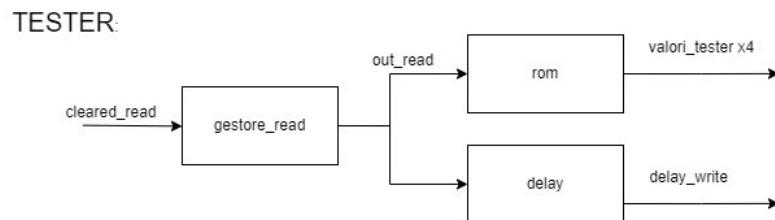


Figure 51: Tester

Il tester è anch'esso realizzato con approccio strutturale. Lo scopo di questo componente è quello di presentare in uscita dei valori da dare in input alla macchina che si vuole testare ed inviare un segnale di scrittura per il salvataggio dei risultati. Il sistema è composto da:

### Gestore read:

```

| entity gestore_read is
|   Generic (N: integer);
|   Port ( cleared_read : in STD_LOGIC;
|          out_read : out STD_LOGIC;
|          clk : in STD_LOGIC);
| end gestore_read;

begin
gest_read: process(clk)
variable stato: status :=NOTREAD;
variable count: integer:=0;
begin
  if(clk='1' and clk'event) then
    if count=N then
      stato := NOTREAD;
      count:=0;
    end if;
    if(stato = NOTREAD) then
      out_read <= '0';
    else
      count:= count+1;
      out_read <= '1';
    end if;
    if(cleared_read = '1') then
      if(stato = NOTREAD) then
        stato := READ;
      else
        stato := NOTREAD;
      end if;
    end if;
  end if;
end process;
end Behavioral;
```

Figure 52: Gestore Read

A partire dal segnale ripulito di READ viene prodotto un segnale di OUT\_READ che rimane alto per tanti periodi di clk quanti sono gli input di test:  $N \cdot T_{clk}$ . E' realizzato come una macchina a stati:

- Stato NOTREAD: in tale state il valore di out\_read è pari a 0 e rimane in attesa di un segnale cleared\_read, appena questo arriva si passa nello stato di READ.
- Stato READ: a tale stato è associata l'uscita out\_read pari a 1; il sistema rimane in questo stato per N colpi di clock per poi tornare allo stato NOTREAD.

## ROM:

```

entity ROM is
    Generic ( N: integer);
    Port ( output : out STD_LOGIC_VECTOR (0 to 3);
            read, clk: in STD_LOGIC );
end ROM;

architecture Behavioral of ROM is
type rom_type is array (0 to N-1) of std_logic_vector(0 to 3);
signal ROM: rom_type:="1100",
           "0111",
           "0001",
           "1110";
signal temp: std_logic_vector(0 to 3);
begin

mem: process(clk)
variable count: integer :=0;
begin
if(clk='0' and clk'event) then
    if(read ='1') then
        temp <= ROM(count);
        count:= (count+1) mod N;
    end if;
end if;

end process;
output <= temp;
end Behavioral;

```

Figure 53: ROM

E' stata realizzata come una memoria precaricata, che all'arrivo del segnale di read visualizza in uscita il valore puntato ed incrementa il puntatore ai valori memorizzati. Il generic permette di memorizzare i valori al variare della variabile N.

### DELAY:

Il blocco di delay è necessario per poter ritardare il segnale di write in memoria, in questo modo si sincronizza l'uscita della macchina combinatoria al segnale di scrittura. Si suppone che la macchina combinatoria abbia un ritardo inferiore al  $\frac{T_{clock}}{2}$ ; se così non fosse allora dovremmo aumentare il periodo del clock con un divisore in frequenza.

```
entity delay is
    Port ( clk : in STD_LOGIC;
           input : in STD_LOGIC;
           delay_input : out STD_LOGIC);
end delay;

architecture Behavioral of delay is

begin
    process(clk)
    begin
        if(clk='0' and clk'event) then
            delay_input <= input;
        end if;
    end process;
end Behavioral;
```

Figure 54: Delay Block

Il blocco di delay e la ROM agiscono sul fronte di discesa del clock in modo che il ritardo ottenuto sia tale da campionare il valore di uscita della macchina quando è stabile e non sulla transizione. Il segnale di write, infatti, varia sul fronte di discesa del clock, mentre la scrittura in memoria avviene sul fronte di salita.

Memoria: è realizzata in maniera comportamentale. La scrittura avviene con la delay\_write (che dura  $N \cdot T_{clk}$ ), quindi permette di scrivere N valori in uscita dalla macchina combinatoria. La lettura avviene con la read\_gestore (che dura  $T_{clk}$ ), la quale permette di scorrere i valori in memoria con periodo di 1s. Agiscono due puntatori: uno per la lettura ed uno per la scrittura.

```

entity memoria is
    Generic( N: integer);
    Port ( input : in STD_LOGIC_VECTOR (0 to 2);
            output : out STD_LOGIC_VECTOR (0 to 2);
            reset, read, write : in STD_LOGIC;
            clk : in STD_LOGIC);
end memoria;

architecture Behavioral of memoria is
type mem_type is array (0 to N-1) of std_logic_vector(0 to 2);
signal mem: mem_type;
signal temp: std_logic_vector (0 to 2) ;
begin
memoria: process(clk)
variable countInput,countOutput: integer range 0 to N-1:=0;
begin
    if(clk='1' and clk'event) then
        if(reset ='1') then
            countInput:=0;
            countOutput:=0;
            for i in 0 to N-1 loop
                mem(i)<=(others => '0');
            end loop;
            temp <= (others => '0');
        elsif(write='1') then
            mem(countInput) <= input;
            countInput:= (countInput+1) mod N;
        elsif(read='1') then
            temp <= mem(countOutput);
            countOutput:= (countOutput +1) mod N;
        end if;
    end if;
end process;
output <= temp;
end Behavioral;

```

---

Figure 55: Memoria Esterna

### Gestore Led:

```

entity gestore_led is
    Generic( N:positive; X:positive);
    Port ( clk,reset,cleared_read : in STD_LOGIC;
            read_mem : out STD_LOGIC;
            valori_salvati : in STD_LOGIC_VECTOR (0 to 2);
            led : out STD_LOGIC_VECTOR (0 to 2));
end gestore_led;

```

È realizzato come una macchina a stati che gestisce la visualizzazione degli output, visualizzandoli ad intervalli di 1s. Il generic N permette di gestire, tramite una variabile count, l'intervallo di visualizzazione degli output. Il generic X permette di stabilire, tramite la variabile count\_valori, il numero di valori da visualizzare.

```

architecture Behavioral of gestore_led is
type state is(q0,q_ready,q_ready2);
signal stato: state := q0;
signal temp: STD_LOGIC_VECTOR (0 to 2);
begin

process(clk)
variable count: integer := 0;
variable count_valori: integer := 0;
begin
    if( clk = '1' and clk'event) then
        if(reset = '1') then
            count := 0;
            count_valori := 0;
            led <= (others => '0');
            stato <= q0;
        end if;
        case stato is
            when q0 => count := 0;
            count_valori := 0;
            if(cleared_read = '1') then
                stato <= q_ready;
            else
                stato <= q0;
            end if;
            when q_ready => if(count_valori /= 0) then
                read_mem <= '1';
                end if;
                count_valori := count_valori + 1 mod X;
                stato <= q_ready2;
            when q_ready2 => read_mem <= '0';
                led <= valori_salvati;
                if(count < N-1) then
                    count := count + 1 mod N;
                    stato <= q_ready2;
                elsif(count_valori < X+1) then
                    count := 0;
                    stato <= q_ready;
                else
                    stato <= q0;
                end if;
            end case;
        . . .
end process;
end Behavioral;

```

Figure 56: Gestore Led

Lista di stati:

- Stato q0: è lo stato in cui si arriva dopo ogni segnale di RESET o dopo la visualizzazione completa dei valori in memoria. Vengono resettati i valori di count, count\_valori e si rimane in attesa di un segnale di READ per poter passare allo stato q\_ready.
- Stato q\_ready: si confronta il valore di count\_valori con 0 per poter stabilire se è la prima iterazione, in questo caso si aspetta che la macchina elabora i dati in ingresso per poi passare alla visualizzazione effettiva, altrimenti si alza il segnale di lettura della memoria. Si incrementa, poi, il valore di count\_valori e si passa allo stato q\_ready2.
- Stato q\_ready2: si abbassa il segnale di lettura della memoria e si mostrano i risultati, attendendo un numero di impulsi di clock pari ad N tramite la variabile count, prima di procedere alla visualizzazione del risultato successivo. Si effettua, poi, un controllo per stabilire se il valore visualizzato è l'ultimo o meno tramite count\_valori: se è così, allora si passa allo stato di q0; altrimenti si passa in q\_ready.

### 6.3 Simulazione e Sintesi

Simulazione:

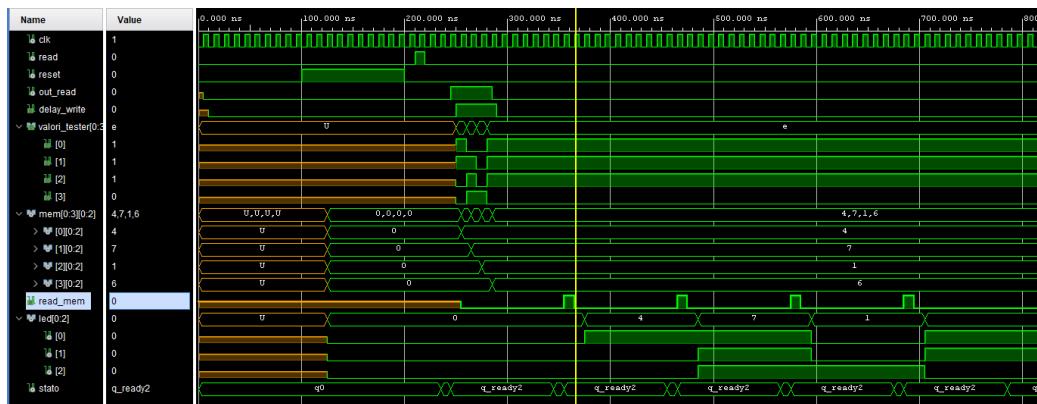


Figure 57: Simulazione Es.6

Sintesi su FPGA:

Per la sintesi su fpga sono stati inseriti i debouncer per i pulsanti di READ e RESET.

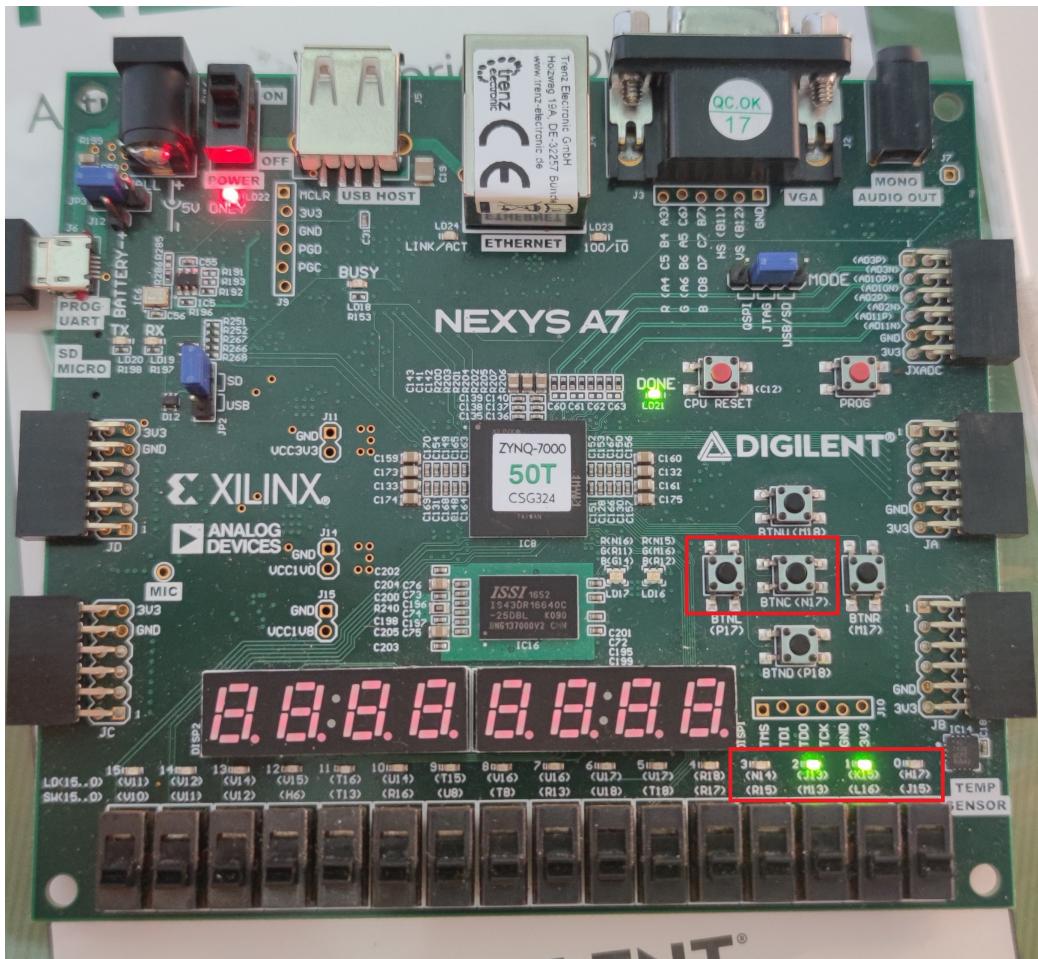


Figure 58: Sintesi Es.6

# 7 Esercizio 7 - Comunicazione Handshaking

## 7.1 Introduzione all'handshaking

Nella comunicazione tra due entità, generalmente, si individua un'entità che trasmette i dati, chiamata master, ed un'entità che riceve ed elabora i dati, chiamata slave. Per far interagire correttamente il master e lo slave bisogna prendere in considerazione 2 fattori: la presenza o meno di un criterio di asservimento e la definizione di un criterio di riferimento temporale. Quando l'entità slave è completamente asservita all'entità master, quest'ultima non deve preoccuparsi dell'istante di tempo in cui inviare i dati, dato che l'entità slave sarà sempre pronta a ricevere nuovi dati. In questo caso, quindi, non vi è la necessità di introdurre alcun segnale aggiuntivo per instaurare una comunicazione, ma basterà immettere i dati sul canale (al più si può pensare di utilizzare un segnale di strobe per evidenziare l'effettiva presenza dei dati). Tuttavia, non sempre tra 2 entità l'asservimento è assoluto, e dunque, bisogna assicurarsi che, quando l'entità master vuole iniziare una nuova trasmissione, l'entità slave sia a sua volta disponibile. Oltre ad un problema di asservimento, in genere, sussiste un problema di riferimento temporale, il quale nasce quando le 2 entità che devono interagire si trovano su 2 schede differenti, e dunque non possono essere fatte considerazioni sui riferimenti temporali. Infatti, anche se le 2 schede presentano clock isofrequenziali, non è detto che i riferimenti temporali siano analoghi, dato che questi possono variare per via alla fase iniziale. Al fine di risolvere queste problematiche e garantire una corretta comunicazione tra 2 entità, si utilizza un protocollo di comunicazione basato su handshaking, che nel senso letterale indica una “stretta di mano” prima di avviare la comunicazione, il quale si basa sull'ipotesi che sia sempre il master ad inviare dati sul canale.

Di un protocollo basato su handshaking vi sono più implementazioni, le quali differiscono per le modalità di inizio della comunicazione e, di conseguenza, per il numero di segnali trasmessi sul canale. Il protocollo più semplice è quello che prevede un segnale strobe da parte del master, che, come detto, evidenzia la presenza di dati sul canale, ed un segnale di risposta (generalmente chiamato di acknowledgment) da parte dello slave, il quale indica la corretta ricezione dei dati e può essere inviato sia appena ricevuti i dati, sia dopo la loro elaborazione. Una implementazione di questo tipo è utile per risolvere un problema di riferimento temporale, ma può non essere adatta a comunicazione prive di asservimento, per le quali generalmente si fa uso di

un protocollo basato su handshake interlacciato.

L'handshaking interlacciato prevede la verifica della disponibilità dello slave alla ricezione di nuovi dati, prima di immettere quest'ultimi sul canale. In tale implementazione, quando il master vuole inviare dei dati, prima invia un segnale di request allo slave, il quale, se disponibile, invia a sua volta un segnale di risposta. A questo punto parte la comunicazione. Un ulteriore protocollo di handshaking, definito semisincrono, viene utilizzato quando lo slave deve inviare un dato di risposta al master, e prevede un ulteriore segnale per indicare quando l'elaborazione è terminata ed il dato di risposta è effettivamente presente sul canale. Si parla di protocollo semi sincrono poiché, il master, vede il segnale solo in corrispondenza di un colpo di clock (un protocollo di questo tipo è generalmente implementato nella comunicazione tra il processore e la memoria cache).

Come detto all'inizio della trattazione, in genere il master e lo slave lavorano con 2 clock differenti: un protocollo di handshaking prevede che, la frequenza del riferimento temporale dello slave, sia N volte quella del master, in modo da riuscire a campionare il segnale ricevuto anche nel caso di sfasamenti del clock. Entrando ora nel merito dell'esercizio, si è deciso di implementare un protocollo con handshaking di tipo semisincrono.

## Approccio Utilizzato

L'esercizio in esame prevede 2 sistemi, A e B, dove il sistema A deve inviare N valori al sistema B il quale, a sua volta, dovrà sommare tali valori a quelli contenuti in memoria per poi salvare il risultato in ulteriori locazioni della memoria stessa. Entrambi i sistemi sono stati definiti mediante un approccio strutturale, dove l'elemento principale è l'interfaccia dei 2 sistemi, la quale è responsabile dell'implementazione del protocollo di handshaking. Prima di passare alla descrizione dei 2 sistemi, nel paragrafo successivo verrà descritto il protocollo di handshaking implementato.

## Descrizione Protocollo

Il protocollo di handshaking implementato prevede un segnale  $r$ , ovvero di request, alzato dal sistema A (il master), il quale indica la volontà di trasmettere dei dati, ed un segnale di ris, alzato dal sistema B (lo slave) in ricezione del segnale di request, che indica la disponibilità nel ricevere i dati e che rimarrà alto fino alla completa elaborazione di quest'ultimi. La variazione da 1 a 0 del segnale di ris indica, dunque, la completa elaborazione dei dati da parte dello slave e viene interpretata dal master come la possibilità di

inoltrare nuovi dati. L'handshake avviene prima della trasmissione di ogni dato. Vediamo ora come i 2 sistemi implementano tale protocollo.

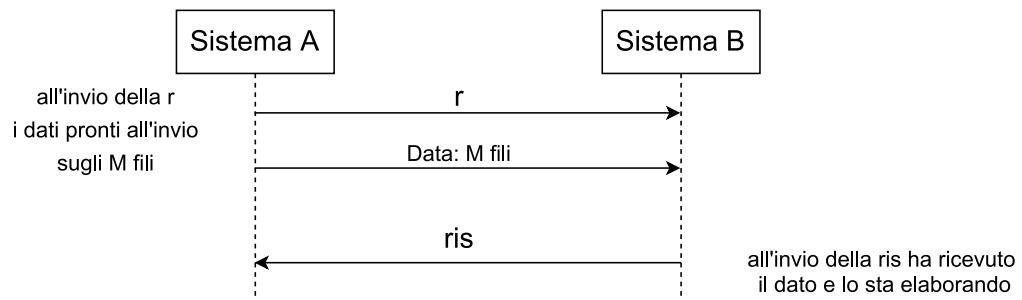
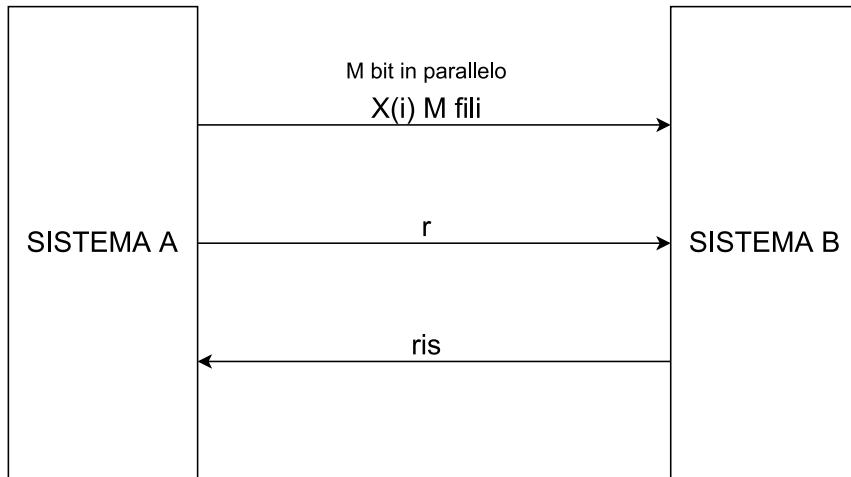


Figure 59: Schema Sistema Completo Es.7

## 7.2 Sistema A

Come precedentemente detto, il sistema A è stato definito tramite un approccio strutturale, utilizzando i seguenti componenti:

- Una memoria ROM dove vengono salvati i valori  $X(i)$ ;
- Un'interfaccia per effettuare l'handshaking e la trasmissione;
- Un contatore che segna il numero di comunicazioni già avvenute.

L'interfaccia rappresenta la parte di controllo mentre il contatore e la rom compongono la parte operativa.

