

**Corso di Laurea
in
Ingegneria Informatica**

Architettura dei Sistemi di Elaborazione Elaborato d'esame

prof. Nicola Mazzocca

Gruppo - 40

M63000989	Fabio d'Andrea
M63000986	Guido Di Chiara
M63001073	Domenico Fordellone
M63001040	Antimo Iannucci
M63001026	Dario Daniele



Università degli Studi di Napoli "Federico II"

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

Anno Accademico 2019/2020
Primo Semestre

Indice

1 Esercizio 1	4
1.1 Traccia	4
1.2 Introduzione	4
1.3 Soluzione	4
1.4 Schematici	8
1.5 Simulazione	8
2 Esercizio 2	11
2.1 Traccia	11
2.2 Introduzione	11
2.3 Soluzione	12
2.4 Schematici	15
2.5 Sintesi	16
2.6 Simulazione	17
3 Esercizio 3	19
3.1 Traccia	19
3.2 Introduzione	19
3.3 Flip-Flop D Edge-Triggered Discesa Sami	19
3.3.1 Soluzione	19
3.3.2 Schematici	25
3.3.3 Simulazione	25
3.4 Flip-Flop D Edge-Triggered Discesa Mazzeo	27
3.4.1 Soluzione	27

3.4.2	Schematici	29
3.4.3	Simulazione	29
3.5	Flip-Flop D Master-Slave non ideale	30
3.5.1	Soluzione	30
3.5.2	Schematici	33
3.5.3	Simulazione	33
3.6	Flip-Flop D MS ideale	35
3.6.1	Soluzione	35
3.6.2	Schematici	37
3.6.3	Simulazione	37
4	Esercizio 4	39
4.1	Traccia	39
4.2	Introduzione	39
4.3	Soluzione	39
4.4	Schematici	43
4.5	Sintesi	44
4.6	Simulazione	49
5	Esercizio 5	52
5.1	Traccia	52
5.2	Introduzione	52
5.3	Soluzione	52
5.4	Schematici	55
5.5	Sintesi	56
5.6	Simulazione	61
5.7	High Level Synthesis	63
5.7.1	Registro Serie-Serie	63
5.7.2	Registro Serie-Serie Circolare	65
5.7.3	Registro Serie-Parallelo	67
5.7.4	Registro Parallelo-Parallelo	68

5.7.5 Registro Parallelo-Serie	70
6 Esercizio 6	72
6.1 Traccia	72
6.2 Introduzione	72
6.3 Contatore Seriale	72
6.3.1 Soluzione	72
6.3.2 Schematici	74
6.3.3 Sintesi	74
6.3.4 Simulazione	82
6.4 Contatore Parallelo	83
6.4.1 Soluzione	83
6.4.2 Schematici	84
6.4.3 Simulazione	84
7 Esercizio 7	86
7.1 Traccia	86
7.2 Introduzione	86
7.3 Soluzione	86
7.4 Schematici	87
7.5 Sintesi	87
7.6 Simulazione	88
8 Esercizio 8	90
8.1 Traccia	90
8.2 Introduzione	90
8.3 Soluzione	90
8.4 Schematici	94
8.5 Sintesi	95
8.6 Simulazione	104

9 Esercizio 9	106
9.1 Traccia	106
9.2 Introduzione	106
9.3 Soluzione	107
9.4 Schematici	111
9.5 Sintesi	112
9.6 Simulazione	117
10 Esercizio 10	119
10.1 Traccia	119
10.2 Introduzione	119
10.3 Soluzione	120
10.3.1 Unità Operativa	121
10.3.2 Unità Di Controllo in logica cablata	128
10.3.3 Unità Di Controllo in logica microprogrammata	131
10.4 Schematici	134
10.5 Sintesi	134
10.5.1 Simulazione	143
11 Esercizio 11	147
11.1 Traccia	147
11.2 Introduzione	147
11.3 UART_TAPPO	148
11.3.1 Soluzione	148
11.3.2 Schematici	149
11.3.3 Sintesi	149
11.3.4 Simulazione	162
11.4 2_UART	164
11.4.1 Soluzione	164
11.4.2 Schematici	165
11.4.3 Sintesi	165