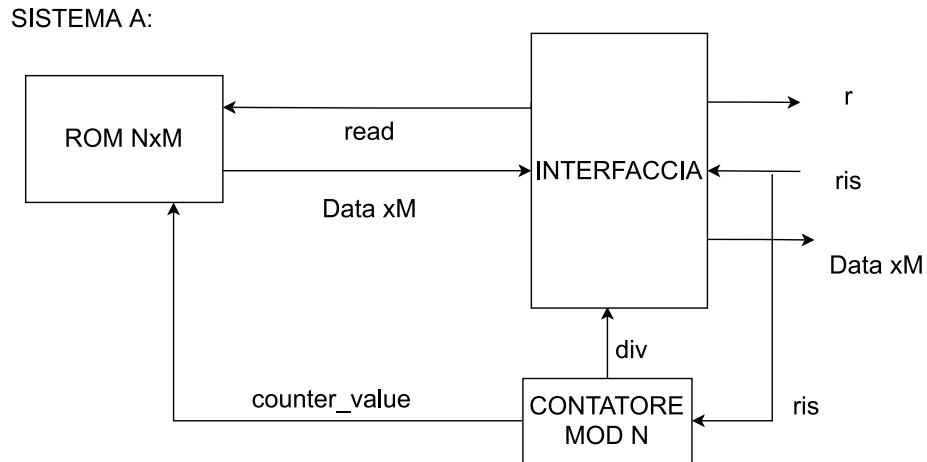


Figure 60: Schema Sistema A Es.7

Il sistema ripete la stessa sequenza di operazioni per ogni valore  $X(i)$  contenuto in ROM.

Alla comunicazione  $i$ -esima, l'interfaccia invia un segnale di read per leggere il valore  $X(i)$  puntato dal valore del contatore. Una volta letto il valore, l'interfaccia pone in uscita i dati per iniziare la nuova comunicazione e alza il segnale di R; abbassa tale segnale solo dopo l'arrivo del RIS. Finché RIS resta alto, il sistema A non può iniziare una nuova comunicazione, per questo rimane in attesa che il ricevitore finisca di elaborare i dati, cioè che RIS si abbassi. Appena RIS = 0 allora posso iniziare la nuova comunicazione  $i+1$ . Appena si raggiungono le  $N$  comunicazioni il segnale di DIV in uscita al contatore si alza e termina la comunicazione complessiva tra il sistema A e B dal lato del trasmettitore.

### 7.2.1 Unità Operativa A

#### Componente contatore:

```
entity contatore is
    Generic( N:integer);
    Port ( clk : in STD_LOGIC;
            ris : in std_logic;
            div : out STD_LOGIC:= '0';
            count: out std_logic_vector ( 0 to integer( ceil( log2(real(N) ) ) )-1 ) :=(others => '0'));
end contatore;
```

Figure 61: Entity Contatore

E' descritto come una macchina a stati che conta il numero di comunicazioni effettuate. A partire da un segnale di ris, il contatore conta quante volte questo si alza, ossia i fronti di salita:

1. q0 → stato in cui se il segnale in ingresso al contatore è basso allora resto in q0, altrimenti effettuo l'incremento del contatore in modulo N e mi sposto nello stato q1. Se raggiungo la fine del conteggio ossia count = N-1 allora alzo DIV (segnale di stop).
2. q1 → stato in cui aspetto che il segnale in ingresso si abbassi: se rimane alto, allora resto in q1, altrimenti passo in q0;

```
architecture Behavioral of contatore is
signal count_temp: std_logic_vector ( 1 to integer( ceil( log2(real(N) ) ) ) ) := (others => '0');
type state is (q0,q1);
signal stato_corrente: state:= q0;
begin
    process(clk)
    begin
        if( clk = '1' and clk'event) then
            case stato_corrente is
                when q0=> if(ris = '0') then stato_corrente <= q0;
                else stato_corrente <= q1;
                    if ( count_temp = N-1) then
                        div <= '1';
                        count_temp <= (others => '0');
                    else
                        div <= '0';
                        count_temp<=count_temp+1;
                    end if;
                end if;
                when q1=> if(ris = '1') then stato_corrente <= q1;
                else stato_corrente <= q0; end if;
                when others => stato_corrente <= q0;
            end case;
        end if;
    end process;
    count <= count_temp;
end Behavioral;
```

Figure 62: Architecture Contatore

## Componente Rom:

```
entity ROM is
    Generic ( N: integer; M: integer);
    Port ( output : out STD_LOGIC_VECTOR (0 to M-1):=(others => '0');
            read, clk: in STD_LOGIC ;
            count: in std_logic_vector ( 1 to integer( ceil( log2(real(N) ) ) ) );
    end ROM;
```

E' progettato come una semplice memoria, che sul fronte di salita del clock valuta il segnale di read: se questo è alto, pone in uscita il dato presente alla locazione indicata dal valore del counter.

```
architecture Behavioral of ROM is
type rom_type is array (0 to N-1) of std_logic_vector(0 to M-1);
signal ROM: rom_type:=( "0001",
                        "0010",
                        "0011",
                        "0100");
signal temp: std_logic_vector(0 to 3);
begin

mem: process(clk)
variable fine: integer:= integer(ceil(log2(real(N)) ));
begin
if(clk='1' and clk'event) then
    if(read ='1') then
        temp <= ROM(to_integer(unsigned(count(1 to fine))));
    end if;
end if;

end process;

output <= temp;

end Behavioral;
```

Figure 63: Architecture ROM

## 7.2.2 Unità di Controllo A

### Componente interfaccia:

```
entity interfacciaA is
    generic( M: positive);
    Port (
        clk : in STD_LOGIC;
        ris : in STD_LOGIC;
        div : in STD_LOGIC;
        datain : in STD_LOGIC_VECTOR(0 to M-1):=(others => '0');
        r : out STD_LOGIC;
        read : out STD_LOGIC:='0';
        dataout : out STD_LOGIC_VECTOR(0 to M-1):=(others => '0'));
end interfacciaA;
```

Figure 64: Entity Interfaccia A Es.7

L’interfaccia è il componente che gestisce l’handshaking e la trasmissione dei dati sul canale. È stata progettata come una macchina a 7 stati, dove:

1. q0 → è lo stato iniziale, dove ci si ritrova prima della trasmissione oppure al termine. Per distinguere i 2 casi si utilizza il segnale di div in uscita dal contatore: quando è basso, indica che ancora non abbiamo trasferito il primo dato (dato che differenziamo la prima trasmissione dalle altre e, dunque, in q0 si valuta solo la prima), mentre, quando è alto, indica che tutti i dati sono stati trasmessi e si permane indefinitamente in q0. Quando il segnale di div è basso, si passa il q1 e viene alzato il segnale di read, in modo da leggere dalla memoria il valore da trasmettere.
2. q1 → viene abbassato il segnale di read, dato che deve durare solo 1 colpo di clock, al fine di evitare letture spurie in memoria. Si passa poi nello stato q2. Anche se può sembrare superfluo, lo stato q1 è necessario, per dare il tempo al componente memoria di porre in uscita il dato i-esimo.
3. q2 → siamo ora sicuri che il dato in uscita dalla memoria è quello corretto: possiamo dunque riporlo sul canale ed alzare il segnale r. Si passa ora nello stato q3.

4.  $q_3 \rightarrow$  tale stato è sostanzialmente di attesa: finchè non si alza il segnale di ris in ingresso, si permane in  $q_3$ . Quando invece tale segnale viene rilevato come alto, i dati sono stati acquisiti dal sistema B e si può passare dunque allo stato  $q_4$ .
5.  $q_4 \rightarrow$  quando ci si ritrova in questo stato, si è sicuri che il sistema B ha ricevuto i dati inviati: si abbassa dunque il segnale di  $r$  e si passa allo stato  $q_5$ .
6.  $q_5 \rightarrow$  tale stato è sostanzialmente analogo a  $q_0$ , solo che viene utilizzato per le trasmissioni successive alla prima. Si valuta quindi il segnale di  $div\ e$ , se alto, si procede verso  $q_0$ , altrimenti si alza il segnale di  $read$  e si procede verso  $q_6$ . È bene notare che, tali controlli sul segnale di  $div$ , potevano essere fatti anche in  $q_4$ ; tuttavia, dato che il contatore si incrementa sul segnale di ris in ingresso al sistema, si è preferito aggiungere uno stato per essere sicuri di vedere tale segnale stabile.
7.  $q_6 \rightarrow$  se ci ritroviamo in questo stato, vuol dire che si deve procedere ad una trasmissione successiva alla prima. Per questo motivo si controlla il segnale di ris: se questo è basso, allora la precedente elaborazione è terminata e si può procedere ad inviare i nuovi dati, passando allo stato  $q_2$ , mentre, se alto, si permane in  $q_6$  in attesa che si abbassi.

```

architecture Behavioral of interfacciaA is
type state is (q0,q1,q2,q3,q4,q5,q6);
signal stato_corrente,stato_prossimo: state:= q0;
signal r_temp : std_logic := '0';
signal data_temp: STD_LOGIC_VECTOR(0 to M-1):=(others => '0');
begin
process(clk)
begin
if(clk = '1' and clk'event) then
    case stato_corrente is
        when q0 => if(div = '1') then
            stato_corrente <= q0;
        else
            stato_corrente <= q1;
            read <= '1';
        end if;
        when q1 => read <= '0';
            --data_temp <= datain;
            stato_corrente <= q2;
            --read <= '1';
        when q2=> data_temp <= datain;
            r_temp <= '1';
            stato_corrente <= q3;
        when q3=> if(ris = '0') then
            stato_corrente <= q3;
        else
            stato_corrente <= q4;
        end if;
        when q4 => r_temp <= '0';
            stato_corrente <= q5;
        when q5 => if(div = '1') then
            stato_corrente <= q0;
        else
            read <= '1';
            stato_corrente <= q6;
        end if;
        when q6 => read <= '0';
            --data_temp<= datain;
            if(ris = '0') then
                stato_corrente <= q2;
            else
                stato_corrente <= q6;
            end if;
        when others => stato_corrente <= q0;
    end case;
end if;
end process;

r <= r_temp;
dataout <= data_temp;

end Behavioral;

```

Figure 65: Architecture Interfaccia A Es.7

### 7.3 Sistema B

Il Sistema B è stato definito tramite un approccio strutturale, utilizzando i seguenti componenti:

- Un'interfaccia, la quale comunica con il sistema A, gestisce i segnali di r e ris e acquisisce i dati da elaborare;
- Un contatore per indicare la terminazione dei dati da ricevere;
- Una control unit, che ha il compito di effettuare le operazioni sui dati ricevuti e di salvarli in memoria;
- Una memoria unica con i dati da sommare a quelli ricevuti dal sistema A e ulteriori locazioni per salvare i risultati.

Anche per tale sistema, l'interfaccia rappresenta la parte di controllo mentre la memoria, il contatore e la control unit compongono la parte operativa. Il nome control-unit può risultare confusionale, ma è stato dato poiché elabora i dati e controlla il componente memoria.

SISTEMA B

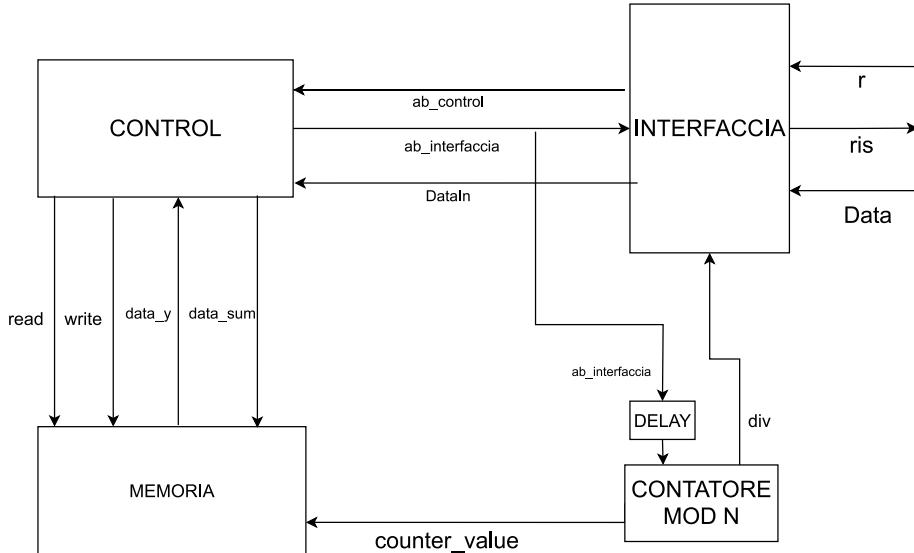


Figure 66: Schema Sistema B Es.7

### 7.3.1 Unita Operativa B

#### Componente Control Unit:

```

)entity control_unit is
  Generic(M:positive);
  Port ( ab_control : in STD_LOGIC := '0';
         ab_interfaccia : out STD_LOGIC := '0';
         DataIn : in std_logic_vector(0 to M-1):=(others=> '0');
         read : out STD_LOGIC:='0';
         write : out STD_LOGIC:='0';
         Data_y : in std_logic_vector(0 to M-1):=(others=> '0');
         Data_sum : out std_logic_vector(0 to M-1);
         clk : in STD_LOGIC);
)end control_unit;

```

Figure 67: Entity Control Unit

La control unit è responsabile dell'elaborazione dei dati ricevuti e della collocazione dei risultati all'interno della memoria. Anch'essa è stata definita come una macchina a stati, dove:

1. q0 → è lo stato in cui la macchina permane fino alla ricezione del primo dato da elaborare, rimanendo in attesa della variazione da 0 a 1 del segnale di ab\_control modificato dall'interfaccia. Quando rileva tale variazione passa allo stato q1 ed abbassa l'ab\_interfaccia.
2. q1 → in tale stato si ripone il segnale di read a 1 e si passa allo stato q2.
3. q2 → dato che il segnale di read si vuole che duri un singolo colpo di clock, in questo stato viene abbassato e si passa poi allo stato q3.
4. q3 → in tale stato si è sicuri che i dati da elaborare sono entrambi disponibili e corretti, motivo per cui vengono immagazzinati come unsigned all'interno di appositi signal, tramite i quali si eseguirà poi una operazione di somma. In questo stato si pone a 1 il segnale di ab\_interfaccia, il quale verrà letto al colpo di clock dopo dall'interfaccia che abbasserà il segnale di ris. Anche se l'elaborazione non è del tutto terminata, si inizia a dare tale segnale dato che al colpo di clock dopo si avrà effettivamente la completa elaborazione e, comunque, il segnale ab\_interfaccia verrà rilevato al colpo di clock successivo. Si passa infine allo stato q4.
5. q4 → viene alzata la write e posto sul segnale s il valore della somma, il quale verrà salvato nella memoria; si pone inoltre a 0 il segnale di ab\_interfaccia. Quest'ultima operazione può risultare ambigua, dato

che porre a 0 tale segnale vuol dire disabilitare l'interfaccia, che in questo caso però deve essere attiva e pronta a ricevere ed inoltrare nuovi dati. Ciò che è bene notare, però, è che l'interfaccia valuta il segnale di ab\_interfaccia solo in alcuni stati e, la variazione di tale segnale permette comunque di proseguire verso alcuni stati, ma la mette in attesa negli stati in cui è giusto che l'interfaccia si fermi una volta ricevuti i dati. Stiamo sostanzialmente dicendo all'interfaccia che, quando arriverà nuovamente in uno di quegli stati dovrà fermarsi e non che in generale non deve lavorare. Si passa infine allo stato q5.

6. q5 → tale stato ha sostanzialmente lo stesso scopo dello stato q4 nell'interfaccia, ovvero quello di replicare lo stato q0 quando però non si è al primo dato da elaborare. Le operazioni sono dunque analoghe a quelle dello stato q0, con l'aggiunta della variazione da 1 a 0 del segnale di write, il quale, come il segnale di read, deve durare 1 colpo di clock.

---

```

architecture Behavioral of control_unit is
type state is (q0,q1,q2,q3,q4,q5);
signal stato_corrente: state:= q0;
signal a,b: unsigned(0 to M-1):=(others =>'0');
signal temp_read,temp_write: std_logic:= '0';
signal sum_temp: unsigned(0 to M-1):= (others =>'0');
begin

process(clk)
begin
if(clk='1' and clk'event) then
    case stato_corrente is
        when q0 => if(ab_control ='0') then
                    stato_corrente <= q0;
            else
                    stato_corrente <= q1;
                    ab_interfaccia<='0';
            end if;

        when q1 => temp_read <= '1';
                    stato_corrente <= q2;

        when q2 => temp_read <= '0';
                    stato_corrente <= q3;

when q3 => a <= unsigned(DataIn);
            b <= unsigned(Data_y);
            stato_corrente <= q4;
            ab_interfaccia<='1';
|  

when q4 => sum_temp <= (a + b);
            temp_write<='1';
            ab_interfaccia<='0';
            stato_corrente <= q5;

when q5 => temp_write<='0';
            if(ab_control ='0') then
                stato_corrente <= q5;
            else
                stato_corrente <= q1;
            end if;

when others => stato_corrente <= q0;
    end case;
end if;
end process;
read <= temp_read;
write <= temp_write;
Data_sum <= std_logic_vector(sum_temp(0 to M-1));
end Behavioral;

```

Figure 68: Entity Control Unit

## Componente Counter:

Tale contatore è impostato sul fronte di salita del clock e incrementa solo quando il segnale di ab\_interfaccia si alza, ovvero quando è terminata l'elaborazione del dato. Siccome tale segnale si alza un colpo di clock prima dell'effettiva terminazione, è stato introdotto un blocco di delay per salvare i valori nella giusta locazione di memoria.

Tale componente è analogo al contatore utilizzato nel sistema A, ovvero progettato come una macchina a stati; varia solamente il segnale che riceve in ingresso.

## Componente Memoria:

```
entity MemoriaB is
  Generic(N:positive; M:positive);--N è il numero di locazioni delle Y, la memoria totale sarà quindi 2N
  Port (clk: in std_logic;
        read, write: in std_logic:='0';
        ready: out std_logic;
        Data_read: out std_logic_vector(0 to M-1):=(others=> '0');
        Data_write: in std_logic_vector(0 to M-1):=(others =>'0');
        counter_value: in std_logic_vector ( 1 to integer( ceil( log2(real(N) ) ) ) );
  end MemoriaB;
```

Tale componente è stato descritto in modo comportamentale ed è uguale al componente utilizzato nell'esercizio 5. Tuttavia, per completezza, riportiamo nuovamente la sua implementazione.

```
architecture Behavioral of MemoriaB is
  type memoria is array (0 to 2*N-1) of std_logic_vector(0 to M-1);
  signal mem: memoria :=("0001",
                         "0010",
                         "0011",
                         "0100",
                         "0000",
                         "0000",
                         "0000",
                         "0000");
  signal temp_Data_read:std_logic_vector(0 to M-1):=(others=> '0');
begin
  process(clk)
    variable fine: integer:= integer(ceil(log2(real(N))));
  begin
    if(clk='1' and clk'event) then
      if(read='1') then
        temp_Data_read<= mem(to_integer(unsigned(counter_value(1 to fine))));
      elsif(write ='1') then
        mem(to_integer(unsigned(counter_value(1 to fine)))+N)<= Data_write;
      end if;
    end if;
  end process;
  Data_read <= temp_Data_read;
end Behavioral;
```

Figure 69: Architecture Memoria B

### 7.3.2 C B

**Componente Interfaccia:** L'interfaccia è stata definita come una

```
entity interfacciaB is
    Generic( M: positive);
    Port ( clk : in STD_LOGIC;
            r : in STD_LOGIC;
            ris : out STD_LOGIC;
            DataIn : in STD_LOGIC_VECTOR (0 to M-1) := (others =>'0');
            DataOut : out STD_LOGIC_VECTOR (0 to M-1) := (others =>'0');
            div : in STD_LOGIC;
            ab_control : out STD_LOGIC := '0';
            ab_interfaccia : in STD_LOGIC := '0');
end interfacciaB;
```

Figure 70: Interfaccia B

macchina a 5 stati, dove:

1. q0 → E' lo stato in cui si trova l'interfaccia prima di ricevere il primo dato da elaborare. In questo stato, si attende la variazione del dato in input r, ricevuto dal sistema A, la cui variazione da 0 a 1 indica la disponibilità dei dati da elaborare. Finché r è basso, l'interfaccia permane in q0 e, quando rileva la variazione sul fronte di salita del clock, acquisisce i dati, alza il segnale di ris in uscita ( il quale viene visto dal sistema A ed è indice che i dati sono stati prelevati e che l'elaborazione di B è iniziata) ed abilita il segnale di ab\_control, che mette in attesa l'interfaccia stessa durante l'elaborazione dei dati da parte della control unit. Passa infine nello stato q1.
2. q1 → In tale stato l'interfaccia permane finché non vede una variazione da 0 a 1 del segnale ab\_interfaccia, il quale viene fatto variare dalla control unit una volta terminata l'elaborazione dei dati ricevuti. Quando viene rilevata la variazione su tale segnale, l'interfaccia abbassa il segnale ab\_control, disabilitando così la control unit fino alla ricezione di nuovi dati. Passa infine nello stato q2.
3. q2 → La prima operazione che si effettua in questo stato è la valutazione del segnale r, verificando se questo è ancora alto o meno. Tale operazione potrebbe risultare inutile, dato che il sistema A abbassa

tale segnale nel momento in cui vede il segnale di ris alzarsi, ma così non è. Bisogna ricordare infatti che il sistema B lavora in genere con un clock a frequenza maggiore (o semplicemente con fase diversa) e potrebbe quindi svolgere le operazioni in maniera molto rapida e non dare il tempo al sistema A di abbassare tale segnale. Se non venisse effettuato tale controllo, negli stati successivi si potrebbe interpretare il segnale di r pari ad 1 come indice di nuova trasmissione, portando alla rielaborazione di dati già processati.

L'interfaccia pertanto rimane in tale stato finché non rileva una variazione da 1 a 0 del segnale di r; quando tale segnale varia, abbassa il segnale di ris e passa nello stato q3.

Valutare il segnale r prima di abbassare quello di ris risolve inoltre un altro problema: dato che il segnale di ris viene valutato dal sistema A, così facendo, si fa in modo che duri almeno 1 colpo di clock di A, in modo che quest'ultimo possa rilevarne le variazioni, indipendentemente dal numero di operazioni effettuate dal sistema B.

4. q3 → Tale stato è il responsabile del controllo dei dati rimanenti da elaborare. Se il segnale di div in uscita dal contatore è alto, allora vuol dire che il dato appena elaborato era l'ultimo e l'interfaccia torna dunque nello stato q0. In caso contrario si procede verso lo stato q4.
5. q4 → Questo stato è sostanzialmente l'analogo dello stato q0, ma quando si sono già ricevuti dei dati. È necessario introdurre questo tipo di stato per effettuare un controllo sul numero di elaborazioni effettuate per determinare la terminazione di esse.

```

architecture Behavioral of interfacciaB is
type state is (q0,q1,q2,q3,q4);
signal stato_corrente: state:= q0;
signal ris_temp,temp_ab_control : std_logic := '0';
signal temp_dataIn:STD_LOGIC_VECTOR (0 to M-1):= (others =>'0');
begin
begin
process(clk)
begin
if(clk='1' and clk'event) then
case stato_corrente is
when q0 => if(r='0') then
stato_corrente <= q0;
else
stato_corrente <= q1;
temp_ab_control <= '1';
ris_temp<= '1';
temp_dataIn <= DataIn;
end if;
when q1 => if(ab_interfaccia='0') then
stato_corrente <= q1;
else
temp_ab_control <= '0';
stato_corrente <= q2;
end if;

when q2 => if (r='0') then
ris_temp <= '0';
stato_corrente <= q3;
else
stato_corrente <= q2;
end if;
when q3 => if(div='0') then
stato_corrente <= q4;
else
stato_corrente <= q0;
end if;
when q4 => if(r='0') then
stato_corrente <= q4;
else
stato_corrente <= q1;
temp_ab_control<= '1';
ris_temp<= '1';
end if;
when others => stato_corrente <= q0;
end case;
end if;
end process;

ab_control <= temp_ab_control;
DataOut <= DataIn;
ris <= ris_temp;
end Behavioral;

```

Figure 71: Architecture Interfaccia B Es.7

## 7.4 Considerazioni Finali

Per come è stato implementato, il protocollo è efficiente con  $f_B = N \cdot f_A$  per ogni valore di N (si da per scontato che il clock di B permette l'assestamento dei segnali in 1 colpo di clock).

In questo modo, non serve che il componente master ed il componente slave si accordino sulle frequenze da utilizzare.

Nel caso in cui siano noti i riferimenti temporali dei 2 componenti, possono essere ridotti il numero di stati delle interfacce di A e di B.

### Simulazione:

Così come richiesto, è stata effettuata una simulazione del funzionamento del protocollo, con una frequenza  $f_B = 2 \cdot f_A$  :

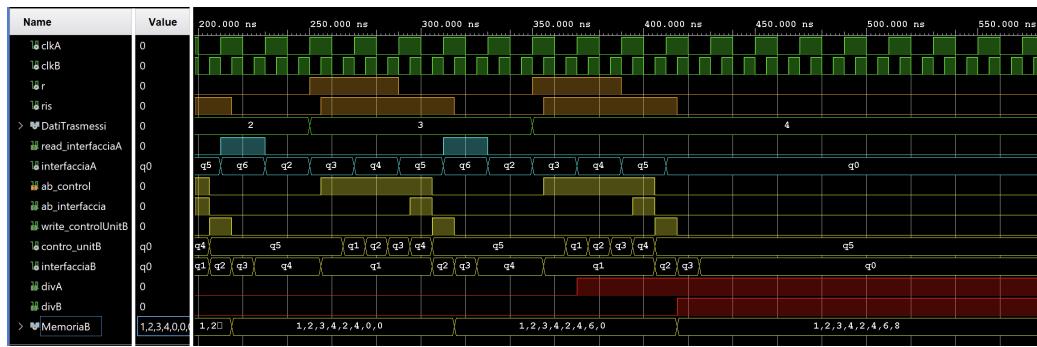
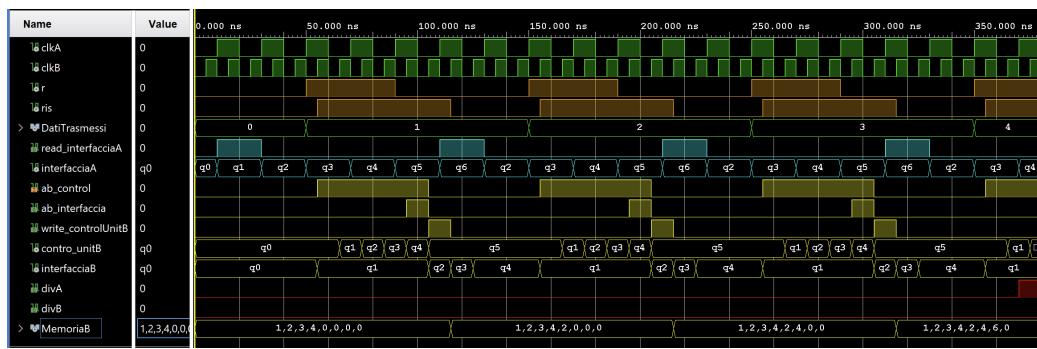


Figure 72: Simulazione Es.7

## 8 Esercizio 8 - Processor

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

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

### 8.1 Introduzione a IJVM

E' un linguaggio di livello macchina, semplificazione dell'Instruction Set Architecture JVM. La semplificazione consiste nell'utilizzo di sole operazioni su interi. L'IJVM organizza la memoria suddividendola in:

- Constant Pool: area che non permette la scrittura da parte del programma IJVM. I dati qui caricati (costanti, puntatori e stringhe), possono essere scritti solamente quando il programma è portato in memoria ed è accessibile tramite il registro Constant Pool Pointer;
- Local Variable Frame: area dove vengono inserite le variabili locali al programma ed i parametri relativi ad esso. L'indirizzo relativo all'inizio di quest'area è gestito tramite il registro LV;
- Stack degli operandi: è localizzato al di sopra del Local Variable Frame e permette di gestire gli operandi durante un'operazione. È accessibile tramite il registro SP, il quale punta sempre all'ultima locazione inserita nello stack;
- Method Area: area in cui risiedono i programmi IJVM da eseguire. Quest'area della memoria viene gestita come un array di byte, mentre tutte le precedenti sono gestite come array di parole da 4 byte.

Per i registri LV,SP e CPP, indicare un offset equivale quindi ad indicare uno spiazzamento di 32 bit, mentre per il PC equivale ad indicare uno spiazzamento di 8 bit.

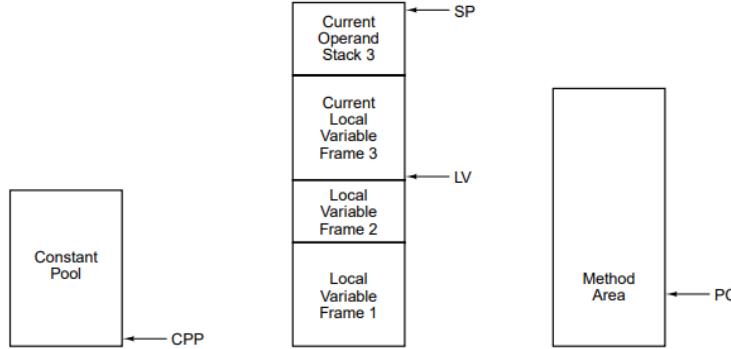


Figure 73: Memoria IJVM

Le istruzioni IJVM sono a lunghezza variabile in base alla presenza di operandi o altri parametri; in generale possono occupare da 8 bit a 24 bit e vengono dunque memorizzate nella Method Area in locazioni contigue. Il primo byte di ogni istruzione è il Codice Operativo, ossia un indirizzo che verrà utilizzato per mappare l'istruzione IJVM con l'indirizzo start del microprogramma contenuto nella control\_store del processore.

### Esempi di istruzioni IJVM:

Hex	Mnemonic	Meaning
0x10	BIPUSH <i>byte</i>	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO <i>offset</i>	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ <i>offset</i>	Pop word from stack and branch if it is zero
0x9B	IFLT <i>offset</i>	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ <i>offset</i>	Pop two words from stack; branch if equal
0x84	IINC <i>varnum const</i>	Add a constant to a local variable
0x15	ILOAD <i>varnum</i>	Push local variable onto stack
0xB6	INVOKEVIRTUAL <i>disp</i>	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE <i>varnum</i>	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W <i>index</i>	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Figure 74: Istruzioni IJVM

## Il processore MIC-1:

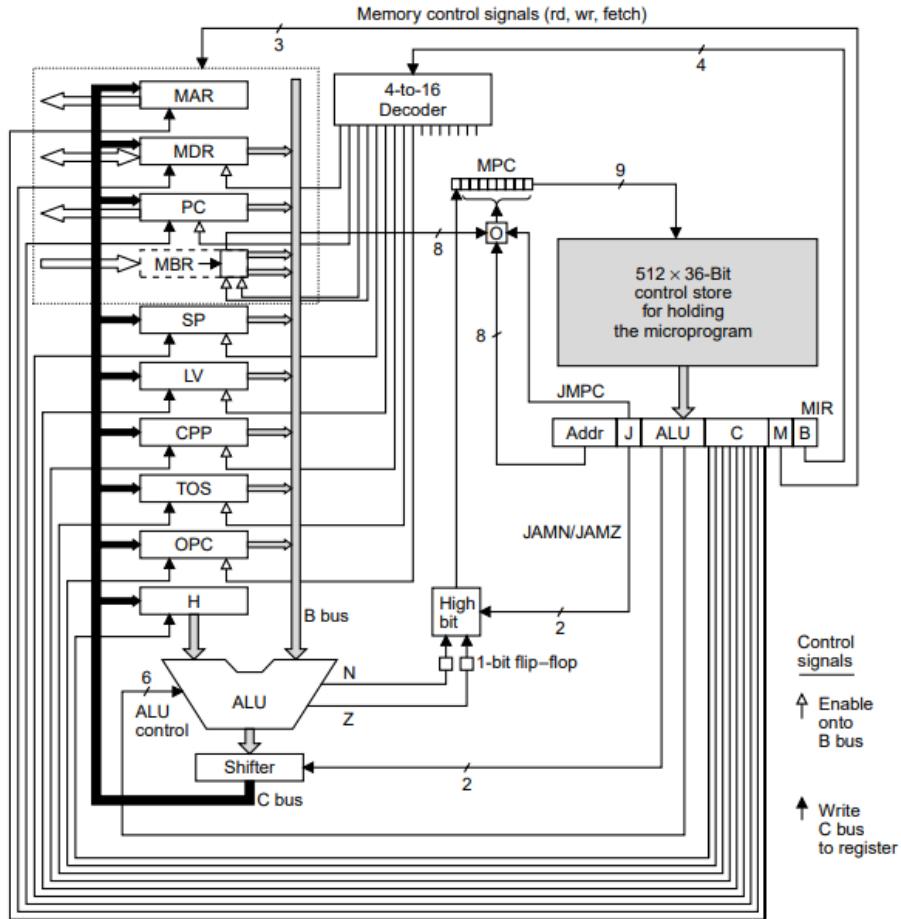


Figure 75: Processore MIC-1

E' una macchina a stack, ossia che non utilizza registri general purpose (ad esempio di tipo indirizzo o dato), ma le istruzioni presentano operandi contenuti in uno stack allocato nella memoria principale. Il processore MIC-1 è organizzato in parte operativa e parte di controllo.

## 8.2 Unità Operativa

La parte operativa realizza il datapath dell'architettura ed è composto da registri a 32 bit, 3 bus, ALU e shifter, come mostrato in figura:

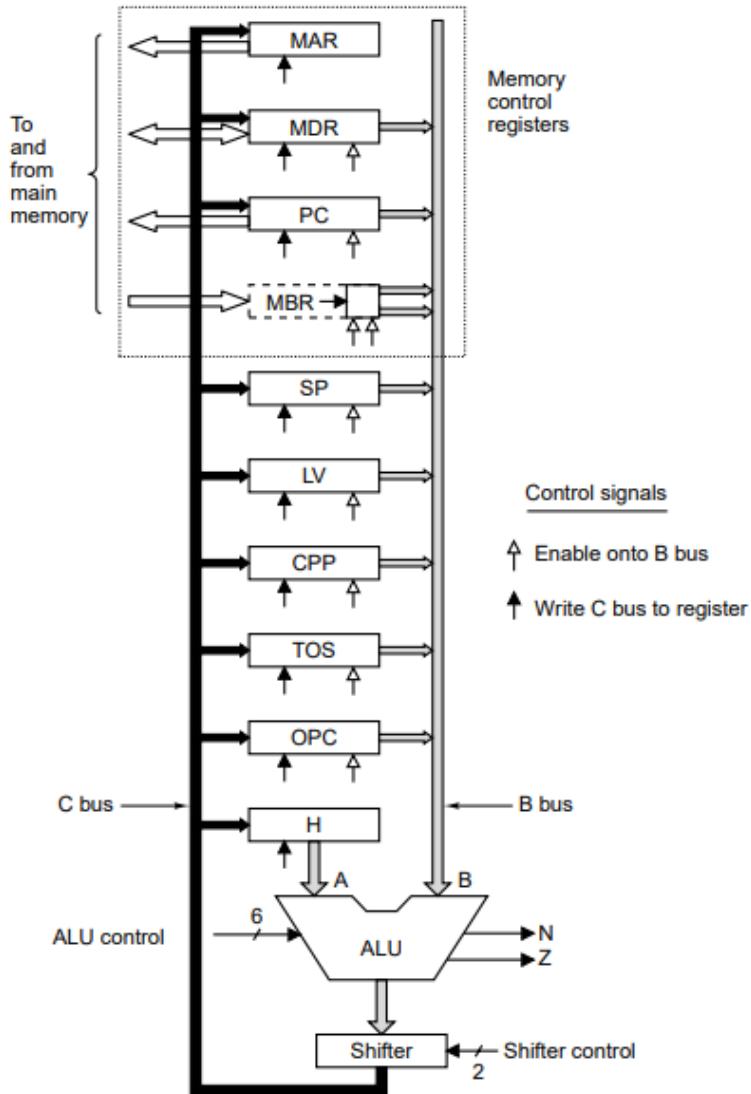


Figure 76: Datapath Processore

## Comunicazione in memoria:

La comunicazione in memoria differisce in tempificazione se si tratta di una read o di una write. In particolare nel caso della write, nello stesso periodo di clock avviene sia l'acquisizione sulle linee dell'indirizzo che del dato da scrivere in memoria, mentre nel caso della read è necessario un colpo di clk in più per la presentazione del dato in uscita alla memoria.

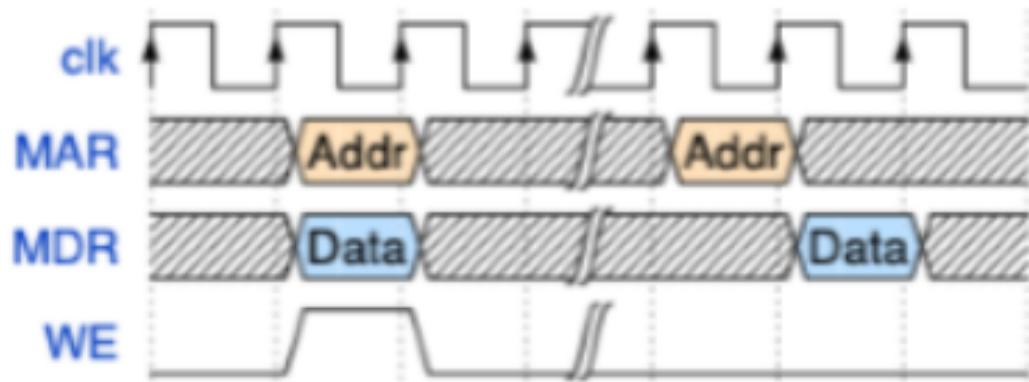


Figure 77: Tempificazione Memoria

I registri utilizzati per la comunicazione in memoria si suddividono in coppie:

- **MAR-MDR**: Quest'interfaccia è relativa alle aree di memoria accessibili in parole di 4 byte (Constant Pool, Local Variable, Stack). Questa consente di specificare l'indirizzo in memoria in MAR a partire dal quale leggere o scrivere 4 byte consecutivi nel MDR.
- **PC-MBR**: Quest'interfaccia è relativa all'area di memoria in cui è contenuto il programma, la quale è accessibile in parole da 1 byte (Method Area). Questa consente di specificare l'indirizzo in memoria in PC a partire dal quale leggere 1 byte ponendolo nel MBR. Ciò significa che se un'istruzione prevede di specificare sia il codice operativo, che un operando, allora verrà prelevato prima il codice operativo e poi l'operando; le microprocedure sono realizzate tenendo conto proprio di questa modalità di comunicazione della memoria dell'interfaccia PC-MBR.

## **Bus:**

L'unità operativa dispone di tre bus con parallelismo a 32 bit:

- A: per utilizzare un secondo operando ed effettuare le operazioni in ALU;
- B: è utilizzato per effettuare la lettura dai registri;
- C: utilizzato per la scrittura dei registri;

## **ALU:**

L'ALU ha due ingressi: A, collegato direttamente al registro tampone H, e B collegato al bus B. Per poter definire l'operazione da effettuare a partire dagli operandi si utilizza una stringa di segnali di controllo di 6 bit. Inoltre, quando il risultato passa nello shifter, vi sono ulteriori 2 segnali che permettono di effettuare lo shift di quest'ultimo, prima che venga posto sul bus C.

### 8.3 Unità di Controllo

E' realizzata tramite una memoria di sola lettura, detta control store, un microPC ed un microIR. La control store memorizza tutti i microprogrammi relativi alle istruzioni del set IJVM, allocandoli in memoria all'indirizzo corrispondente al codice operativo dell'istruzione. Ad ogni ciclo di clk, viene continuamente letta dalla control store la microistruzione all'indirizzo puntato dal mPC.

Il formato delle microistruzioni è dato dalla seguente figura:

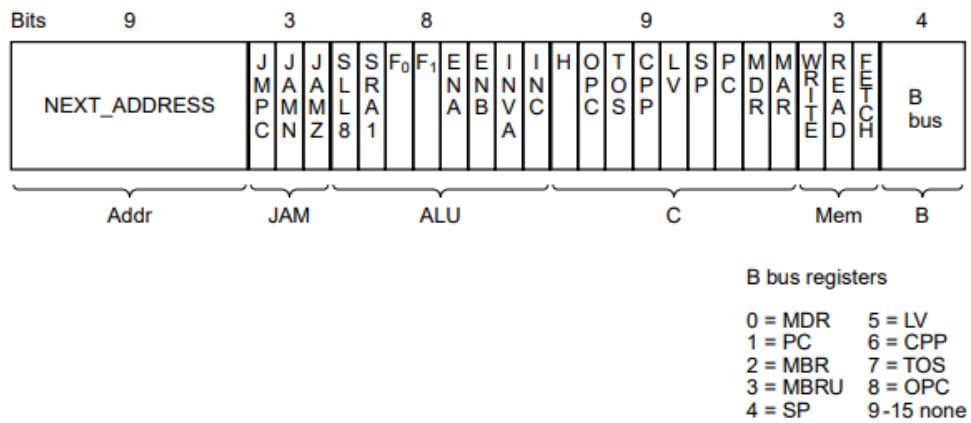


Figure 78: Formato Microistruzione

Ogni microistruzione è legata alla successiva tramite i bit di NEXT\_ADDRESS e i bit di salto JAM. È necessario effettuare questo concatenamento delle microistruzioni poiché, quelle relative allo stesso microprogramma, potrebbero non essere salvate in locazioni contigue della control\_store (si pensi a POP e DUP). Oltre a questo concatenamento è possibile che ci sia la presenza dei salti in base al tipo di microistruzione (ad esempio quando vengono utilizzati i costrutti if, else e goto).

In particolare, se JAM = "000", allora mPC = NEXT\_ADDRESS, altrimenti, se almeno uno dei flag di salto è alto, occorre calcolare il nuovo mPC:

- JAMN = 1: il bit più significativo di mPC viene messo in OR con il flag N dell'ALU (risultato negativo in uscita).

- JAMZ = 1: il bit più significativo viene messo in OR con il flag Z dell'ALU (risultato nullo in uscita).
- JAMC = 1: gli 8 bit meno significativi di NEXT\_ADDRESS sono messi in OR con MBR. In questo modo ottengo un salto di un valore pari a quello contenuto in MBR.

Per caricare la prossima microistruzione è necessario che trascorra un certo intervallo di tempo, necessario al caricamento del nuovo mPC. Il ciclo di clock inizia sul fronte di discesa in cui è disponibile il nuovo valore del mPC, il quale è stato caricato sul fronte di salita dello stesso impulso di clock. E' necessario scegliere una frequenza adeguata del clock, in modo tale che tutti i ritardi dovuti ai vari componenti del datapath siano contenuti e quindi i segnali sui bus si stabiliscano.

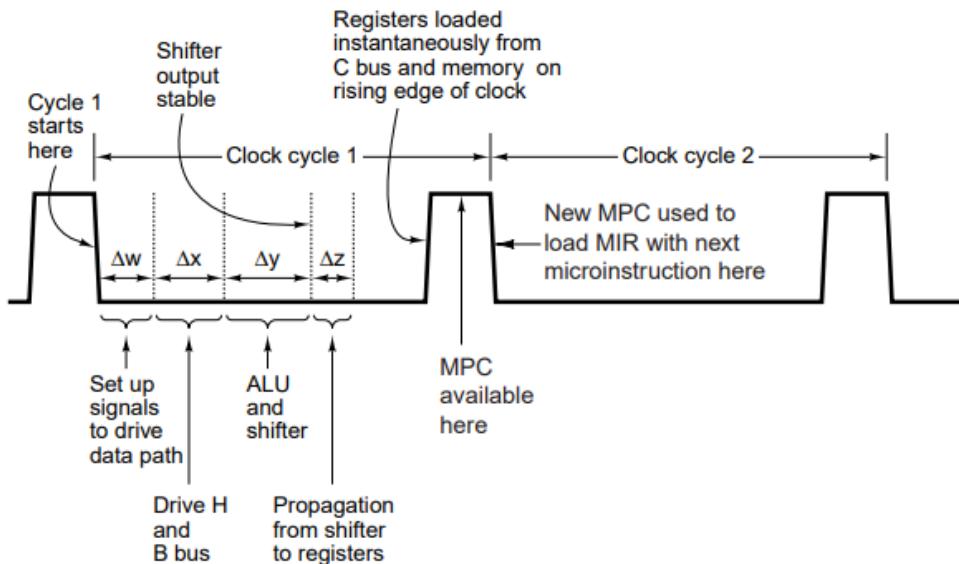


Figure 79: Timing Clock

## **Flusso di esecuzione delle istruzioni:**

Il processore MIC-1 dispone del set di istruzioni relative all' Integer Java Virtual Machine e sfrutta la logica microprogrammata, ovvero realizza ogni istruzione tramite un microprogramma.