11.4.4 Simulazione	170
11.5 UART_PC	172
11.5.1 Soluzione	172
11.5.2 Schematici	174
11.5.3 Sintesi	175
11.5.4 Simulazione	179
12 Esercizio 12	182
12.1 Traccia	182
12.2 Introduzione	182
12.3 Soluzione	182
12.3.1 Esercizio A	184
12.3.2 Esercizio B	194
12.3.3 Esercizio C	199
13 Esercizio 13	206
13.1 Traccia	206
13.2 Introduzione	207
13.3 Soluzione	207
13.3.1 Unità Operativa	210
13.3.2 Unità di Controllo	219
13.4 Schematici	221
13.5 Sintesi	222
13.6 Simulazione	227

Introduzione

Contestualizzazione dell'elaborato

Nel seguente elaborato si vuole presentare il lavoro svolto per l'esame di *Architettura dei Sistemi di Elaborazione* (ASE). L'obiettivo di questo capitolo è quello di contestualizzare gli esercizi proposti e descrivere l'approccio utilizzato nella risoluzione degli stessi.

Nei Capitoli 1 e 7 sono state realizzate le prime **macchine combinatorie**. Nello specifico un multiplexer 8:1 ed un arbitro 2 su 3. Per realizzare il **multiplexer** è stato utilizzato un approccio di tipo strutturale, seguendo il principio del *divide et impera*. In un primo momento sono stati quindi implementati componenti più semplici e successivamente è stato ottenuto un componente più complesso per composizione. Per l'**arbitro 2 su 3** è stato invece sufficiente implementare la funzione combinatoria ingresso-uscita del componente. Essendo richiesta la sintesi su FPGA del componente realizzato, in tale esercizio si è avuto il primo contatto con la board di sviluppo *Digilent Basys*. È stata quindi necessario programmare l'FPGA e connettere opportunamente le periferiche della board.

Nel Capitolo 2, è stato approfondito lo studio relativo alle periferiche presenti sulla board. Nello specifico è stato realizzato un componente dedicato alla gestione del **display a 7 segmenti**. A partire da quest'ultimo sono stati derivati ulteriori componenti presenti in diversi esercizi per visualizzare le uscite sul display.

Nei Capitoli 3, 5, 6 e 8 sono state realizzate le principali macchine sequenziali. Nello specifico flip-flop, registri e contatori. La realizzazione dei **flip-flop** ha previsto uno studio preliminare delle diverse possibili implementazioni degli stessi (Mazzeo e Sami). Chiarite le principali differenze tra le soluzioni proposte si è proseguito quindi con l'implementazione dei componenti richiesti. Acquisite le conoscenze sui principali elementi di memorizzazione, si è poi passati allo sviluppo di elementi di memoria più complessi, i **registri**. Anche in questo caso è stato utilizzato un approccio strutturale. Inoltre, sono stati utilizzati tali argomenti per approfondire lo studio di strumenti di *High Level Synthesis*, in particolare Simulink. Successivamente è stato portato avanti lo studio di un'ulteriore macchina sequenziale fondamentale, il **contatore**. Nello specifico, sono state realizzate due versioni del contatore, ovvero seriale e parallelo. Infine è stato approfondito un possibile utilizzo dei contatori, rappresentato dall'**orologio**. Si è visto quindi come, connettendo opportunamente più contatori e gestendone la relativa tempificazione, sia possibile realizzare un orologio.