Per innescare l'esecuzione di un programma assembler è necessario effettuare prima la traduzione da assembler a microassembler e poi da microassembler a parole di controllo.

A partire dal set di istruzioni IJVM, l'assemblatore Java effettua la corrispondenza tra le istruzioni ed i microprogrammi, i quali verranno tradotti dal microassemblatore MAL in sequenze di segnali di controllo rispetto all'architettura MIC-1. Tutte le sequenze generate vengono inserite all'interno della control\_store del processore.

Una volta scritto il programma in AJVM, tutte le istruzioni IJVM sono tradotte in sequenze di bit che contengono il codice operativo dell'istruzione, ovvero l'indirizzo start del microprogramma contenuto nella control\_store del processore, ed eventuali operandi. Il programma tradotto viene poi inserito nella memoria del sistema.

## **Flusso di esecuzione di esempio:**

```
.main
.var
a
.endvar
BIPUSH 0xA
BIPUSH 0xE
IADD
ISTORE a
HALT // HALT is a no operand instruction but translates as a GOTO (short
value)
.endmethod
```

Per ogni programma vengono eseguite le microprocedure seguenti:

1. Viene prima eseguita la routine mic1\_entry, la quale inizializza i registri del datapath ed inizializza la memoria locale del programma.
2. Viene inizializzato il valore a cui punta LV, il quale contiene l'indirizzo di memoria da cui deve partire lo stack pointer, ossia dopo l'allocazione di memoria per le variabili locali.
3. Viene poi eseguita la procedura main, ed inizia l'effettivo programma.

### **Flusso di esecuzione BIPUSH:**

4. Si effettua il push dell'operando 0xA sullo stack. L'op. bipush prevede:
  - 4.1 Lo spostamento del puntatore SP una cella di memoria in avanti. Il nuovo valore di SP viene poi assegnato al registro MAR;
  - 4.2 Fetch della 2a parte dell'istruzione (operando 0xA);
  - 4.3 Pone in MDR il valore estratto 0xA ed effettua la scrittura in memoria all'indirizzo MAR, ovvero in cima allo stack;
  - 4.4 Go to main
5. Si effettua il push dell'operando 0xE sullo stack. L'op. bipush è analoga.

### **Flusso di esecuzione IADD:**

6. Si effettua l'operazione di IADD, la quale prende gli ultimi due valori nello stack (ossia quello in cima allo stack e quello immediatamente precedente) ed effettuare la somma tra i due, ponendola nello stack alla posizione del primo operando.
  - 6.1 Decremento lo stack pointer per puntare al valore 0xA, lo assegno a MAR ed effettuo la lettura (in MDR avrò 0xA)
  - 6.2 Il registro tampone viene assegnato con il valore in cima allo stack (H = TOS);
  - 6.3 Effettuo la somma tra H e MDR e la pongo in MDR. Tale somma è assegnata al registro TOS dato che la cima dello stack è cambiata. Infine viene effettuata la write del risultato sullo stack alla posizione puntata dal registro MAR.

6.4 Go to main

### **Flusso di esecuzione ISTORE:**

7. si effettua l'operazione di ISTORE, la quale memorizza nella memoria locale del programma il valore dell'operazione.

7.1  $H = LV$

7.2 In MAR viene inserito l'indirizzo della locazione di memoria nella quale sarà effettuata la store. Tale locazione è ricavata dalla somma tra MBRU (unsigned perchè MBR è un registro a 8 bit ed è necessario che abbia spiazzamento a 32 bit, in modo da essere un indirizzo di memoria) ed il registro H, che contiene il valore di LV.

7.3 Si inserisce in MDR il valore che si vuole scrivere in memoria locale, ossia quello della cima dello stack, contenente la somma, e si effettua la scrittura all'indirizzo puntato da MAR.

7.4 Si decrementa lo stack pointer effettuando una pop della somma, si legge il nuovo valore da inserire nel TOS e lo si pone in MDR.

7.5 Si effettua la fetch per spostarsi all'istruzione successiva, poichè l'istruzione ISTORE necessita di due fetch per poter essere eseguita completamente: una per saltare all'operando ed una per andare all'istruzione successiva (questa viene fatta nel main).

7.6 Go to main

## 8.4 Analisi delle istruzioni IADD e ISTORE

### 8.4.1 Istruzione IADD

*iadd = 0x65:*

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

La traduzione in segnali di controllo, rispetto al formato delle microistruzioni descritto precedentemente, è la seguente:

**Microistruzione 1:**  $MAR = SP = SP - 1; rd$

next\_address: 102  
jmp: 000  
alu: 00110110  
bus C: 000001001  
mem: 010  
bus B: 0100

**Microistruzione 2:**  $H = TOS$

next\_address: 103  
jmp: 000  
alu: 00010100  
bus C: 100000000  
mem: 000  
bus B: 0111

**Microistruzione 3:**  $MDR = TOS = MDR + H; wr; goto main$

next\_address: 6  
jmp: 000  
alu: 00111100  
bus C: 001000010  
mem: 100  
bus B: 0000

## Simulazione IADD

Programma ajvm associato:

```
.main
.var
a
.endvar
BIPUSH 0xA
BIPUSH 0xE
IADD
HALT // HALT is a no operand instruction but translates as a GOTO (short
value)
.endmethod
```

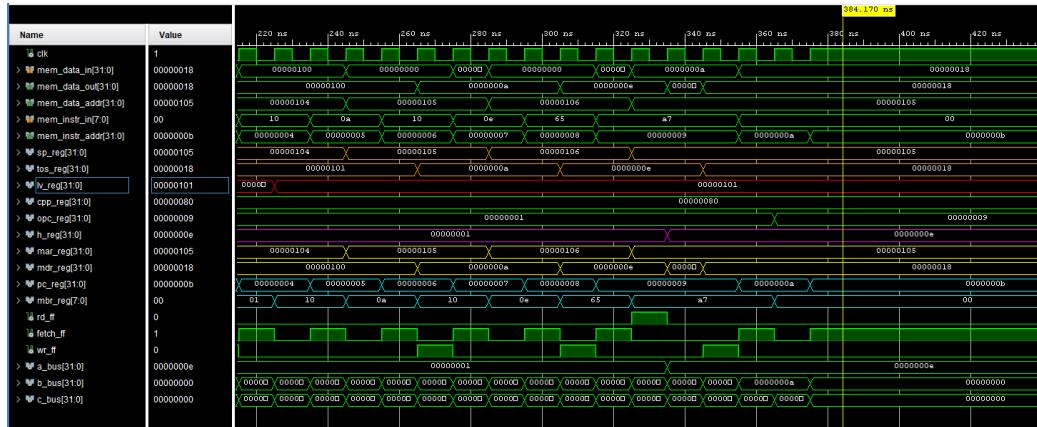


Figure 80: Simulazione IADD

Dalla simulazione si nota che all'esecuzione dell'istruzione IADD (0x36) vengono estratti: 0xA, decrementando lo stack pointer a 0x105, 0xE che viene mantenuto nel registro TOS e viene poi effettuata la somma tra 0xA e 0xE che produce in uscita 0x18, che verrà scritto sulla cima dello stack all'indirizzo 0x105, con un ritardo dovuto al posizionamento nei registri dei valori degli operandi e dalla scrittura in memoria. La simulazione termina appena PC raggiunge il valore 0xB.

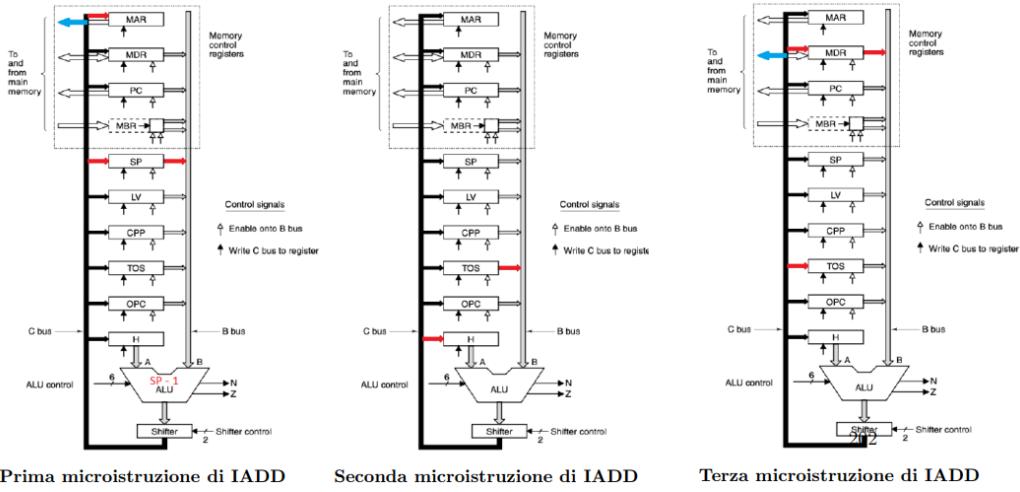


Figure 81: Datapath esecuzione IADD

### 8.4.2 Istruzione ISTORE

*istore* = 0x36:

$H = LV$

$MAR = MBRU + H$

*istore\_cont*:

$MDR = TOS; wr$

$SP = MAR = SP - 1; rd$

$PC = PC + 1; fetch$

$TOS = MDR; goto main$

La traduzione in segnali di controllo, rispetto al formato delle microistruzioni descritte precedentemente, è la seguente:

**Microistruzione 1:**  $H = LV$

next\_address: 55

jmp: 000

alu: 00010100

bus C: 100000000

mem: 000

bus B: 0101

**Microistruzione 2:**  $MAR = MDRU + H$

next\_address: 56

jmp: 000

alu: 00111100

bus C: 000000001

mem: 000

bus B: 0011

**Microistruzione 3:**  $MDR = TOS; wr$

next\_address: 57

jmp: 000

alu: 00010100

bus C: 000000010

mem: 100

bus B: 0111

**Microistruzione 4:**  $SP = MAR = SP - 1; rd$

next\_address: 58  
jmp: 000  
alu: 00110110  
bus C: 000001001  
mem: 010  
bus B: 0100

**Microistruzione 5:**  $PC = PC + 1; fetch$

next\_address: 59  
jmp: 000  
alu: 00110101  
bus C: 000000100  
mem: 001  
bus B: 0001

**Microistruzione 6:**  $TOS = MDR; goto main$

next\_address: 6  
jmp: 000  
alu: 00010100  
bus C: 001000000  
mem: 000  
bus B: 0000

## Simulazione ISTORE

Programma AJVM associato:

```
.main
.var
a
.endvar
BIPUSH 0xA
ISTORE
HALT // HALT is a no operand instruction but translates as a GOTO (short
value)
.endmethod
```

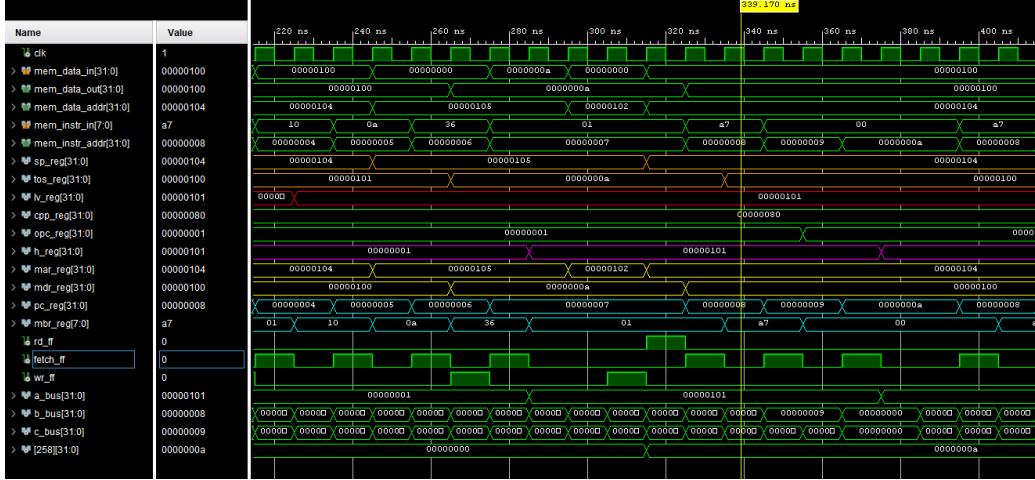


Figure 82: Simulazione ISTORE

Dalla simulazione si nota che:

- Viene caricato il registro H con il valore di LV, ossia 0x101;
- All'indirizzo base LV viene aggiunto l'offset di 0x01 per puntare alla variabile A nel Local Variable Frame e viene inserito nel MAR ottenendo 0x102;
- Si effettua poi la write in memoria all'indirizzo 0x102 del valore in TOS, ossia 0xA;
- Si decrementa poi lo stack pointer a 0x104 e si riaggiorano i registri.

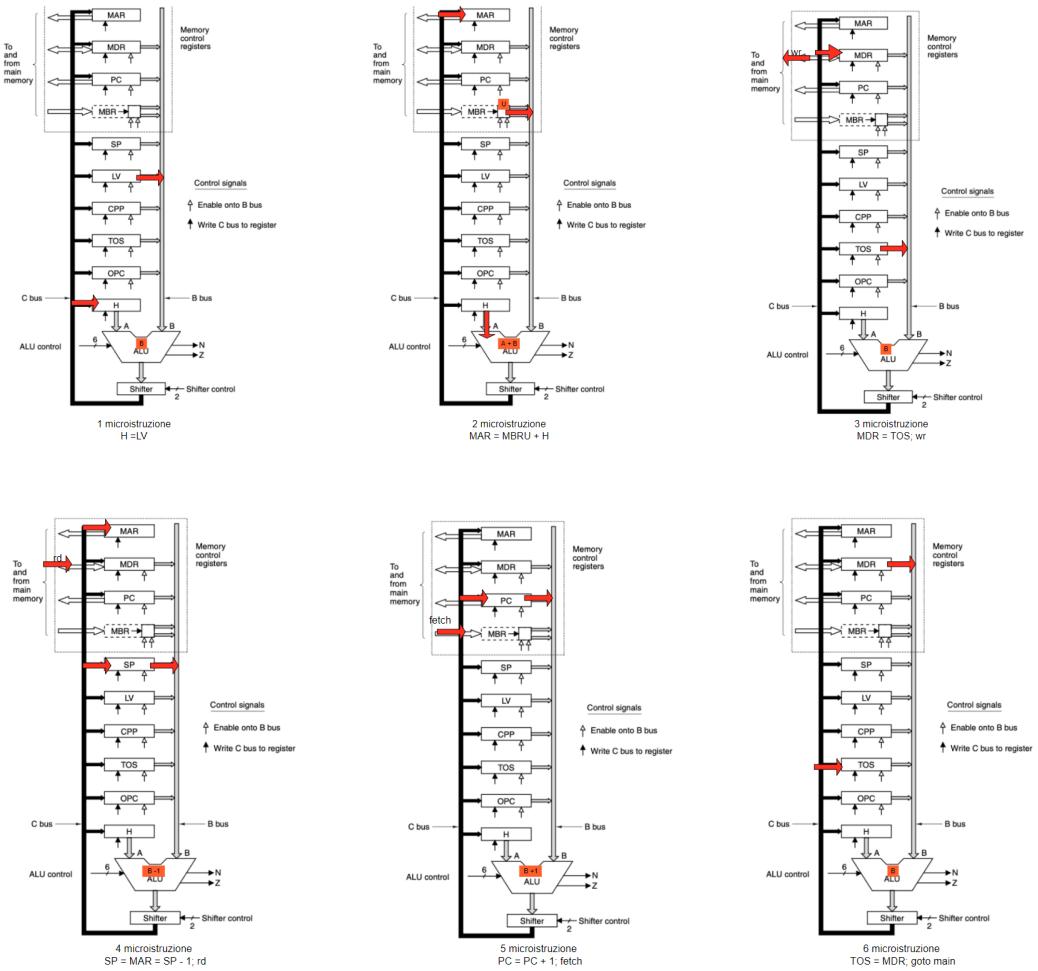


Figure 83: Datapath esecuzione ISTORE

### 8.4.3 Modifica istruzione IADD

E' stato modificato il microprogramma relativo all'istruzione IADD in questo modo:

*iadd = 0x65:*

$$MAR = SP = SP - 1; rd$$

$$H = TOS$$

$$H = H+1$$

$$MDR = TOS = MDR + H; wr; goto main$$

L'istruzione di IADD, oltre ad effettuare la somma tra i due operandi, effettua anche l'incremento di 1 al valore ottenuto. Nel microprogramma viene incrementato di 1 il valore dell'operando in cima allo stack, contenuto nel registro H. L'istruzione IADD, in questo caso, impiegherà un ciclo di clk in più ad essere eseguita, per via della microistruzione di incremento.

Il programma AJVM associato alla istruzione IADD è lo stesso precedente.

### Simulazione IADD Modificata

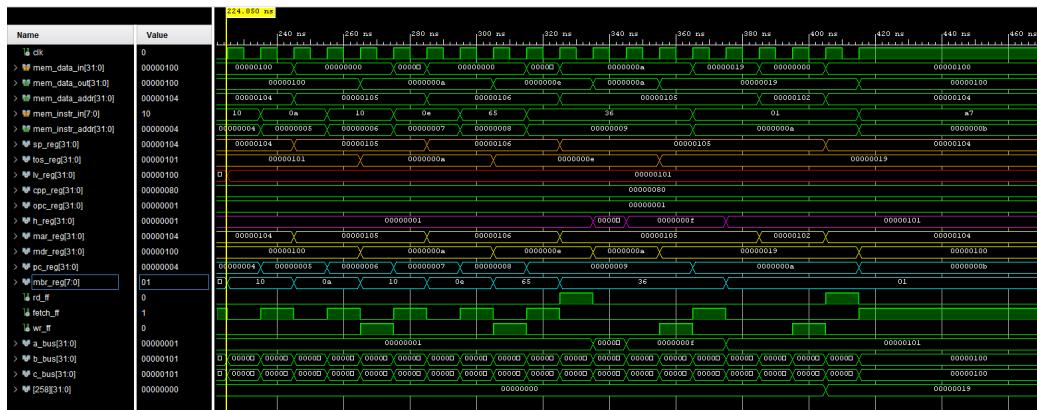


Figure 84: Simulazione IADD Modificata

Dalla simulazione si nota che l'esecuzione dell'istruzione IADD (0x36) è esattamente uguale a quella precedente, se non per l'incremento intermedio del valore in cima allo stack, che passa da 0xE a 0xF.

# 9 Esercizio 9 - Interfaccia UART

## 9.1 Introduzione all'UART

L'UART (Universal Asynchronous Receiver-Transmitter) è un dispositivo hardware che converte flussi di bit di dati da un formato parallelo ad uno seriale e viceversa e, generalmente, è parte di un circuito integrato.

Il componente fondamentale di tale dispositivo è uno shift register, il quale ci permette di convertire frame paralleli in seriali e viceversa.

Il protocollo di trasmissione implementato dall'UART è di tipo asincrono ed il frame trasmesso è composto da:

- 1 bit di start
- 8 bit su cui è espresso il carattere
- 1 bit di controllo
- 1 o più bit di stop

Implementando un protocollo di tipo asincrono, a differenza di uno sincrono, manca la serie di byte iniziali che definisce proprio il sincronismo tra le 2 entità che vogliono comunicare.

Quando non ci sono dati da trasmettere, la linea dati si definisce “a riposo” ed è impostata sul livello logico alto; appena invece ci sono nuovi dati da trasmettere, il valore logico passa da 1 a 0, per poi trasmettere effettivamente il dato.

La trasmissione parte, dunque, con un valore logico basso e, di conseguenza, termina con valore logico alto: ciò implica che ci saranno almeno 2 variazioni del segnale durante la trasmissione.

Al fine di interpretare i dati trasmessi correttamente, il dispositivo in ricezione deve capire, in un intervallo di tempo, quanti bit ha ricevuto; per tale motivo, i 2 dispositivi devono accordarsi su alcuni parametri della trasmissione:

- Velocità dei bit (baud rate)
- Numero di bit per carattere
- Numero di bit di stop
- Numero di bit di parità

Generalmente, il numero di bit totali trasmessi è pari a 10, con 8 bit per carattere, un bit di start, uno di stop e nessun bit di parità.

Il bit di parità è un metodo di rilevamento, ma non di correzione, di un errore avvenuto su un numero dispari di bit, che prevede l'aggiunta di un bit al termine del carattere, in modo che il numero di bit alti all'interno del dato sia pari.

Tale errore, noto come parity error, non è però l'unico che potrebbe verificarsi; ci sono infatti altri 2 tipi di errore da tenere in considerazione:

- Overrun error: si verifica quando il destinatario sta ancora elaborando il dato precedente e ne riceve uno nuovo; tale errore porta alla perdita del dato.
- Farming error: dopo la rilevazione di tutti i bit relativi al carattere, il dispositivo in ricezione si aspetta un bit di stop; se questo non arriva, vuol dire che c'è stato un errore di campionamento oppure un errore durante la trasmissione.

Per evidenziare la presenza di errori, tuttavia, il dispositivo in ricezione deve essere sicuro di campionare correttamente i valori ricevuti; a questo scopo, la parte in ricezione dell'uart lavora con un clock interno a frequenza molto più alta (in genere 16 volte), in modo da campionare il valore trasmesso “al centro”, distante da fasi transitorie.

Le 2 frequenze di clock presenti vengono implementate mediante degli stati di wait nelle FSM e gestiti tramite i valori di baud rate e baud divide presenti nell'implementazione.

Un ulteriore modo per valutare se il bit ricevuto è pari a 0 o 1 è quello di campionare il dato N volte in ingresso tramite il clock e contare il numero di volte in cui leggo 0 e 1: il valore che leggo più volte sarà quello che salvo.

Un'implementazione di questo tipo è però più costosa della precedente e raramente viene implementata; in genere si preferisce ritrasmettere il dato piuttosto che effettuare tali controlli.

Al termine di questa breve introduzione sul funzionamento dell'UART, vediamo ora ingressi ed uscite relative alla parte di ricezione e di trasmissione.

## Trasmissione

Ingressi:

- DBIN: dato da trasmettere, formato parallelo;
- WR: segnale di write che dà il via alle operazioni per la trasmissione del dato sul canale.

Uscite:

- TXD: dati trasmessi in formato seriale;
- TBE: indica lo stato del buffer di trasmissione: quando è vuoto è pari a 1, 0 altrimenti.

## Ricezione

Ingressi:

- RXD: dati in formato seriale in ingresso;
- RD: segnale di strobe, ovvero di confermata lettura del dato ricevuto;
- RDA: indica la disponibilità di nuovi dati.

Uscite:

- DBOUT: dato ricevuto, formato parallelo;
- RDA: indica la disponibilità di nuovi dati in ingresso;
- PE: parity error;
- OE: Overrun error;
- FE: Farming error.

Entrambe le unità, inoltre, presentano un ingresso per il clock ed uno per il segnale di reset.

## 9.2 Parte 1

Tale esercizio richiedeva una trasmissione di 1 carattere da 8 bit tra 2 sistemi A e B, utilizzando il componente standard UART fornito dalla Digilent.

Si sono implementati 2 componenti, A e B, progettati tramite approccio strutturale ma composti unicamente dal componente UART, dato che la stringa da trasferire viene passata dall'esterno, così come i segnali di write e reset.

```
entity sistemaA is
  Port ( input : in STD_LOGIC_VECTOR (7 downto 0);
         output : out STD_LOGIC;
         clk : in STD_LOGIC;
         reset : in STD_LOGIC;
         write: in std_logic);
end sistemaA;

architecture Structural of sistemaA is
component UARTcomponent
Generic (
  BAUD_DIVIDE_G : integer := 54;  --BAUD_RATE/16
  BAUD_RATE_G   : integer := 869 --100Mhz/115200
);
port(CLK: in std_logic;
      TXD: out std_logic := '1';
      DBIN: in std_logic_vector (7 downto 0);
      WR: in std_logic;
      RST   : in std_logic:= '0'
);
end component;
for all: UARTcomponent use entity work.UARTcomponent(Behavioral);

begin
uartA: UARTcomponent port map(CLK => clk, RST => reset, DBIN => input, WR => write, TXD => output);
end Structural;
```

Figure 85: Codice Sistema A

```

entity sistemaB is
    Port ( input : in STD_LOGIC;
           output : out STD_LOGIC_VECTOR (7 downto 0);
           clk : in STD_LOGIC;
           reset : in STD_LOGIC);
end sistemaB;

architecture Structural of sistemaB is
component UARTcomponent
Generic (
    BAUD_DIVIDE_G : integer := 54;
    BAUD_RATE_G   : integer := 869
);
port(CLK: in std_logic;
     RXD: in std_logic;
     DBOUT: out std_logic_vector (7 downto 0);
     RST   : in std_logic:= '0');

end component;
for all: UARTcomponent use entity work.UARTcomponent(Behavioral);
begin

uartB: UARTcomponent port map(CLK => clk, RST => reset, RXD => input, DBOUT => output);

end Structural;

```

Figure 86: Codice Sistema B

Il sistema complessivo comprende inoltre 2 debouncer, utilizzati per ripulire i bottoni per i segnali di write e reset.

Si è deciso di realizzare 2 entità e di non utilizzare direttamente i componenti UART per poterle riutilizzare nell'esercizio 9.2.

## Sintesi su FPGA

Per la sintesi su board, così come richiesto dalla traccia, sono stati utilizzati gli switch per definire la stringa di input al sistema A e dei led per visualizzare la stringa riconosciuta dal sistema B.

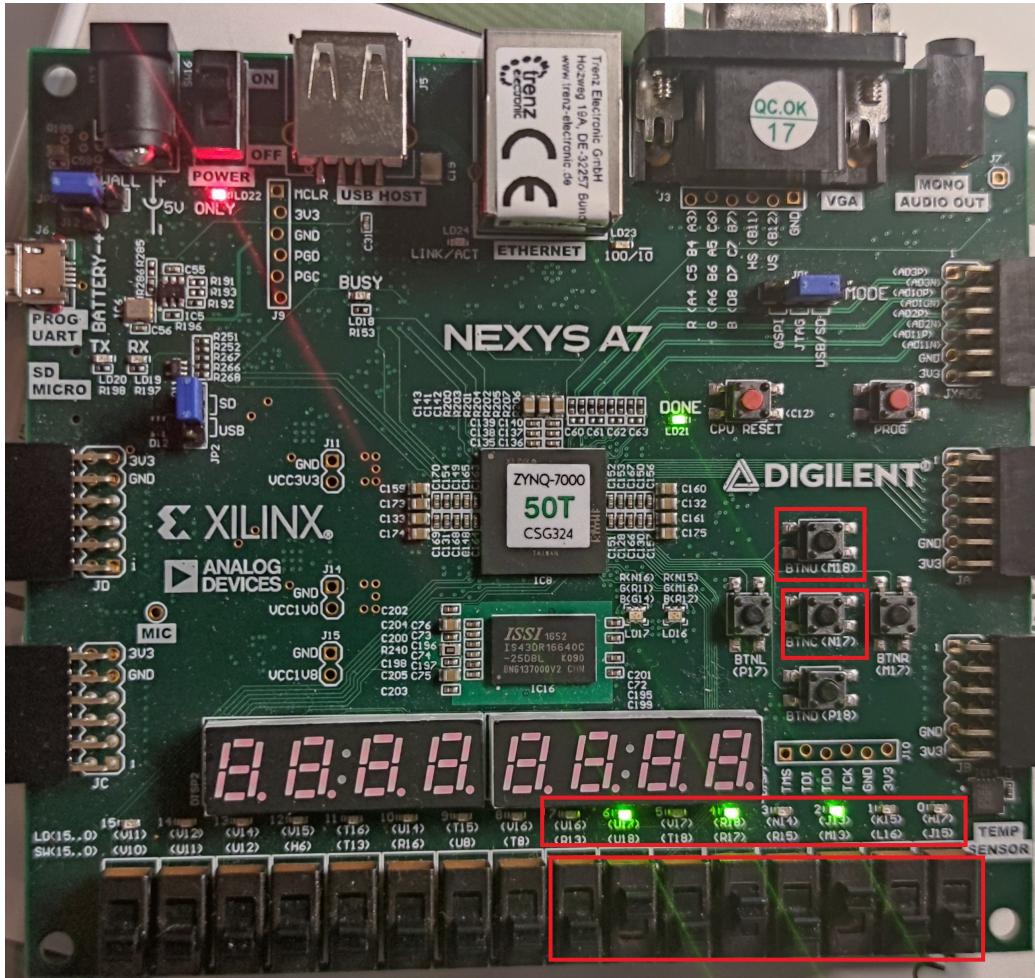


Figure 87: Sintesi su FPGA Es. 9.1

## 9.3 Parte 2

L'esercizio è stato impostato sulla falsa riga dell'esercizio 7, dato che la richiesta di leggere un valore in memoria e trasferirlo era analoga.

La differenza sta nel fatto che il protocollo di trasmissione non è parallelo, ma seriale ed è implementato dall'UART. Così come fatto però nell'esercizio 7, viene implementato anche un protocollo di handshaking per la comunicazione tra i due sistemi, al fine di evitare errori di overrun dei dati.

### 9.3.1 Sistema A

Il sistema A è stato realizzato attraverso un approccio strutturale partendo dall'entità sviluppata nell'esercizio 9.1, e comprende:

- Control unit: ha il compito di effettuare l'handshaking con il sistema B, inviare il segnale di read alla memoria e contemporaneamente quello di write all'UART, per definire la trasmissione. La control unit ha inoltre il compito di effettuare una ritrasmissione nel caso in cui il sistema B notifichi un errore in ricezione.
- Contatore: per scandire il termine della trasmissione ed i valori da trasmettere.
- ROM: contenente i valori da trasferire.
- Delay block: per applicare un delay ai segnali quando necessario.
- UART: responsabile della trasmissione dei dati sul canale.

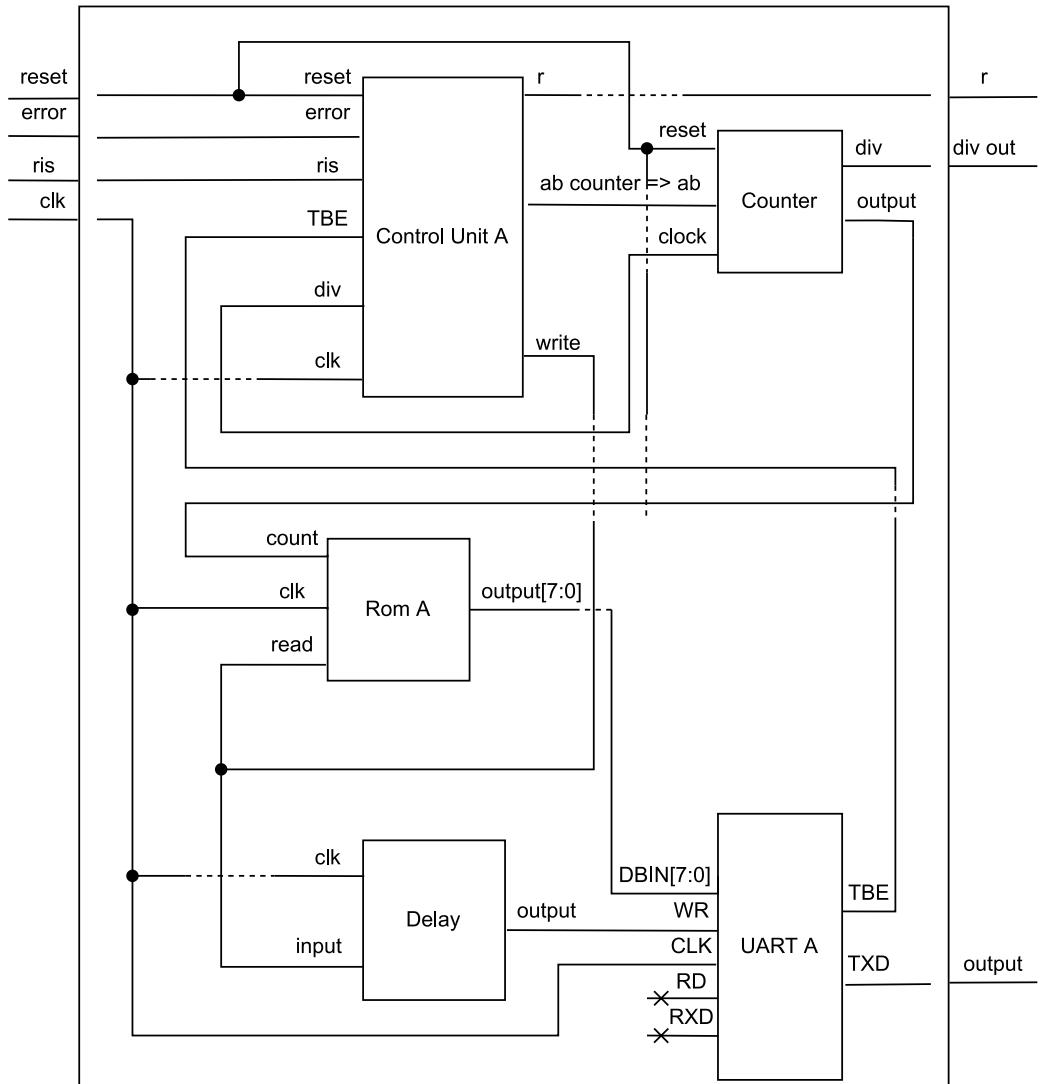


Figure 88: Schema Sistema A Es. 9.2

### 9.3.1.1 Unità Operativa A

#### Componente Contatore:

```
entity contatore is
  Generic( N:integer);
  Port ( clock : in STD_LOGIC;
          reset : in STD_LOGIC;
          ab : in STD_LOGIC:='0';
          output : out STD_LOGIC_VECTOR (1 to integer( ceil( log2(real(N) ) ) )):=(others =>'0');
          div : out STD_LOGIC:='0');
end contatore;
```

Figure 89: Entity Contatore

E' stato realizzato tramite approccio comportamentale e l'incremento è definito dal segnale di ab\_counter:

```
architecture Behavioral of contatore is
signal tmp: std_logic_vector(1 to integer(ceil(log2(real(N))))):=(others =>'0');
signal temp_div:std_logic:='0';
begin
process (clock)
begin
  if( clock = '1' and clock'event) then
    if(reset = '1') then
      temp_div<= '0';
      tmp <= (others => '0');
    elsif ( ab = '1') then
      if ( tmp >= N-1) then
        tmp <= (others => '0');
        temp_div <= '1';
      else
        tmp <= tmp + 1;
      end if;
    end if;
  end process;
  output <= tmp;
  div <= temp_div;
end Behavioral;
```

Figure 90: Architecture Contatore

#### Componente ROM:

Tale componente è analogo a quello utilizzato nei progetti precedenti e pertanto non viene riportata nuovamente la sua implementazione.

### 9.3.1.2 Unità di Controllo A

Control Unit A:

```
entity control_unitA is
    Port ( clk : in STD_LOGIC;
            div : in STD_LOGIC;
            ris,error : in STD_LOGIC;
            r : out STD_LOGIC:='0';
            reset : in STD_LOGIC;
            write, ab_counter : out STD_LOGIC:='0';
            TBE :  in STD_LOGIC);
end control_unitA;
```

Figure 91: Entity Control Unit A Es. 9.2

La control unit è definita come una macchina a stati, dove alcune uscite sono associate direttamente agli stati, mentre altre sono associate alla transizione tra stati in base agli ingressi:

1. q\_start → tale stato è quello in cui si trova la macchina prima di iniziare la trasmissione. La macchina permane in tale stato finché non vede alzarsi il segnale di reset (essendo uscita di un debouncer durerà solo 1 colpo di clock) che viene interpretato come segnale per partire con la trasmissione. A questo punto la macchina passa nello stato q\_trasmett.
2. q\_trasmett → in questo stato la control unit verifica che il segnale di ris in uscita dal sistema B sia basso, ovvero che il sistema non stia elaborando un dato precedentemente inviato. Nel caso in cui il dato trasmesso sia il primo, tale segnale sarà sempre basso; questo controllo è necessario per i dati successivi al primo. Se il segnale di ris viene visto basso, allora il sistema alza il segnale di r e passa nello stato q0, altrimenti permane nello stato q\_trasmett.
3. q0 → tale stato è uno stato di transizione per completare l'handshaking, prima di iniziare la trasmissione vera e propria del dato. Alzando il segnale di r, il sistema A ha espresso la sua intenzione nel trasmettere dei

dati, avendo visto che il sistema B è disponibile a ricevere ed elaborarli, ma non ha ancora saputo da esso se effettivamente è pronto a ricevere tali dati o è occupato in operazioni con priorità maggiore. Pertanto, il sistema A rimane nello stato q0 finché non vede alzarsi il segnale di ris da parte del sistema B, sintomo che quest'ultimo è pronto per elaborare i dati trasmessi. Quando ciò accade, si passa nello stato q1 e si abbassa il segnale di r.

4. q1 → arrivati in questo stato vuol dire che può iniziare la vera e propria trasmissione dei dati. Per questo motivo, a tale stato è associata l'uscita write = '1', che viene passata alla memoria ROM come un segnale di read per emettere il dato, e viene inoltre passata con delay di un periodo di clock all'UART, per segnalare la possibilità di trasmettere i dati in ingresso. Il motivo del colpo di delay è dato dal tempo che la memoria impiega per leggere il dato e porlo in uscita. Si passa a questo punto nello stato q2.
5. q2 → allo stato q2 è associata l'uscita write ='0', dato che tale segnale deve durare 1 colpo di clock per evitare ulteriori letture non richieste alla memoria. Il fatto che il segnale di write però si abbassi, non vuol dire che la trasmissione sia effettivamente iniziata. Per questo motivo si effettua un controllo su TBE, ovvero il segnale in uscita dall'uart che, quando è 0 indica che il canale è occupato e dunque la trasmissione è iniziata, altrimenti inidica un canale ancora libero. A questo punto, quando il valore di TBE è diventato 0, si passa nello stato q3, altrimenti si permane in q2.
6. q3 → è lo stato in cui avviene il controllo dell'errore. Finchè ris è alto, ovvero il sistema B sta ancora elaborando i dati, si permane nello stato q3, altrimenti si valuta il segnale di error in entrata dal sistema B. Se il segnale di errore è pari ad 1, non si incrementa ab\_counter e dunque verrà ritrasmesso lo stesso dato, altrimenti il contatore viene incrementato e si procede alla trasmissione del dato successivo se ce ne sono di nuovi. Indistintamente si passa poi nello stato q4.
7. q4 → in tale stato semplicemente si valuta il valore del div del contatore. Se questo è pari a 1, si ritorna nello stato di start e si attende un nuovo segnale di reset per trasmettere nuovi dati, altrimenti si ripassa nello stato q0 e si attende che il segnale di ris da parte del sistema B si abbassi

per procedere ad un nuovo handshaking ed una nuova trasmissione. Si noti inoltre che a tale stato è associata l'uscita ab\_counter='0'. Il segnale di ab\_counter serve a dare il segnale di conteggio al contatore e dunque trasmettere il dato seguente, cosa che avviene se il dato è stato ricevuto correttamente dal sistema B e viene abbassato poiché deve durare un colpo di clock per evitare conteggi spuri.

```

architecture Behavioral of control_unitA is
type state is (q_start,q_trasmett,q0,q1,q2,q3,q4);
signal current_state:state:=q_start;
signal temp_write, temp_ab_counter: std_logic:='0';
begin
process(clk)
variable count:integer:=0;
begin
    if(clk='0' and clk'event) then
        case current_state is
            when q_start =>
                if(reset = '1') then
                    current_state <= q_trasmett;
                else
                    current_state <= q_start;
                end if;
            when q_trasmett => if(ris = '1') then
                    current_state <= q_trasmett;
                else
                    current_state <= q0;
                    r <= '1';
                end if;
        end case;
    end if;
end process;
end Behavioral;

```

```

when q0 => temp_ab_counter <= '0';
if( ris = '0') then
    current_state <= q0;
else
    current_state <= q1;
    r<= '0';
end if;

when q1 => temp_write <= '1';
current_state <= q2;

when q2 => temp_write <= '0';
if(TBE = '1') then
    current_state <= q3;
else
    current_state <= q2;
end if;

when q3 => if(ris = '1') then
    current_state <= q3;
elsif( error = '0') then
    temp_ab_counter <= '1';
    current_state <= q4;
    --r <= '1';
else
    current_state <= q4;
    --r <= '1';
end if;

when q4 => temp_ab_counter <= '0';
if(div = '0') then
    current_state<= q0;
    r<='1';
else
    current_state<= q_start;
end if;

when others => current_state <= q_start;
end case;
end if;

end process;
write <= temp_write;
ab_counter <= temp_ab_counter ;
end Behavioral;

```

Figure 92: Architecture Control Unit A Es. 9.2

## **Considerazioni finali:**

La control unit ha come particolarità quella di lavorare sul fronte di discesa del clock, ovvero in opposizione rispetto a tutti i componenti a cui fornisce dei segnali, in modo che questi li vedano quando sono assestati e non durante la fase di transizione.

Il sistema di rilevamento dell'errore e ritrasmissione, implementata tramite il segnale di error, potrebbe andare in loop infinito se gli errori persistono e non vengono risolti con una semplice ritrasmissione. Per evitare ciò andrebbe elaborato un sistema di correzione oltre che uno di ritrasmissione.

### **9.3.2 Sistema B**

Il sistema B è stato realizzato attraverso un approccio strutturale partendo dall'entità sviluppata nell'esercizio 9.1, e comprende:

- Control Unit: per coordinare i dispositivi ed effettuare l'handshaking con il sistema A;
- UART: responsabile della ricezione dei dati da un canale seriale e del loro posizionamento in parallelo in uscita;
- Contatore: per fornire l'indice della prima cella di memoria disponibile per salvare il dato ricevuto;
- Memoria: per contenere i dati ricevuti.

Siccome i dati salvati non vengono riportati in uscita, è stato introdotto un debouncer interno per il segnale di read dato dall'esterno tramite un bottone, in modo che ad ogni segnale si scorra di una posizione la memoria e viene riportato in uscita il valore corrispondente.

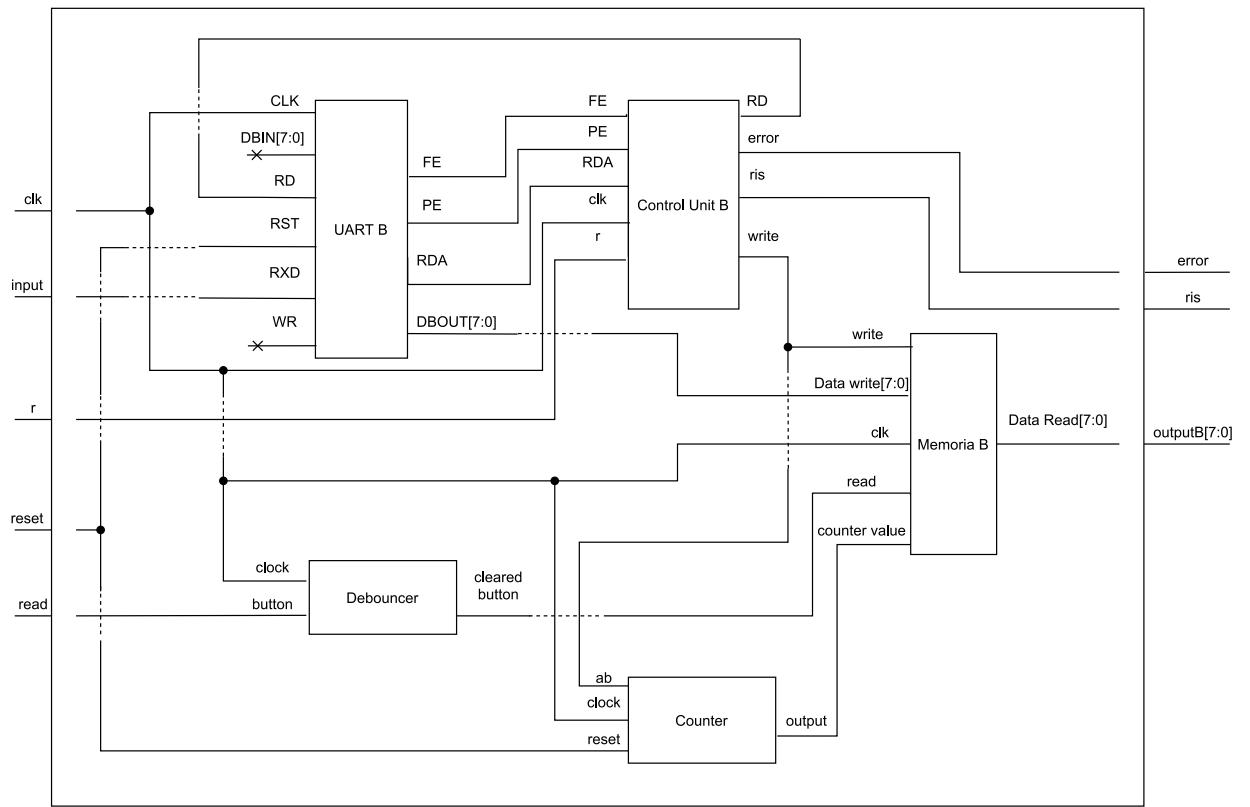


Figure 93: Schema Sistema B Es. 9.2

### 9.3.2.1 Unità Operativa B

Il componente contatore è analogo a quello utilizzato per il sistema A, mentre il componente ROM è analogo a quello presentato nei progetti precedenti; pertanto non verrà riproposta la loro implementazione.