Nei Capitoli 4, 9 e 10 sono state trattate alcune delle macchine aritmetiche principali. Per completezza, sono stati realizzati un sommatore, un moltiplicatore ed un divisore. I primi due sono componenti puramente combinatori, l'ultimo è invece una macchina se-

quenziale. Il sommatore è un **Ripple Carry Adder** (RCA), ottenuto per composizione di dispositivi più semplici. Il componente elementare del sommatore è il full-adder. Il moltiplicatore è un **Moltiplicatore a celle MAC**, realizzato ancora una volta compонendo dispositivi elementari, rappresentati dalle singole celle MAC. Infine, il divisore è un **Divisore non-Restoring**. Tale componente si distingue dai precedenti essendo una macchina sequenziale. A causa della sua complessità, è stata necessaria un'attenta fase di progettazione, che ha previsto la suddivisione dell'architettura in parte operativa e parte di controllo.

Nel Capitolo 11 è stato affrontato lo studio di una **periferica UART**, la cui implementazione è fornita dalla Digilent. In particolare, l'esercizio ha previsto l'utilizzo della periferica secondo diverse configurazioni.

Nel Capitolo 13 è invece riportato uno degli esercizi a scelta proposti. Nello specifico, si è scelto di realizzare una **rete neurale**. In questo modo è stato possibile approfondire come una rete neurale venga effettivamente implementata a livello hardware.

Nel Capitolo 12 è stato infine analizzato il funzionamento di un **processore** operante secondo il modello IJVM. In questo modo è stato possibile approfondire i concetti di istruzione, microprocedura e microistruzione.

Si consiglia di leggere l'elaborato nell'ordine con cui sono stati presentati gli esercizi svolti.

Struttura dei capitoli

Per ogni esercizio svolto si è cercato di utilizzare una struttura ben definita, presentando nell'ordine le seguenti sezioni:

1. *Introduzione*, in cui è brevemente descritto l'approccio utilizzato nella risoluzione dell'esercizio.
2. *Soluzione*, in cui viene esposta la soluzione adottata, giustificandone le scelte progettuali.
3. *Schematici*, in cui viene illustrata l'architettura dei componenti realizzati. Laddove possibile, gli schematici sono stati generati automaticamente attraverso l'ambiente di sviluppo *Xilinx ISE*.
4. *Sintesi* (opzionale), in cui è presentato tutto ciò che è inherente alla sintesi sulla board del dispositivo realizzato, dove richiesto.
5. *Simulazione*, contenente i testbench realizzati per verificare il corretto funzionamento dei componenti in simulazione. La simulazione è stata frequentemente utilizzata come metodo di *debugging*.

Scelte progettuali

È bene sottolineare che le diverse scelte di progettazione, specie nella stesura degli esercizi più complessi, sono state influenzate dalla board di sviluppo a disposizione e dalle prestazioni desiderate per i componenti sviluppati.

Inoltre, dove possibile, si è cercato di rendere l'architettura dei componenti realizzati più generica possibile, attraverso l'utilizzo del costrutto VHDL generic e di for spaziali.

Capitolo 1

Esercizio 1

1.1 Traccia

Progettare ed implementare in VHDL un multiplexer 8:1 indirizzabile utilizzando una descrizione di tipo structural che componga opportunamente multiplexer più piccoli.

Nota: il progetto deve fare uso di almeno un multiplexer 4:1, progettato con una tecnica a scelta dello studente.

1.2 Introduzione

Per la realizzazione di un multiplexer 8:1 è stato seguito un approccio di tipo strutturale, basato sul principio *divide et impera*. Il primo passo ha previsto quindi lo sviluppo del componente elementare multiplexer 2:1. Successivamente sono stati realizzati multiplexer di maggiori dimensioni a partire da quello elementare.

1.3 Soluzione

È stato realizzato il multiplexer 8:1 indirizzabile utilizzando il livello di astrazione **structural**. Si è scelto di utilizzare come componente di base un multiplexer 2:1 descritto attraverso il livello di astrazione **data-flow**, essendo il livello di astrazione più appropriato per la descrizione di una macchina combinatoria.