### 9.3.2.2 Unità di Controllo B

Componente Control Unit B:

```
entity control_unitB is
    Port ( RDA : in STD_LOGIC;
            ris,error : out STD_LOGIC:='0';
            r, PE, FE: in STD_LOGIC:='0';
            write, RD : out STD_LOGIC:='0';
            clk : in STD_LOGIC);
end control_unitB;
```

Figure 94: Entity Control Unit B Es. 9.2

La Control Unit è stata realizzata come una macchina a stati, dove alcune uscite sono associate direttamente agli stati, altre alle transizioni in funzione degli ingressi:

1. q\_start → tale stato è quello in cui permane la macchina prima di rilevare un valore di r alto da parte del sistemaA, segno di richiesta di iniziare una nuova trasmissione. Quando viene rilevato tale segnale, si passa nello stato q0 e si pone alto il segnale di ris.
2. q0 → si permane in questo stato finchè la ricezione dei dati da parte dell'UART non è terminata e quest'ultimi sono presenti nel buffer di uscita. Per capire quando effettivamente ciò avviene, si sfrutta il segnale RDA in uscita dall'UART il quale, quando è pari a 1, indica la presenza dei dati nel buffer. Quando tale segnale viene visto alto quindi, si alza il segnale di RD, che viene dato come strobe per la lettura, e si effettua un controllo sui valori di PE ed FE in uscita dall'UART. Se uno dei due è alto vuol dire che c'è stato un errore nella trasmissione e dunque viene alzato il segnale di error, ma non quello di write, ovvero viene notificato l'errore ad A ed il valore letto non viene salvato in memoria. Nel caso di ricezione corretta, invece, viene alzato il segnale di write ed abbassato il segnale di error, il quale che poteva essere alto nella trasmissione precedente. Non abbiamo necessità che il segnale di error duri 1 colpo di clock, ma solo che il suo valore venga aggiornato prima

che si abbassi il segnale di ris, poiché è solo in quel momento che tale valore verrà letto dal sistema A. Da qui, si passa nello stato q1.

3. q1 → a tale stato è associata l'uscita write = '0', poiché si vuole che tale segnale rimanga alto solo un colpo di clock per evitare conteggi spuri, dato che tale segnale viene dato come abilitazione al contatore. Viene inoltre abbassato il segnale di RD, il quale sostanzialmente comunica all'UART che il dato posto in uscita è stato letto. Così come definito dal protocollo di handshaking implementato nell'esercizio 7 e riutilizzato per questo progetto, si attende che il segnale di r sia basso prima di abbassare il segnale di ris e passare nello stato q\_start.

```

architecture Behavioral of control_unitB is
type state is(q_start,q0,q1);
signal current_state: state:=q_start;
signal temp_write:std_logic:= '0';

begin
process(clk)
begin
if(clk='0' and clk'event) then
    case current_state is
        when q_start => if(r = '0') then
            current_state <= q_start;
        else
            current_state <= q0;
            ris <= '1';
        end if;

        when q0 => if(RDA = '1') then
            RD <= '1';
            if(PE = '1' or FE = '1') then
                current_state <= q1;
                error <= '1';
            else
                temp_write <= '1';
                current_state <= q1;
                error<= '0';
            end if;
            else
                current_state <= q0;
            end if;
        end if;
        when q1 => temp_write <= '0';
        RD <= '0';
        if(r = '0') then
            current_state <= q_start;
            ris <= '0';
        else
            current_state <= q1;
        end if;
        when others => current_state <= q0;
    end case;
end if;
end process;
write <= temp_write;
end Behavioral;

```

Figure 95: Architecture Control Unit B Es.9.2

Così come fatto per la Control Unit A, anche questa Control Unit è impostata sul fronte di discesa del clock, per far sì che le unità controllate vedano i segnali stabili e non sulle trasizioni.

## Sintesi su FPGA

A causa dell'alto tempo di simulazione e del numero di segnali da visualizzare, si è ritenuto poco opportuno riportare esempi di simulazione (la quale può essere mandata in esecuzione tramite il file del progetto) e si presenterà unicamente il funzionamento sull'fpga.

Il led V11 (sulla sinistra) corrisponde al segnale di div del contatore del sistema A e rappresenta la fine della trasmissione.

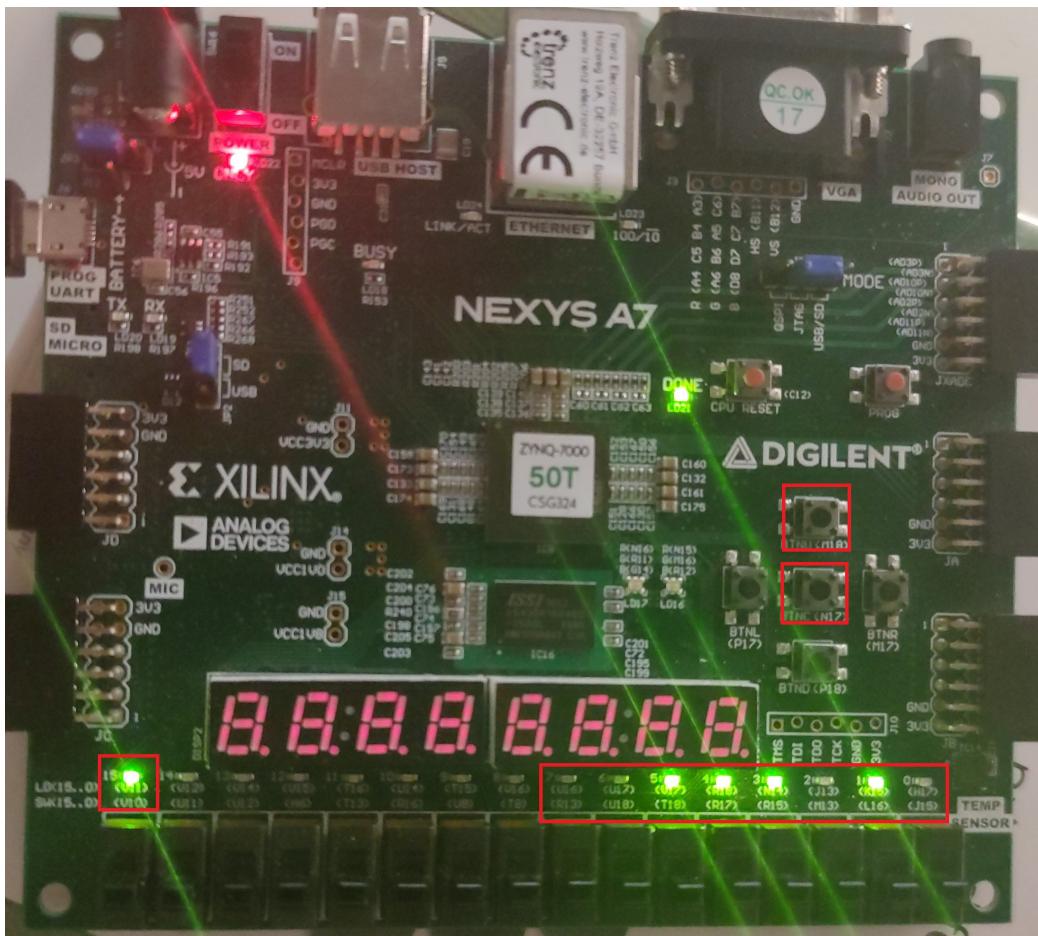


Figure 96: Sintesi su FPGA Es. 9.2

## 10 Esercizio 10 - Switch Multistadio

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.

### 10.1 Approccio utilizzato

Per la realizzazione dello switch multistadio utilizzando il modello omega network, è stato utilizzata un'architettura composta da una parte operativa ed una parte di controllo. Il numero di nodi sia in ingresso che in uscita è 4 e la priorità è fissa. Si suppone che, per iniziare la comunicazione, ci sia un segnale di START in ingresso al sistema e che ogni nodo invii un segnale di ENABLE in ingresso per abilitare la comunicazione. Inoltre, si presuppone che possa avvenire una sola comunicazione in base alla priorità, quindi, se lo switch è occupato ad instradare un messaggio, allora eventuali altri pacchetti in arrivo andranno perduti.

Si suppone che il pacchetto dati da inviare da una sorgente allo switch sia composto da 2 bit dato e 2 bit indirizzo destinazione, mentre l'uscita dello switch è composto dai soli 2 bit dato.

## 10.2 Parte Operativa

La parte operativa realizza il modello di Omega Network su 4 nodi sorgente e 4 nodi destinazione, è stata realizzata con un approccio strutturale, utilizzando come componenti elementari dei blocchi di instradamento. Questi non sono altro che la composizione di un Mux 2:1 e di un Demux 1:2 : per il primo l'indirizzo di selezione è quello sorgente, mentre per il secondo è quello destinazione. Tale componente è stato realizzato con un approccio Dataflow.

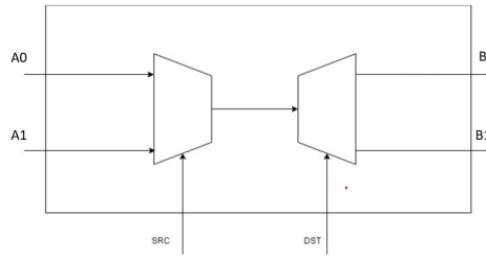


Figure 97: Schema blocco instradamento

```
entity blocco_instr is
  port(
    data_in1: in std_logic_vector(0 to 1);
    data_in2: in std_logic_vector(0 to 1);
    data_out1: out std_logic_vector(0 to 1);
    data_out2: out std_logic_vector(0 to 1);
    source: in std_logic;
    dest: in std_logic
  );
end blocco_instr;

architecture Dataflow of blocco_instr is

begin
  data_out1 <= data_in1 when source = '0' and dest = '0' else
    data_in2 when source = '1' and dest = '0' else
    "00";
  data_out2 <= data_in1 when source = '0' and dest = '1' else
    data_in2 when source = '1' and dest = '1' else
    "00";

end Dataflow;
```

Figure 98: Architecture blocco instradamento

Sfruttando l'approccio strutturale, si collegano i componenti elementari in accordo con il perfect shuffling. Secondo questa tecnica, avendo M carte, dopo un numero di iterazioni del mescolamento perfetto pari a  $\log_2 M$  si riottiene l'ordinamento di partenza; analogamente, per connettere completamente M nodi serviranno  $\log_2 M$  stadi. Nel nostro caso abbiamo 4 nodi da connettere, quindi 2 stadi con 2 switch ciascuno. Le stringhe contenenti gli indirizzi sono lunghe 2 bit, in quanto i nodi destinazione sono 4 e, avendo assunto che la comunicazione si verifichi da sinistra verso destra, i bit di destinazione vanno dati ai vari stadi (uno per ciascuno) a partire dal bit più significativo. I bit di sorgente, invece, vanno dati dal bit meno significativo al più significativo.

```

entity operative_unit is
  GENERIC ( N : integer:= 2);
  Port (
    ind_source: in std_logic_vector(0 to 1);
    ind_dest: in std_logic_vector(0 to 1);
    data_in: in std_logic_vector(0 to 4*N-1);
    data_out: out std_logic_vector(0 to 4*N-1)
  );
end operative_unit;

```

Figure 99: Entity Unità Operativa

Il generic N permette di scegliere il numero di bit dato trasmessi nella comunicazione all'interno dell'Omega Network.

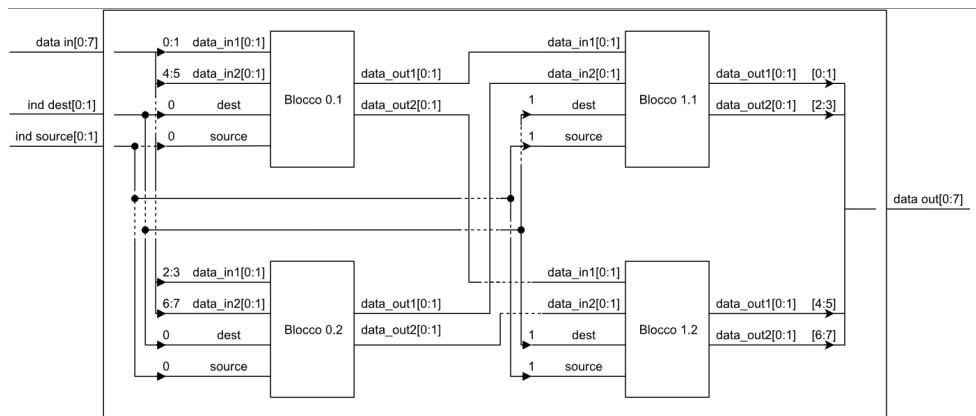


Figure 100: Unità Operativa

```

architecture Structural of operative_unit is
  component blocco_instr
    port(
      data_in1: in std_logic_vector(0 to N-1);
      data_in2: in std_logic_vector(0 to N-1);
      data_out1: out std_logic_vector(0 to N-1);
      data_out2: out std_logic_vector(0 to N-1);
      source: in std_logic;
      dest: in std_logic
    );
  end component;
  type temp_vector is array(0 to 3) of std_logic_vector(0 to N-1);
  signal vettore_segnali: temp_vector;

begin
  blocco0_1: blocco_instr port map(
    data_in1 => data_in(0 to N-1),
    data_in2 => data_in(2*N to 3*N-1),
    data_out1 => vettore_segnali(0),
    data_out2 => vettore_segnali(1),
    source => ind_source(0),
    dest => ind_dest(0)
  );
  blocco0_2: blocco_instr port map(
    data_in1 => data_in(N to 2*N-1),
    data_in2 => data_in(3*N to 4*N-1),
    data_out1 => vettore_segnali(2),
    data_out2 => vettore_segnali(3),
    source => ind_source(0),
    dest => ind_dest(0)
  );
  bloccol_1: blocco_instr port map(
    data_in1 => vettore_segnali(0),
    data_in2 => vettore_segnali(2),
    data_out1 => data_out(0 to N-1),
    data_out2 => data_out(N to 2*N-1),
    source => ind_source(1),
    dest => ind_dest(1)
  );
  bloccol_2: blocco_instr port map(
    data_in1 => vettore_segnali(1),
    data_in2 => vettore_segnali(3),
    data_out1 => data_out(2*N to 3*N-1),
    data_out2 => data_out(3*N to 4*N-1),
    source => ind_source(1),
    dest => ind_dest(1)
  );
end Structural;

```

Figure 101: Architecture Unità Operativa

### 10.3 Parte di Controllo

```

component control_unit
  GENERIC ( N : integer:= 2);
  port(
    clk: in std_logic;
    enable: in std_logic_vector(0 to 3);
    start: in std_logic;
    data_in: in std_logic_vector(0 to (N+2)*4-1);
    data_out: out std_logic_vector(0 to 4*N-1);
    ind_source: out std_logic_vector(1 downto 0);
    ind_dest: out std_logic_vector(1 downto 0)
  );
end component;

```

Figure 102: Interfaccia Unità di Controllo

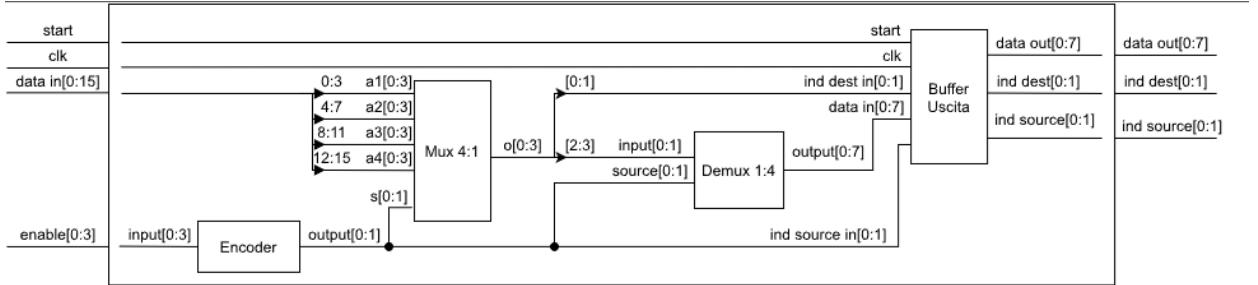


Figure 103: Unità di Controllo

La parte di controllo è realizzata con un approccio strutturale ed ha come componenti: MUX 4:1, DEMUX 1:4, Encoder 4:2 e Buffer; i primi 3 componenti sono stati realizzati utilizzando un approccio Dataflow. La parte di controllo ha il compito di stabilire la priorità tra i nodi. Questa riceve in ingresso il pacchetto per ogni linea, 4 segnali di ENABLE provenienti dai rispettivi 4 nodi sorgente ed il segnale di START. In uscita produce il pacchetto composto da 2 bit dato, 2 bit indirizzo sorgente e 2 bit indirizzo destinazione.

A partire dai segnali di ENABLE, l'encoder produce in uscita la codifica della linea abilitata in ingresso, dando al nodo 0 la priorità massima, mentre al nodo 3 la priorità minima. Una volta trovato l'indirizzo sorgente si abilitano le linee dato relative ad esso tramite il MUX 4:1: i primi due bit

sono relativi all'indirizzo destinazione, mentre i restanti sono di tipo dato. Quest'ultimi attraversano, poi, il DEMUX per avere lo stesso parallelismo dell'Omega network. Il compito del demultiplexer è dunque quello di dare in ingresso all'omega network un vettore di 8 bit: tale vettore presenterà tutti bit pari a 0, fatta eccezione per i 2 bit riferiti al nodo con priorità maggiore tra quelli che hanno richiesto la trasmissione.

I dati, l'indirizzo destinazione e sorgente vengono poi inseriti all'interno di un buffer che li memorizza temporaneamente, ad ogni comunicazione, per poterli inserire all'interno dell'Omega Network.

## MUX 4:1

```
entity mux_4_1 is
    generic( N: integer:= 4);
    Port ( al : STD_LOGIC_VECTOR (0 to N-1) ;
           a2 : STD_LOGIC_VECTOR (0 to N-1) ;
           a3 : STD_LOGIC_VECTOR (0 to N-1) ;
           a4 : STD_LOGIC_VECTOR (0 to N-1) ;
           s : in STD_LOGIC_VECTOR (0 to 1);
           o : out STD_LOGIC_VECTOR(0 to N-1));
end mux_4_1;

architecture Behavioral of mux_4_1 is

begin
    o <= al when s = "00" else
        a2 when s = "01" else
        a3 when s = "10" else
        a4 when s = "11" else
        "----";
end Behavioral;
```

Figure 104: Mux 4:1

## Demux 4:1

```
entity demux_4_1 is
  Generic (N: integer := 2);
  Port ( output: out STD_LOGIC_VECTOR (0 to 4*N-1);
          source : in STD_LOGIC_VECTOR (0 to 1);
          input : in STD_LOGIC_VECTOR(0 to N-1));
end demux_4_1;

architecture Behavioral of demux_4_1 is
signal temp: std_logic_vector(0 to N-1) := (others => '0');
begin
  output <= input & temp & temp & temp when source="00" else
    temp & input & temp & temp when source="01" else
    temp & temp & input & temp when source="10" else
    temp & temp & temp & input when source="11" else
    (others => '0');

end Behavioral;
```

Figure 105: Demux 4:1

## Encoder

```
entity encoder is
  Port (
    input: in std_logic_vector(0 to 3);
    output: out std_logic_vector(0 to 1)
  );
end encoder;

architecture Behavioral of encoder is

begin
  output <= "00" when input(0) = '1' else
    "01" when input(1) = '1' else
    "10" when input(2) = '1' else
    "11" when input(3) = '1' else
    "--";

end Behavioral;
```

Figure 106: Encoder

## Buffer

```
entity buffer_uscita is
  generic (N : integer:=2);
  port(
    clk : in std_logic;
    data_in: in std_logic_vector(0 to 4*N-1);
    ind_source_in: in std_logic_vector(0 to 1);
    ind_dest_in: in std_logic_vector(0 to 1);
    data_out: out std_logic_vector(0 to 4*N-1);
    ind_source: out std_logic_vector(0 to 1);
    ind_dest: out std_logic_vector(0 to 1);
    start: in std_logic
  );
end buffer_uscita;

architecture Behavioral of buffer_uscita is
type stato is (qinit,q1);
signal temp_ind_source,temp_ind_dest: std_logic_vector(0 to 1);
signal data_temp: std_logic_vector(0 to 4*N-1);
signal stato_corrente: stato:=qinit;
begin
  ctrl_state: process(clk)
  begin
    if(clk = '1' and clk'event) then
      case stato_corrente is
        when qinit =>
          if(start = '1') then
            temp_ind_source <= (others => '0');
            temp_ind_dest <= (others => '0');
            data_temp <= (others => '0');
            stato_corrente <= q1;
          else
            stato_corrente <= qinit;
          end if;
        when q1 => temp_ind_source <= ind_source_in;
            temp_ind_dest <= ind_dest_in;
            data_temp <= data_in;
            stato_corrente <= qinit;
        when others => stato_corrente <= qinit;
      end case;
    end if;
  end process;
  data_out <= data_temp;
  ind_source <= temp_ind_source;
  ind_dest <= temp_ind_dest;
end Behavioral;
```

Figure 107: Buffer

Il buffer è stato realizzato con un approccio Behavioral, cioè come una macchina a stati che ad ogni segnale di start effettua il refresh del pacchetto memorizzato:

- qinit → Stato iniziale in cui si aspetta un nuovo segnale di start. Appena arriva tale segnale si effettua il reset del buffer e si passa allo stato q1.
- q1 → Stato in cui avviene l'acquisizione del nuovo pacchetto, ed al colpo di clock successivo si sposta in qinit, mantenendo memorizzato il pacchetto corrente.