L'architettura del multiplexer 8:1 è sviluppata su due livelli, come riportato in Figura 1.1. Il primo livello è costituito da 4 multiplexer 2:1, il secondo è costituito da 1 multiplexer 4:1, realizzato a sua volta per composizione di multiplexer 2:1 elementari. Di seguito è riportata in Tabella 1.1 la tabella di verità del multiplexer 2:1:

Ingressi			Uscite
S	X0	X1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabella 1.1: Tabella di verità Multiplexer 2:1

File: MUX_2_1_Nbit.vhd

Il componente multiplexer 2:1 è stato implementato in modo da rendere il numero di ingressi costante e pari a 2. Al contrario, attraverso il costrutto generic, è possibile variare il numero di bit degli ingressi (e dell'uscita), modificando il valore di N nel momento in cui il componente viene istanziato. Nel caso specifico è stato posto N pari ad 1.

Codice Multiplexer 2:1

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

entity MUX_2_1_Nbit is
    generic( N : integer);
    port( X0 : in STD_LOGIC_VECTOR(N-1 downto 0);
          X1 : in STD_LOGIC_VECTOR(N-1 downto 0);
          S   : in STD_LOGIC;
          Y   : out STD_LOGIC_VECTOR(N-1 downto 0)
        );
end MUX_2_1_Nbit;

architecture dataflow of MUX_2_1_Nbit is
begin
    with S select
        Y    <=  X0 when '0',
                  X1 when '1',
                  (others => '-') when others;
end dataflow;
```

File: MUX_4_1_Nbit.vhd

Codice Multiplexer 4:1

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

entity MUX_4_1_Nbit is
    generic( N : integer);
    port( X0 : in STD_LOGIC_VECTOR(N-1 downto 0);
          X1 : in STD_LOGIC_VECTOR(N-1 downto 0);
          X2 : in STD_LOGIC_VECTOR(N-1 downto 0);
          X3 : in STD_LOGIC_VECTOR(N-1 downto 0);
          S   : in STD_LOGIC_VECTOR(1 downto 0);
          Y   : out STD_LOGIC_VECTOR(N-1 downto 0)
        );
end MUX_4_1_Nbit;
```

```

end MUX_4_1_Nbit;

architecture structural of MUX_4_1_Nbit is

COMPONENT MUX_2_1_Nbit
generic( N : integer);
PORT (
    X0 : IN STD_LOGIC_VECTOR(N-1 downto 0);
    X1 : IN STD_LOGIC_VECTOR(N-1 downto 0);
    S : IN STD_LOGIC;
    Y : OUT STD_LOGIC_VECTOR(N-1 downto 0)
);
END COMPONENT;

signal Y_TEMP_0 : STD_LOGIC_VECTOR(N-1 downto 0);
signal Y_TEMP_1 : STD_LOGIC_VECTOR(N-1 downto 0);

begin
    Inst_MUX_2_1_Nbit_0: MUX_2_1_Nbit
    GENERIC MAP (N=> N)
    PORT MAP(
        X0 => X0,
        X1 => X1,
        S => S(0),
        Y => Y_TEMP_0
    );
    Inst_MUX_2_1_Nbit_1: MUX_2_1_Nbit
    GENERIC MAP (N=> N)
    PORT MAP(
        X0 => X2,
        X1 => X3,
        S => S(0),
        Y => Y_TEMP_1
    );
    Inst_MUX_2_1_Nbit_2: MUX_2_1_Nbit
    GENERIC MAP (N=> N)
    PORT MAP(
        X0 => Y_TEMP_0,
        X1 => Y_TEMP_1,
        S => S(1),
        Y => Y
    );
end structural;

```

File: MUX_8_1_Nbit.vhd

Per rendere il codice più compatto si è deciso di istanziare i 4 multiplexer 2:1 del primo livello architettonico mediante il costrutto iterativo `for`.

I multiplexer sono stati quindi opportunamente connessi, assegnando ai componenti del primo livello il bit di selezione meno significativo e a quello del secondo livello i due bit di selezione più significativi. Le uscite dei multiplexer del primo livello sono poste in ingresso a quello del secondo.

Codice Multiplexer 8:1

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MUX_8_1_Nbit is
    generic( N : integer := 1);
    port(
        X0 : in STD_LOGIC_VECTOR(N-1 downto 0);
        X1 : in STD_LOGIC_VECTOR(N-1 downto 0);
        X2 : in STD_LOGIC_VECTOR(N-1 downto 0);
        X3 : in STD_LOGIC_VECTOR(N-1 downto 0);
        X4 : in STD_LOGIC_VECTOR(N-1 downto 0);
        X5 : in STD_LOGIC_VECTOR(N-1 downto 0);
        X6 : in STD_LOGIC_VECTOR(N-1 downto 0);
        X7 : in STD_LOGIC_VECTOR(N-1 downto 0);
        S : in STD_LOGIC_VECTOR(2 downto 0);
        Y : out STD_LOGIC_VECTOR(N-1 downto 0)
    );

```

```

end MUX_8_1_Nbit;

architecture structural of MUX_8_1_Nbit is

type ARRAY_TEMP_OUT is array(3 downto 0) of STD_LOGIC_VECTOR(N-1 downto 0);
type ARRAY_TEMP_IN is array(7 downto 0) of STD_LOGIC_VECTOR(N-1 downto 0);

COMPONENT MUX_2_1_Nbit
GENERIC (N : integer);
PORT(
    X0 : IN STD_LOGIC_VECTOR(N-1 downto 0);
    X1 : IN STD_LOGIC_VECTOR(N-1 downto 0);
    S : IN STD_LOGIC;
    Y : OUT STD_LOGIC_VECTOR(N-1 downto 0)
);
END COMPONENT;

COMPONENT MUX_4_1_Nbit
GENERIC (N : integer);
PORT(
    X0 : IN STD_LOGIC_VECTOR(N-1 downto 0);
    X1 : IN STD_LOGIC_VECTOR(N-1 downto 0);
    X2 : IN STD_LOGIC_VECTOR(N-1 downto 0);
    X3 : IN STD_LOGIC_VECTOR(N-1 downto 0);
    S : IN std_logic_vector(1 downto 0);
    Y : OUT STD_LOGIC_VECTOR(N-1 downto 0)
);
END COMPONENT;

signal Y_TEMP : ARRAY_TEMP_OUT;
signal X_TEMP : ARRAY_TEMP_IN;

begin

X_TEMP(0) <= X0;
X_TEMP(1) <= X1;
X_TEMP(2) <= X2;
X_TEMP(3) <= X3;
X_TEMP(4) <= X4;
X_TEMP(5) <= X5;
X_TEMP(6) <= X6;
X_TEMP(7) <= X7;

mux_all: for i in 0 to 3 generate
    mux_i: MUX_2_1_Nbit
        generic map (N => N)
        port map(
            X0 => X_TEMP(2*i),
            X1 => X_TEMP(2*i+1),
            S => S(0),
            Y => Y_TEMP(i)
        );
    end generate;