L'architettura Structural complessiva dell'unità di controllo è mostrata in figura:

```

architecture Structural of control_unit is
component mux_4_1
generic( N: integer:= N+2);
Port ( a1 : in STD_LOGIC_VECTOR (0 to N-1) ;
      a2 : in STD_LOGIC_VECTOR (0 to N-1) ;
      a3 : in STD_LOGIC_VECTOR (0 to N-1) ;
      a4 : in STD_LOGIC_VECTOR (0 to N-1) ;
      s : in STD_LOGIC_VECTOR (0 to 1);
      o : out STD_LOGIC_VECTOR(0 to N-1));
end component;
component demux_4_1
Generic (N: integer := N);
Port ( output: out STD_LOGIC_VECTOR (0 to 4*N-1);
       source : in STD_LOGIC_VECTOR (0 to 1);
       input : in STD_LOGIC_VECTOR(0 to N-1));
end component;
component encoder is
    Port (
        input: in std_logic_vector(0 to 3);
        output: out std_logic_vector(0 to 1)
    );
end component;
component buffer_uscita is
port(
    clk: in std_logic;
    data_in: in std_logic_vector(0 to 4*N-1);
    ind_source_in: in std_logic_vector(0 to 1);
    ind_dest_in: in std_logic_vector(0 to 1);
    data_out: out std_logic_vector(0 to 4*N-1);
    ind_source: out std_logic_vector(0 to 1);
    ind_dest: out std_logic_vector(0 to 1);
    start: in std_logic
);
end component;
signal source: std_logic_vector(0 to 1);
signal output_mux: std_logic_vector(0 to 3);
signal output_demux: std_logic_vector(0 to 7);
.
.
.

```

```

begin

    mux: mux_4_1 port map(
        a1 => data_in(0 to 3),
        a2 => data_in(4 to 7),
        a3 => data_in(8 to 11),
        a4 => data_in(12 to 15),
        s  => source,
        o  => output_mux
    );
    demux: demux_4_1 port map(
        source => source,
        input => output_mux(2 to 3),
        output => output_demux
    );
    enc: encoder
        Port map(
            input => enable,
            output => source
        );
    buffer_u: buffer_uscita
        port map(
            clk => clk,
            data_in => output_demux,
            ind_source_in => source,
            ind_dest_in => output_mux(0 to 1),
            data_out => data_out,
            ind_source => ind_source,
            ind_dest => ind_dest,
            start => start
        );
end Structural;

```

Figure 108: Architecture Unità di Controllo

## 10.4 Simulazione

Per effettuare la simulazione, nel testbench abbiamo utilizzato un sistema con periodo di clock pari a 10ns. Vengono inserite nel sistema i segnali di ENABLE e DATA\_IN per rappresentare la volontà di comunicazione dei nodi sorgente. Si suppone che il segnale di start si alzi per un solo periodo di clock.

```
signal clk: std_logic;
signal enable: std_logic_vector(0 to 3) := (others => '0');
signal start: std_logic:='0';
signal data_in: std_logic_vector(0 to 15):= (others => '0');
signal data_out: std_logic_vector(0 to 7):= (others => '0');
constant CLK_PERIOD: time := 10ns;

begin
    sist_comp: sistema_complessivo port map(
        clk => clk,
        enable => enable,
        start => start,
        data_in => data_in,
        data_out => data_out
    );
    clk_proc: process
    begin
        clk <= '0';
        wait for CLK_period/2;
        clk <= '1';
        wait for CLK_period/2;
    end process;

    sis_proc:process
    begin
        enable <= "1111";
        data_in(0 to 3) <= "1010" ; --prima linea vuole comunicare con 2 e dato= 2
        data_in(4 to 7) <= "0011";
        data_in(8 to 11) <= "0000";
        data_in(12 to 15) <= "1111";
        start <= '1';
        wait for CLK_period;
        start <= '0';
        wait;
    end process;
end Behavioral;
```

Figure 109: Testbench

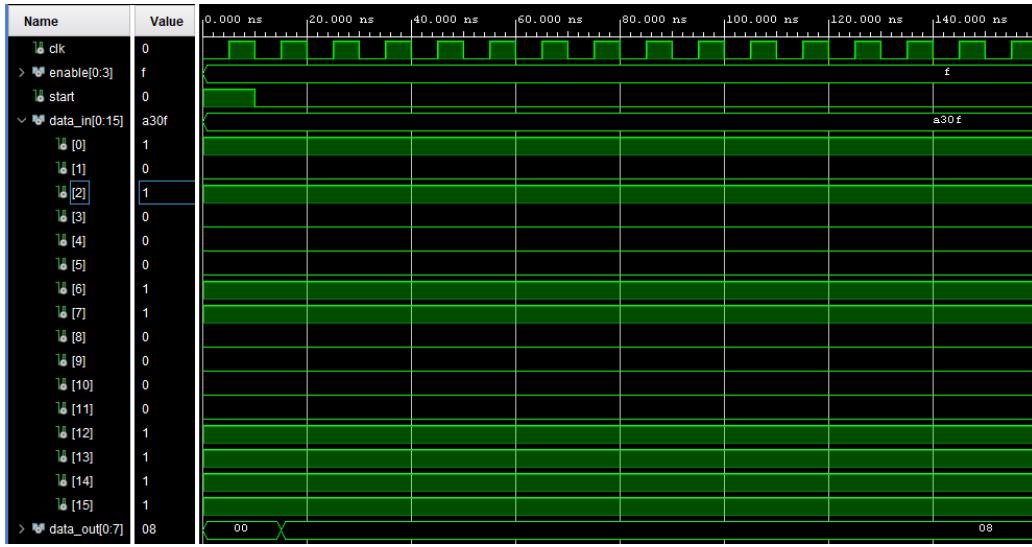


Figure 110: Simulazione

Dalla simulazione si ottiene che ponendo tutti i bit di ENABLE alti, solo la prima linea riesce a comunicare perché è quella a priorità più alta. Questa comunica con la linea 2 di uscita il dato 2. Essendo tutti i blocchi dell'Omega Network combinatori, la presentazione del risultato in simulazione è istantanea ed avviene non appena è caricato il Buffer con il pacchetto. Nella realtà vi è la presenza di ritardi dovuti alle porte combinatorie.

## 10.5 Considerazioni finali

Il sistema è funzionante solo per una comunicazione alla volta, secondo una priorità fissa. In realtà sarebbe possibile effettuare anche più comunicazioni all'interno dell'Omega Network, a patto di gestire le collisioni tra nodi sorgente che vogliono comunicare con una stessa destinazione e tra nodi sorgente che comunicano con diverse destinazioni, ma durante il percorso per raggiungere i nodi destinatari attraversano nodi intermedi comuni.

Una possibile soluzione per poter gestire tali collisioni prevede di complicare il blocco di instradamento con una coda ed un'interfaccia che permettano di memorizzare i dati in arrivo sulle due linee di ingresso. È necessario, inoltre, realizzare un protocollo di handshaking asincrono (aumentando il numero di linee) tra il blocco trasmettitore ed il blocco ricevente per ogni linea d'ingresso, in modo da stabilire quando il blocco trasmettitore è pronto ad inviare e quando il blocco ricevente è pronto ad acquisire su ogni linea dato.

Ad ogni ciclo di trasmissione da uno stadio al successivo, il blocco in presenza di collisioni ritarda, tramite la coda, la trasmissione di uno dei pacchetti ricevuto in base ad una priorità. In questo modo si realizza un'architettura sincrona rispetto al clock e quindi aumenta il ritardo di trasmissione dal nodo sorgente a quello destinazione per via della gestione dell'handshaking.

Inoltre, bisogna gestire le comunicazioni concorrenti in ingresso al sistema switch, in particolare se più nodi inviano pacchetti allo stesso switch è necessario che quest'ultimo li acquisisca evitandone la perdita. Una possibile soluzione sarebbe quella di adottare un sistema di handshaking asincrono tra i vari nodi e lo switch.

# 11 Esercizio 11 - Divisore Restoring

## 11.1 Introduzione

L'obiettivo di questo esercizio è quello di progettare, tramite un approccio strutturale, un divisore che ci permetta di effettuare una divisione tra un dividendo espresso su M bit ed un divisore, espresso su N bit. Il risultato sarà composto da un resto, codificato su al più N bit, ed un quoziente, espresso al più su M-N+1 bit (supponendo che il primo degli  $\frac{m}{n}$  bit non sia nullo). Analizziamo ora il procedimento di divisione, per comprendere quali sono i componenti da utilizzare:

- Passo iniziale: si confrontano gli n bit più significativi del dividendo (dividendo parziale) con gli n bit del divisore: se il divisore è contenuto nel dividendo parziale, allora il quoziente parziale ( $q_i$ ) sarà pari ad 1, 0 altrimenti (Il risultato delle operazioni di divisione parziale può essere al massimo 1, quindi in binario si ha un grande vantaggio). Si effettua quindi la sottrazione tra il dividendo parziale ed il prodotto  $q_i \cdot V$  (dove V rappresenta il divisore), ottenendo così il primo resto parziale.
- Generico passo i: si pone il dividendo parziale  $D_i$  pari al resto parziale  $R_i$  (calcolato all'iterazione i-1) e si confronta con il divisore. A questo punto i passaggi successivi sono analoghi a quelli riportati al passo 1, con l'unica differenza che il resto parziale andrà riportato con uno shift a destra di i posizioni.

La formula generale per calcolare i resti parziali al passo  $i+1$  è dunque:

$$R_{i+1} = R_i - q_i \cdot V \cdot 2^{(-i)}$$

Mantenendo questo procedimento avremmo dunque bisogno di uno shift register con shift variabile, in modo da poter effettuare uno shift diverso ad ogni passo.

Si è tuttavia ricavato un algoritmo alternativo, al fine di effettuare uno shift di una sola posizione ad ogni iterazione: invece di effettuare uno shift del resto parziale  $i+1$  al termine delle operazioni, si effettua (ad ogni iterazione) uno shift a sinistra del dividendo parziale  $R_i$  e, con tale valore che in binario coincide a  $2R_i$ , si calcola  $R_{i+1}$ .

L'algoritmo alternativo è del tutto equivalente a quello standard, con l'unica differenza che richiede uno shift di un'unica posizione ad ogni iterazione, a differenza del precedente, dove lo shift variava in base al passo  $i$ .

Valutiamo ora il numero di registri necessari per effettuare completamente le operazioni: il registro A, in cui immettiamo il dividendo, ad ogni iterazione viene svuotato di 1 bit (a causa degli shift verso sinistra dei dividendi parziali), mentre, il registro Q ( contenente il quoziente) si riempie di 1 bit. Si può utilizzare quindi un unico registro AQ, il quale, al termine delle operazioni conterrà il resto negli  $N$  bit più significativi e nei restanti  $M-N+1$  bit il quoziente.

Utilizzando tale algoritmo però, il registro AQ dovrà presentare la prima cella vuota, in modo da effettuare lo shift anche al primo passo senza perdere alcun bit.

Come visto dalla procedura manuale, l'operazione di divisione richiede una successione di sottrazioni e confronti; l'operazione di comparazione, idealmente, potrebbe essere effettuata mediante un componente comparatore. Tuttavia, anziché definire un nuovo componente, l'operazione di comparazione viene effettuata mediante un sottrattore, valutando il bit più significativo del risultato. Supponiamo infatti di voler effettuare un confronto tra A e B: effettuando l'operazione A-B, se il risultato presenta il bit più significativo, che chiameremo S per comodità, pari ad 1 (ovvero il risultato è negativo in una rappresentazione in complemento a 2), allora  $A < B$ , altrimenti  $A \geq B$ .

Utilizzando questo metodo, possiamo sfruttare il sottrattore per effettuare anche la comparazione, il che porta un ulteriore vantaggio: siccome la comparazione avviene tra  $2R_i$  e V, il risultato ci fornisce anche il valore di  $R_{i+1}$ , nel caso in cui  $2R_i > V$ . Nel caso in cui, invece,  $2R_i < V$  va effettuata un'operazione di ripristino del valore di  $R_{i+1}$ , riportandolo pari a  $R_i$ .

Per gestire tale rispristino ci sono 2 algoritmi possibili:

- Restoring: effettua il ripristino sempre al passo  $i$ -esimo; se il risultato della sottrazione da esito negativo, nello stesso passo si somma nuovamente V al valore di  $R_{i+1}$ , in modo da ottenere nuovamente il valore di  $R_i$ .
- Not restoring: effettua una correzione al passo  $i+1$ ; tramite manipolazioni algebriche si è visto che, se al passo  $i+1$  effettuo una somma al posto di una sottrazione, ottengo lo stesso risultato del restoring.

Dato che entrambe le tecniche prevedono sia l'addizione che la sottrazione, avremo bisogno di un componente che implementa entrambe operazioni. È bene notare che, a differenza dei moltiplicatori, i divisori sono implementati unicamente con un approccio sequenziale, dato che il risultato al passo  $i$ -esimo dipende dal risultato del passo  $i-1$ , non permettendo quindi un'implementazione di tipo parallelo.

Come tutte le macchine complesse, il divisore restoring è composto da una parte operativa ed una di controllo.

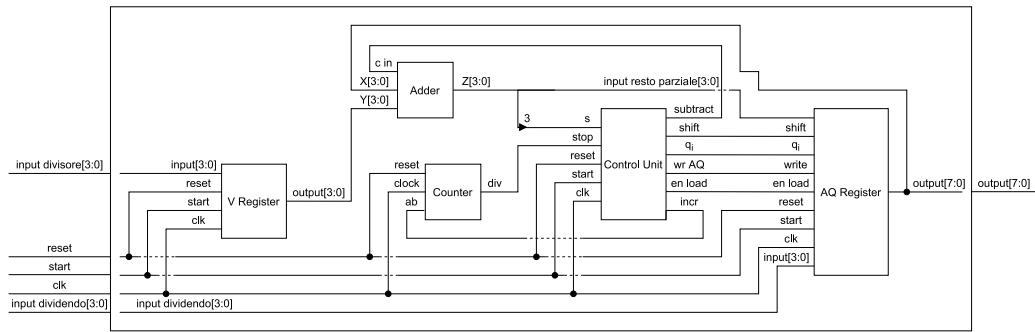


Figure 111: Schema Divisore Es. 11

## 11.2 Unità Operativa

La parte operativa è stata progettata tramite un approccio strutturale, ed è composta da:

- Registro AQ: registro che inizialmente contiene il dividendo e che, al termine delle operazioni, conterrà il quoziente ed il resto.
- Registro V: registro che contiene il divisore.
- Sommatore RippleCarry: sommatore/sottrattore.
- Contatore: necessario ad indicare il termine delle operazioni da svolgere.

## Componente Registro AQ:

```
entity registroAQ is
  generic( N: integer := 4);
  port(
    clk:in std_logic;
    reset: in std_logic;
    start: in std_logic;
    input: in std_logic_vector(N-1 downto 0); --dividendo
    input_resto_parziale: in std_logic_vector(N-1 downto 0);
    shift: in std_logic := '0'; -- shifta a sx di 1
    en_load: in std_logic := '0';
    q_i: in std_logic := '0'; -- valore della cifra q_i
    write: in std_logic;
    output: out std_logic_vector(2*N-1 downto 0)
  );
end registroAQ;
```

Figure 112: Entity Registro AQ

E' stato progettato tramite un approccio comportamentale, dato che ci sono più segnali di controllo da gestire.

Implicitamente, con tale implementazione, è stato definita anche una priorità tra i segnali in ingresso, anche se ha poca rilevanza dato che, tali segnali, non saranno mai alti contemporaneamente (fa eccezione il solo segnale di reset, il quale potrebbe alzarsi in qualsiasi momento poiché proveniente dall'esterno e non dall'unità di controllo).

La dimensione  $i$  tale registro è pari a  $2N$ , dato che sia il resto che il quoziente saranno codificati su  $N$  bit.

```

architecture Behavioral of registroAQ is
signal temp_reg: std_logic_vector(2*N-1 downto 0) := (others => '0');
begin
proc_reg: process(clk)
begin
  if(clk = '1' and clk'event) then
    if(reset = '1') then
      temp_reg <= (others => '0');
    elsif(start = '1') then
      temp_reg(N-1 downto 0) <= input(N-1 downto 0);
    elsif(en_load ='1') then
      temp_reg(0) <= q_i;
    elsif(shift = '1') then
      temp_reg <= temp_reg (2*N-2 downto 0) & '0';
    elsif(write = '1') then
      temp_reg(2*N-1 downto N) <= input_resto_parziale(N-1 downto 0);
    end if;
  end if;
end process;
output <= temp_reg;
end Behavioral;

```

Figure 113: Architecture Registro AQ

## Componente Registro V:

```

entity registroV is
  generic( N: integer:= 4); |
  port(
    clk:in std_logic;
    reset: in std_logic;
    start: in std_logic;
    input: in std_logic_vector(N-1 downto 0);
    output: out std_logic_vector(N-1 downto 0)
  );
end registroV;

```

Figure 114: Entity Registro V

Così come il registroAQ, anche tale registro è stato implementato mediante un approccio di tipo comportamentale.

```

architecture Behavioral of registroV is
signal temp_reg: std_logic_vector(N-1 downto 0) := (others => '0');
begin
proc_reg: process(clk)
begin
    if(clk = '1' and clk'event) then
        if(reset = '1') then
            temp_reg <= (others => '0');
        elsif(start = '1') then
            temp_reg <= input(N-1 downto 0);
        end if;
    end if;
end process;
output <= temp_reg;
end Behavioral;

```

Figure 115: Architecture Registro V

### Componente Contatore:

Tale componente è analogo a quello presentato nell'esercizio 9; pertanto non verrà riportata nuovamente la sua implementazione.

### Componente Adder-Subtracter:

```

entity adder_subtracter is
    Generic(N:integer:=8);
    Port ( X : in STD_LOGIC_VECTOR (N-1 downto 0);
           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 adder_subtracter;

```

Figure 116: Entity Adder-Subtracter

Le operazioni necessarie per la divisione richiedono sia operazioni di somma che di differenza; pertanto, la scelta è ricaduta su un unico componente in grado di effettuare entrambe le operazioni. È stato progettato tramite un approccio strutturale, utilizzando un Ripple carry adder ed un signal che effettua il complemento a 2 di uno degli ingressi.

```

architecture Structural of adder_subtractor is
component ripple_curry_adder Generic (N:integer:=8);
    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;
for all: ripple_curry_adder use entity work.ripple_curry_adder(Structural);

signal complementoy: std_logic_vector(N-1 downto 0);
begin
complemento: for i in 0 to N-1 generate
    complementoy(i) <= Y(i) xor c_in;
end generate;
RA: ripple_curry_adder generic map( N => N) port map( X => X, Y => complementoy, c_in => c_in, c_out => c_out, Z => Z);
end Structural;

```

Figure 117: Architecture Adder-Subtracter

## Component Ripple Carry Adder:

```

entity ripple_curry_adder is
    Generic (N:integer:=8);
    Port ( X : in STD_LOGIC_VECTOR (N-1 downto 0);
           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_curry_adder;

```

Figure 118: Entity Ripple Carry Adder

Nonostante dal punto di vista prestazionale tale componente non sia tra i più efficienti, la scelta è ricaduta comunque su di esso data la sua struttura semplice e ritardi comunque trascurabili per operandi a 4 bit. È stato realizzato mediante interconnessione di componenti full-adder:

```

architecture Structural of ripple_carry_adder is
component FA Port ( a : in STD_LOGIC;
                     b : in STD_LOGIC;
                     c_in : in STD_LOGIC;
                     c_out : out STD_LOGIC;
                     s : out STD_LOGIC);
end component;
for all: FA use entity work.FA(DataFlow);

signal c: std_logic_vector(N-1 downto 1);
begin
FA_first: FA port map( a => X(0), b => Y(0), c_in => c_in, c_out => c(1), s => Z(0));
FA_last: FA port map( a => X(N-1), b => Y(N-1), c_in => c(N-1), c_out => c_out, s => Z(N-1));

middle_adder: for i in 1 to N-2 generate
    FA_middle: FA port map(a => X(i), b=> Y(i), c_in => c(i), c_out => c(i+1), s=> Z(i));
end generate;
end Structural;
```

Figure 119: Architecture Ripple Carry Adder

### Componente Full-Adder:

```

entity FA is
    Port ( a : in STD_LOGIC;
           b : in STD_LOGIC;
           c_in : in STD_LOGIC;
           c_out : out STD_LOGIC;
           s : out STD_LOGIC);
end FA;
```

Figure 120: Entity Full-Adder

E' il componente base che ci permette di effettuare la somma su 2 bit ed è stato progettato mediante approccio Dataflow:

```
architecture DataFlow of FA is

begin
  s <= ((a xor b) xor c_in);
  c_out <= ((a and b) or (c_in and (a xor b)));
end DataFlow;
```

Figure 121: Architecture Full-Adder

### 11.3 Unità di Controllo

Al termine di tale panoramica sui componenti presenti nell'unità operativa, passiamo ora allo studio dell'unità di controllo.