Inst_MUX_4_1_Nbit: MUX_4_1_Nbit
    GENERIC MAP (N=> N)
    PORT MAP(
        X0 => Y_TEMP(0),
        X1 => Y_TEMP(1),
        X2 => Y_TEMP(2),
        X3 => Y_TEMP(3),
        S => S(2 downto 1),
        Y => Y
    );
end structural;

```

1.4 Schematici

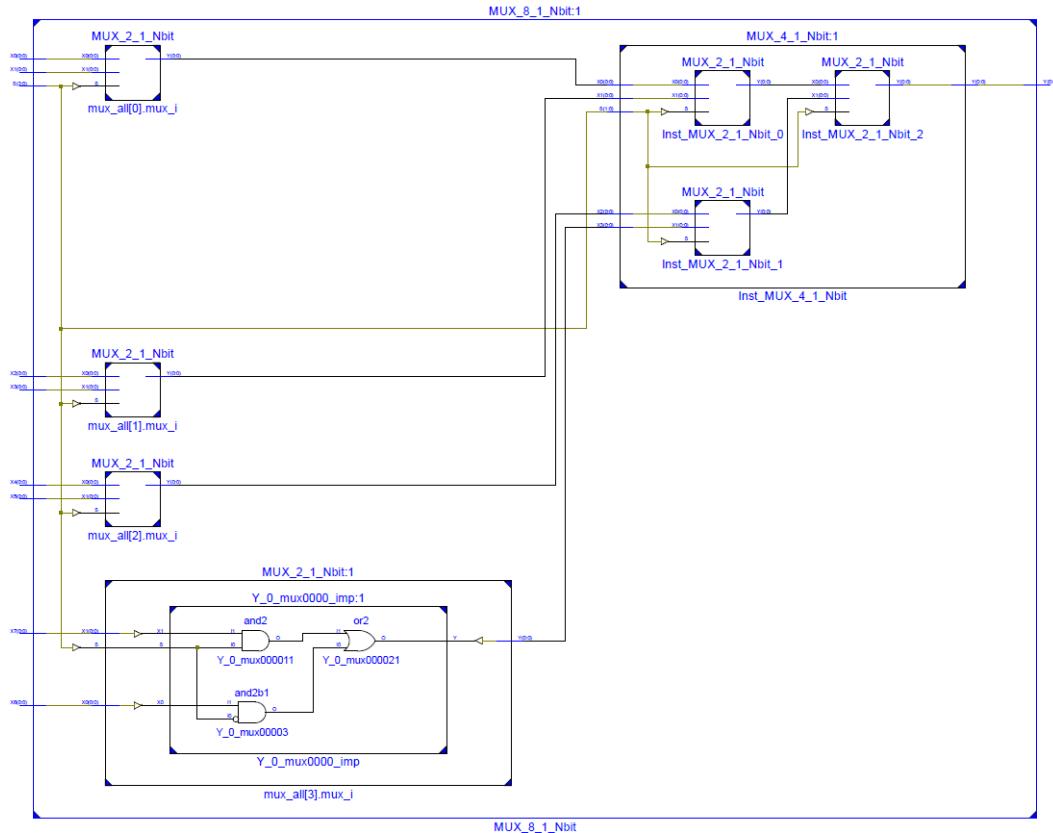


Figura 1.1: Schematico Multiplexer 8:1

1.5 Simulazione

File: MUX_8_1_Nbit_tb.vhd

Codice Testbench Multiplexer 8:1

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY MUX_8_1_Nbit_tb IS
END MUX_8_1_Nbit_tb;

ARCHITECTURE behavior OF MUX_8_1_Nbit_tb IS
    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT MUX_8_1_Nbit
        GENERIC(N : integer);
        PORT(
            X0 : IN std_logic_vector(0 downto 0);
            X1 : IN std_logic_vector(0 downto 0);
            X2 : IN std_logic_vector(0 downto 0);
            X3 : IN std_logic_vector(0 downto 0);
            X4 : IN std_logic_vector(0 downto 0);
            X5 : IN std_logic_vector(0 downto 0);
            X6 : IN std_logic_vector(0 downto 0);
            X7 : IN std_logic_vector(0 downto 0);
            S : IN std_logic_vector(2 downto 0);
            Y0 : OUT std_logic
        );
    END COMPONENT;

```

```

        Y : OUT std_logic_vector(0 downto 0)
    );
END COMPONENT;

--Inputs
signal X0 : std_logic_vector(0 downto 0) := (others => '0');
signal X1 : std_logic_vector(0 downto 0) := (others => '0');
signal X2 : std_logic_vector(0 downto 0) := (others => '0');
signal X3 : std_logic_vector(0 downto 0) := (others => '0');
signal X4 : std_logic_vector(0 downto 0) := (others => '0');
signal X5 : std_logic_vector(0 downto 0) := (others => '0');
signal X6 : std_logic_vector(0 downto 0) := (others => '0');
signal X7 : std_logic_vector(0 downto 0) := (others => '0');
signal S : std_logic_vector(2 downto 0) := (others => '0');

--Outputs
signal Y : std_logic_vector(0 downto 0);

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: MUX_8_1_Nbit
    GENERIC MAP (N => 1)
    PORT MAP (
        X0 => X0,
        X1 => X1,
        X2 => X2,
        X3 => X3,
        X4 => X4,
        X5 => X5,
        X6 => X6,
        X7 => X7,
        S => S,
        Y => Y
    );

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;

        -- insert stimulus here
        X0 <= "1";
        X1 <= "1";
        X2 <= "0";
        X3 <= "0";
        X4 <= "0";
        X5 <= "0";
        X6 <= "1";
        X7 <= "0";

        S <= "010"; -- Selezione X2

        wait for 50 ns;
        assert Y = "0"
        report "errore0"
        severity failure;

        X0 <= "1";
        X1 <= "0";
        X2 <= "0";
        X3 <= "1";
        X4 <= "0";
        X5 <= "0";
        X6 <= "1";
        X7 <= "0";

        S <= "011"; -- Selezione X6

        wait for 50 ns;
        assert Y = "1"
        report "errore0"
        severity failure;

        wait;
    end process;
END;

```

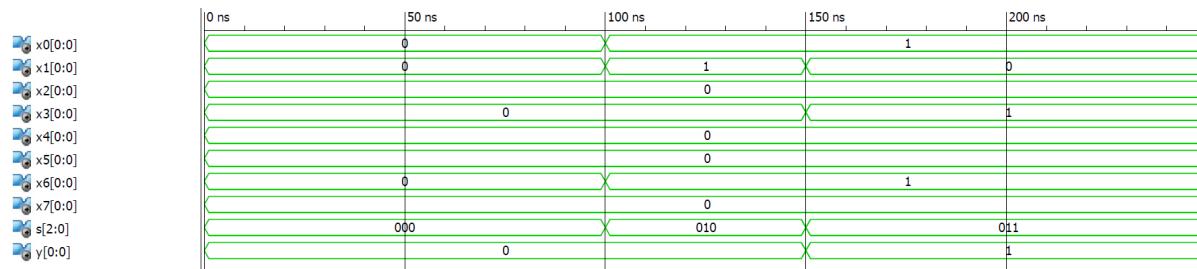


Figura 1.2: Simulazione Multiplexer 8:1

Capitolo 2

Esercizio 2

2.1 Traccia

Progettare un controller per un display a 7 segmenti che, data una stringa in ingresso di 4 bit che codifica un numero naturale fra 0 e 15, fornisca in uscita i segnali a, b, c, d, e, f, g, come in Figura 2.1, che consentono di rappresentare sul display il numero fornito in ingresso rappresentato nella codifica esadecimale (cifre 0 ... 9 A ... F).

Nota: per consentire la visualizzazione di tutte le cifre in maniera univoca è possibile riprodurre le lettere A, C, E ed F in maiuscolo e le lettere B e D in minuscolo.

Il controller deve essere sintetizzato sulla board e deve utilizzare gli switch per acquisire l'input e una cifra delle 4 disponibili per visualizzare l'output.

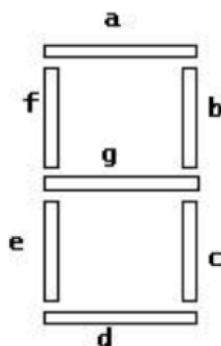


Figura 2.1: Display 7 Segmenti

2.2 Introduzione

Data la semplicità del componente da realizzare, non è stato necessario decomporre l'architettura del dispositivo in parte operativa e parte di controllo. Si è scelto quindi di realizzare un primo componente dedicato alla gestione del display a 7 segmenti ed un secondo componente dedicato all'acquisizione degli ingressi.