#### Control Unit

```
entity control_unit is
  Port (
    clk, reset, start, s: in STD_LOGIC;
    wr_AQ: out STD_LOGIC:='0';
    incr: out std_logic;      --incrementa contatore
    subtract: out std_logic; --tipo di operazione da effettuare
    shift: out std_logic;    -- shifta a sx di 1
    en_load : out std_logic;
    q_i: out std_logic;      -- valore della cifra q_i da porre nel registro AQ
    stop: in std_logic --indica la fine delle iterazioni
  );
end control_unit;
```

Figure 122: Entity Control Unit

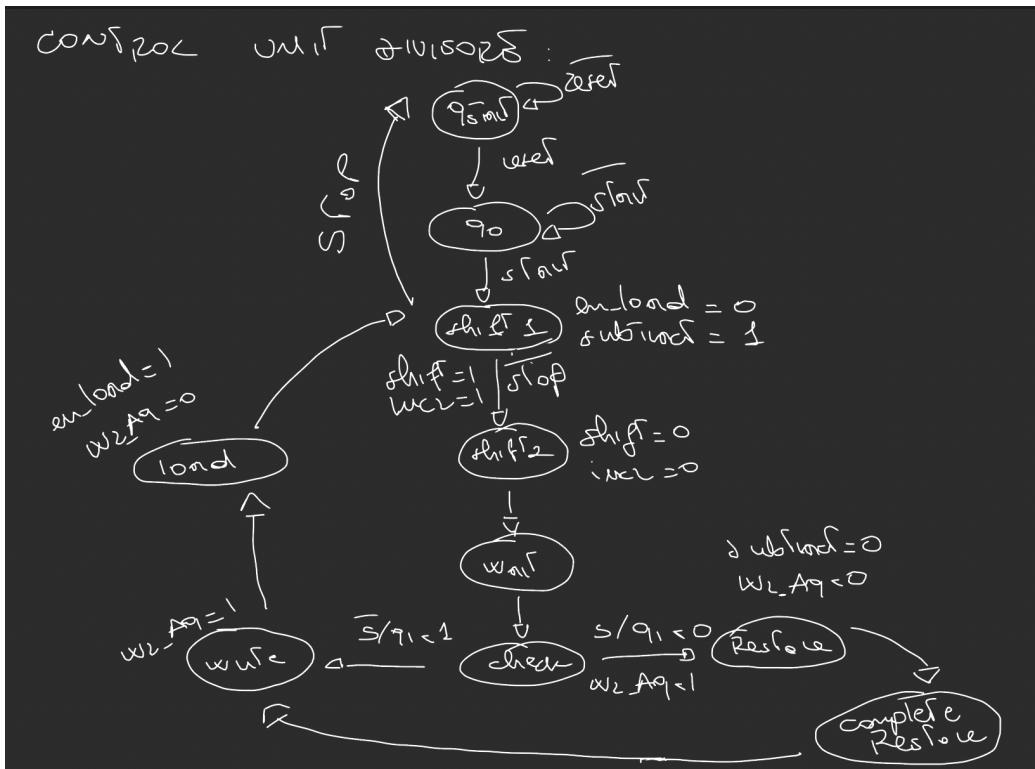


Figure 123: Schema Control Unit

L'unità di controllo è stata realizzata come una macchina a stati:

1.  $q_{start} \rightarrow$  In tale stato si attende il segnale di reset. Quando arriva, si passa in  $q_0$ ;
2.  $q_0 \rightarrow$  In tale stato si attende il segnale di start. Quando questo arriva si passa nello stato  $shift1$ ;
3.  $Shift1 \rightarrow$  A tale stato sono associate le uscite  $en\_load = 0$  e  $subtract = 1$ . Queste servono a modificare le uscite dello stato di  $load$ :  $en\_load$  perché deve essere alto 1 colpo di clock,  $subtract$  perché di base la prima operazione che facciamo svolgere all'adder è la sottrazione per il confronto, poi se necessario viene effettuata l'addizione per il restore.

Si effettua poi un controllo sul valore di stop, corrispondente all'uscita div del contatore: se questa è alta, ovvero abbiamo effettuato tutte le operazioni, non si effettua un nuovo shift ma si ritorna nello stato di start e ci si pone in attesa di un reset. Se invece il valore del div è basso, si passa allo stato di shift2 e si alza il segnale di shift, per abilitare lo scorrimento del registro AQ, e si alza il segnale di abilitazione per il conteggio del contatore;

4. Shift2 → In tale stato semplicemente si abbassano i segnali alzati nella transizione, poiché vogliamo siano alti solo 1 colpo di clock per evitare conteggi o shift spuri. A questo punto si passa allo stato di wait;
5. Wait → In tale stato si attende semplicemente che il sommatore abbia effettuato il confronto. Siccome è una macchina puramente combinatoria e non ha nessun segnale di abilitazione, il sommatore darà sempre un valore in uscita; sta a noi aspettare il tempo necessario, per far sì che il risultato sia relativo agli operandi immessi e non ai precedenti. Il sommatore ha il suo ritardo dovuto alla propagazione, per questo è necessario tale stato. Con operandi a 4 bit, questo stato intermedio è sufficiente a garantire che il risultato in uscita sia quello corretto; tuttavia, all'aumentare del numero di bit, potrebbe essere necessario aggiungere un nuovo stato di attesa oppure abbassare la frequenza del clock. Si passa poi allo stato di check;
6. Check → In tale stato si effettua il controllo sul confronto avvenuto tramite sottrazione:
  - Se il bit più significativo è basso, ovvero il risultato è positivo, si passa allo stato di write e non è necessario il

restore. Si pone, inoltre, il valore del quoziente parziale pari a 1;

- Se il bit più significativo è alto, ovvero il risultato è negativo, si passa allo stato di restore. Si pone, inoltre, il valore del quoziente parziale pari a 0 e si alza il segnale di wr\_aq, al fine di scrivere il valore in uscita dal sommatore nel registro aq per effettuare il restore.
7. Restore → In tale stato si abbassa il valore di subtract, poiché è necessario fare la somma per effettuare il restore, e si abbassa inoltre il segnale di wr\_aq, per evitare sovrascritture indesiderate. Si passa poi allo stato di complete restore;
  8. Complete Restore → Tale stato ha sostanzialmente lo stesso scopo dello stato di wait, ovvero dare il tempo all'adder di effettuare il restore e prendere il risultato corretto. Si passa poi allo stato di write;
  9. Write → Tale stato scrive l'uscita Z dell'adder nella parte del registro AQ dedicata ad A, che ovviamente sarà diversa in base a se viene effettuato il restore o meno: nel primo caso il valore scritto coinciderà con il precedente, altrimenti sarà pari al valore della differenza calcolata. Si alza quindi il segnale di wr\_aq e si passa nello stato di load;
  10. Load → In tale stato si abbassa il segnale di wr\_aq e si alza quello di en\_load, per caricare nella parte dedicata a Q il bit del risultato parziale. Si ritorna poi nello stato di shift1.

```

architecture Behavioral of control_unit is
type stato is (q_start,q0,shift1, shift2, q_wait, check, restore, complete_restore, write,load);
signal stato_corrente : stato := q_start;

begin
process(clk)
begin
  if(clk = '1' and clk'event) then
    case stato_corrente is
      when q_start => if(reset = '1') then
          stato_corrente <= q0;
        else
          stato_corrente <= q_start;
        end if;
      when q0 => if(start = '1') then
          stato_corrente <= shift1;
        else
          stato_corrente <= q0;
        end if;

      when shift1 => en_load <= '0';
                      subtract <= '1';
                      if(stop = '1') then
                        stato_corrente <= q_start;
                      else
                        shift <= '1';
                        incr <='1';
                        stato_corrente <= shift2;
                      end if;

      when shift2 => shift <= '0';
                      incr <= '0';
                      stato_corrente <= q_wait;

      when q_wait => stato_corrente <= check;

      when check => if(s = '0') then
          q_i <= '1';
          stato_corrente <= write;
        else
          q_i <= '0';
          wr_AQ <= '1';
          stato_corrente <= restore;
        end if;
    end case;
  end if;
end process;
end;

```

```

when restore => wr_AQ <= '0';
                  subtract <= '0';
                  stato_corrente <= complete_restore;

when complete_restore => stato_corrente <= write;

when write => wr_AQ <= '1';
                  stato_corrente <= load;

when load => wr_AQ <= '0';
                  en_load <= '1';
                  stato_corrente <= shift1;

when others => stato_corrente <= q_start;
end case;
end if;
end process;

end Behavioral;

```

Figure 124: Architecture Control Unit

## 11.4 Simulazione e Sintesi

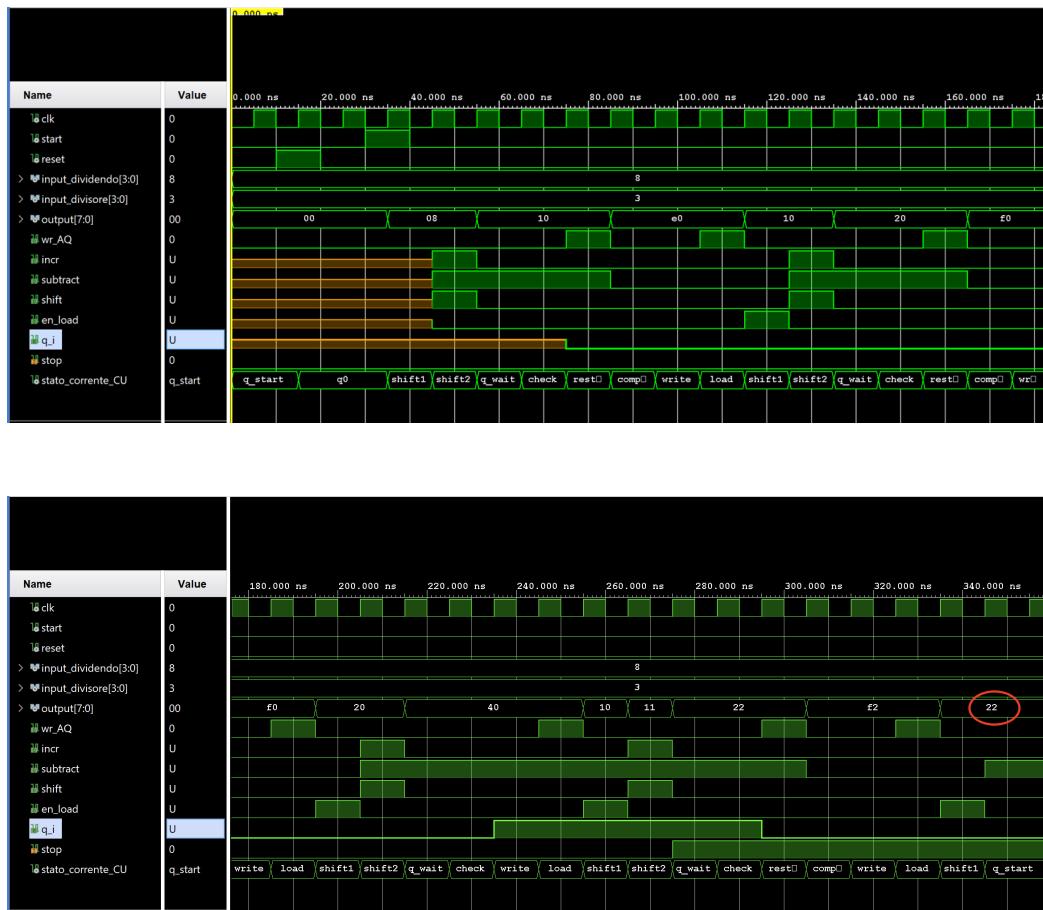


Figure 125: Simulazione Sistema Es. 11

### Implementazione su fpga:

Per l'implementazione su scheda sono stati introdotti 2 debouncer per i segnali di start e reset ed il componente display, necessario per la visualizzazione dei risultati.

Il problema principale del passaggio da simulazione ad fpga, per un progetto di questo tipo, riguarda la frequenza del clock: quest'ultimo potrebbe essere troppo veloce e non dare il tempo

all'adder di svolgere le operazioni e di fornire il risultato corretto. Nel nostro caso, dato che gli operandi sono su soli 4 bit, non è stato necessario introdurre un divisore di frequenza; all'aumentare però della dimensione degli operandi, potrebbe tuttavia essere necessario introdurlo.

```

entity sistema_complessivo is
    generic( N: integer := 4);
    Port (
        clk : in std_logic;
        start: in std_logic;
        reset: in std_logic;
        switch: in std_logic_vector(2*N-1 downto 0); --op1,op2
        anodi: out std_logic_vector(7 downto 0);
        catodi: out std_logic_vector(7 downto 0)
    );
end sistema_complessivo;

architecture Structural of sistema_complessivo is

component display_seven_segments
    port ( CLK : in STD_LOGIC;
            RST : in STD_LOGIC;
            VALUE : in STD_LOGIC_VECTOR (31 downto 0);
            ENABLE : in STD_LOGIC_VECTOR (7 downto 0);
            DOTS : in STD_LOGIC_VECTOR (7 downto 0);
            ANODES : out STD_LOGIC_VECTOR (7 downto 0);
            CATHODES : out STD_LOGIC_VECTOR (7 downto 0));
end component;

component divisore
    generic( N: integer := N);
    port(
        clk:in std_logic;
        start: in std_logic;
        reset: in std_logic;
        input_dividendo: in std_logic_vector(N-1 downto 0);
        input_divisore: in std_logic_vector(N-1 downto 0);
        output: out std_logic_vector(2*N-1 downto 0) --op1,op2
    );
end component;

```

```

component debouncer
    Port ( button : in STD_LOGIC;
            clock : in STD_LOGIC;
            cleared_button : out STD_LOGIC);
end component;

signal start_cleared,reset_cleared: STD_logic := '0';
signal enable: STD_LOGIC_VECTOR (7 downto 0):= "11111111";
signal output: STD_logic_vector(31 downto 0);
signal output_divisore: STD_logic_vector(2*N-1 downto 0);
begin

deb_reset: debouncer port map( button=> reset, clock=> clk, cleared_button=> reset_cleared);
deb_start: debouncer port map( button=> start, clock=> clk, cleared_button=> start_cleared);

divisore_1: divisore
generic map ( N => N)
port map(
    clk => clk,
    start => start_cleared,
    reset => reset_cleared,
    input_dividendo => switch(2*N-1 downto N),
    input_divisore => switch(N-1 downto 0),
    output => output_divisore
);

```

Figure 126: Sistema Complessivo Es. 11

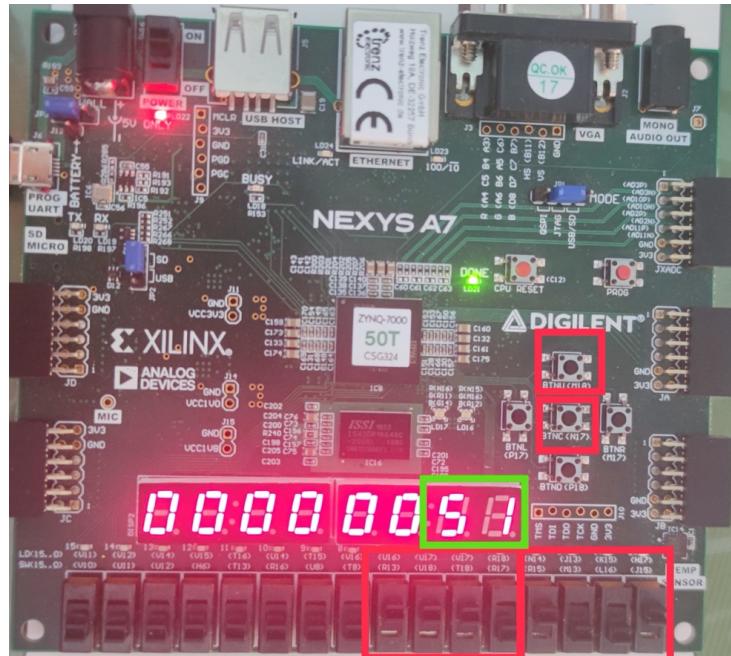


Figure 127: FPGA Es. 11

## **12 Esercizio 12 - Interfaccia VGA**

## List of Figures

1	Mux 4:1 . . . . .	3
2	Mux 4:1 Dataflow . . . . .	3
3	Mux 16:1 . . . . .	4
4	Mux 16:1 Dataflow . . . . .	5
5	Mux 16:1 Testbench . . . . .	6
6	Mux 16:1 Simulazione . . . . .	7
7	Demux 1:4 Dataflow . . . . .	8
8	Rete 16:1 . . . . .	8
9	Rete 16:4 Structural . . . . .	9
10	Rete 16:4 Testbench . . . . .	10
11	Rete 16:4 Simulazione . . . . .	10
12	Costraints . . . . .	11
13	Rete 16:4 Analysis . . . . .	11
14	Codice Arbitro . . . . .	14
15	Codice Encoder 10:4 . . . . .	15
16	Codice Encoder Complessivo . . . . .	16
17	Display Segment . . . . .	17
18	Codice Display Manager . . . . .	18
19	Codice Sistema Completo . . . . .	19
20	Codice Testbench . . . . .	20
21	Simulazione Encoder . . . . .	21
22	Codice Sistema Pt1 . . . . .	23
23	Codice Sistema Pt2 . . . . .	25
24	Debouncer . . . . .	26
25	Codice filtro modo . . . . .	27
26	Codice gestore modo . . . . .	28
27	Codice riconoscitore . . . . .	30
28	Simulazione riconoscitore . . . . .	30
29	Entity Shift Bidirezionale . . . . .	31

30	Architecture Shift Register Bidirezionale . . . . .	32
31	Entity Flip Flop D . . . . .	33
32	Architecture Flip Flop D . . . . .	34
33	Mux 2:1 . . . . .	35
34	Generate FFD . . . . .	36
35	Progetto Completo Esercizio 1 . . . . .	37
36	Simulazione Shift Register . . . . .	38
37	Entity Contatore . . . . .	39
38	Architecture Contatore . . . . .	39
39	Entity Sistema Contatori . . . . .	40
40	Struct Sistema Contatori . . . . .	41
41	Wait Block . . . . .	42
42	Gestore set . . . . .	43
43	Architecture Gestore set . . . . .	44
44	Convertitore per display . . . . .	46
45	Entity Gestore modo . . . . .	46
46	Architecture Gestore modo . . . . .	48
47	Entity Memoria . . . . .	49
48	Architecture Memoria . . . . .	50
49	Schema Completo Esercizio 5 . . . . .	50
50	Sistema Complessivo . . . . .	52
51	Tester . . . . .	52
52	Gestore Read . . . . .	53
53	ROM . . . . .	54
54	Delay Block . . . . .	55
55	Memoria Esterna . . . . .	56
56	Gestore Led . . . . .	57
57	Simulazione Es.6 . . . . .	58
58	Sintesi Es.6 . . . . .	59
59	Schema Sistema Completo Es.7 . . . . .	62

60	Schema Sistema A Es.7 . . . . .	63
61	Entity Contatore . . . . .	64
62	Architecture Contatore . . . . .	64
63	Architecture ROM . . . . .	65
64	Entity Interfaccia A Es.7 . . . . .	66
65	Architecture Interfaccia A Es.7 . . . . .	68
66	Schema Sistema B Es.7 . . . . .	69
67	Entity Control Unit . . . . .	70
68	Entity Control Unit . . . . .	72
69	Architecture Memoria B . . . . .	73
70	Interfaccia B . . . . .	74
71	Architecture Interfaccia B Es.7 . . . . .	76
72	Simulazione Es.7 . . . . .	77
73	Memoria IJVM . . . . .	79
74	Istruzioni IJVM . . . . .	79
75	Processore MIC-1 . . . . .	80
76	Datapath Processore . . . . .	81
77	Tempificazione Memoria . . . . .	82
78	Formato Microistruzione . . . . .	84
79	Timing Clock . . . . .	85
80	Simulazione IADD . . . . .	90
81	Datapath esecuzione IADD . . . . .	91
82	Simulazione ISTORE . . . . .	94
83	Datapath esecuzione ISTORE . . . . .	95
84	Simulazione IADD Modificata . . . . .	96
85	Codice Sistema A . . . . .	100
86	Codice Sistema B . . . . .	101
87	Sintesi su FPGA Es. 9.1 . . . . .	102
88	Schema Sistema A Es. 9.2 . . . . .	104
89	Entity Contatore . . . . .	105

90	Architecture Contatore . . . . .	105
91	Entity Control Unit A Es. 9.2 . . . . .	106
92	Architecture Control Unit A Es. 9.2 . . . . .	109
93	Schema Sistema B Es. 9.2 . . . . .	111
94	Entity Control Unit B Es. 9.2 . . . . .	112
95	Architecture Control Unit B Es.9.2 . . . . .	114
96	Sintesi su FPGA Es. 9.2 . . . . .	115
97	Schema blocco instradamento . . . . .	117
98	Architecture blocco instradamento . . . . .	117
99	Entity Unità Operativa . . . . .	118
100	Unità Operativa . . . . .	118
101	Architecture Unità Operativa . . . . .	119
102	Interfaccia Unità di Controllo . . . . .	120
103	Unità di Controllo . . . . .	120
104	Mux 4:1 . . . . .	121
105	Demux 4:1 . . . . .	122
106	Encoder . . . . .	122
107	Buffer . . . . .	123
108	Architecture Unità di Controllo . . . . .	125
109	Testbench . . . . .	126
110	Simulazione . . . . .	127
111	Schema Divisore Es. 11 . . . . .	131
112	Entity Registro AQ . . . . .	132
113	Architecture Registro AQ . . . . .	133
114	Entity Registro V . . . . .	133
115	Architecture Registro V . . . . .	134
116	Entity Adder-Subtracter . . . . .	134
117	Architecture Adder-Subtracter . . . . .	135
118	Entity Ripple Carry Adder . . . . .	135
119	Architecture Ripple Carry Adder . . . . .	136

120	Entity Full-Adder . . . . .	136
121	Architecture Full-Adder . . . . .	137
122	Entity Control Unit . . . . .	137
123	Schema Control Unit . . . . .	138
124	Architecture Control Unit . . . . .	142
125	Simulazione Sistema Es. 11 . . . . .	143
126	Sistema Complessivo Es. 11 . . . . .	145
127	FPGA Es. 11 . . . . .	145