2.3 Soluzione

È stato realizzato un controller per un display a 7 segmenti, seguendo un architettura multi-livello, come riportato in Figura 2.2. Al livello più alto si trova il *top-level-module*, ovvero il modulo `display_on_board` (**structural**), in cui sono definiti i segnali di ingresso e uscita del dispositivo e descritte le connessioni dei componenti appartenenti al livello inferiore. Tali componenti sono:

- `display_seven_segments` (**structural**): riceve in ingresso un valore numerico rappresentato su 4 bit e restituisce in uscita le configurazioni di anodi e catodi¹ tali da mostrare sul display la codifica in esadecimale del valore di ingresso. Tale componente è a sua volta costituito da:
 - `cathodes_manager` (**behavioural**): componente responsabile della gestione dei catodi. Viene selezionata la configurazione dei catodi a seconda del valore da visualizzare. Oltre ai 7 bit relativi ai 7 segmenti del display è necessario specificare un ulteriore bit associato al punto della cifra.
 - `anodes_manager` (**behavioural**): componente responsabile della gestione degli anodi. Dal momento che si vuole visualizzare un numero esadecimale rappresentato su 4 bit, viene abilitata un'unica cifra del display.
- `buffer_reg` (**behavioural**): ricevuti in ingresso i segnali di clock, reset ed un segnale di abilitazione, gestisce la lettura del valore numerico da visualizzare. In particolare, quando il segnale di abilitazione è alto, sul fronte di salita successivo del clock, il dato di ingresso viene letto, e fornito in ingresso al componente `display_seven_segments`.

File: `display_on_board.vhd`

Codice Display On Board

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

entity display_on_board is
  Port(
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    load_value : in STD_LOGIC;
    in_value : in STD_LOGIC_VECTOR(3 downto 0);
    anodes : out STD_LOGIC_VECTOR (3 downto 0);
    cathodes : out STD_LOGIC_VECTOR (7 downto 0)
  );
end display_on_board;

architecture Structural of display_on_board is

  COMPONENT display_seven_segments
    PORT(
      value : IN std_logic_vector(3 downto 0);
      anodes : OUT std_logic_vector(3 downto 0);
      cathodes : OUT std_logic_vector(7 downto 0)
    );
  END COMPONENT;

  COMPONENT buffer_reg
    PORT(
      clock : IN std_logic;
```

¹Anodi e catodi lavorano in logica negata.

```

        reset : IN std_logic;
        load_value : IN std_logic;
        in_value : IN std_logic_vector(3 downto 0);
        value : OUT std_logic_vector(3 downto 0)
    );
END COMPONENT;

signal cu_value : std_logic_vector(3 downto 0);

begin

    seven_segment_array: display_seven_segments
    PORT MAP (
        value => cu_value,
        anodes => anodes,
        cathodes => cathodes
    );

    br: buffer_reg PORT MAP (
        clock => clock,
        reset => reset,
        load_value => load_value,
        in_value => in_value,
        value => cu_value
    );
end Structural;

```

File: display_seven_segments.vhd

Codice Display 7 Segmenti

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity display_seven_segments is
    Port ( value : in STD_LOGIC_VECTOR (3 downto 0);
            anodes : out STD_LOGIC_VECTOR (3 downto 0);
            cathodes : out STD_LOGIC_VECTOR (7 downto 0)
        );
end display_seven_segments;

architecture Structural of display_seven_segments is

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

COMPONENT anodes_manager
    GENERIC(
        enable_digit : std_logic_vector(3 downto 0) := "0001"
    );
    PORT(
        anodes : OUT std_logic_vector(3 downto 0)
    );
END COMPONENT;

begin
    cathodes_instance: cathodes_manager port map(
        value => value,
        cathodes => cathodes
    );

    anodes_instance: anodes_manager port map(
        anodes => anodes
    );
end Structural;

```

File: cathodes_manager.vhd

Al fine di effettuare la gestione dei catodi sono state dichiarate all'interno del componente `cathodes_manager` un insieme di costanti, ognuna delle quali rappresenta la configurazione dei catodi necessaria per visualizzare una determinata cifra esadecimale. Il componente riceve in ingresso un valore numerico di 4 bit e ne restituisce in uscita la configurazione dei catodi corrispondente. Infine, viene concatenato alla relativa configurazione un bit pari ad 1^2 necessario per disabilitare il punto della cifra.

Codice Cathodes Manager

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

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

architecture Behavioral of cathodes_manager is

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

signal cathodes_for_digit : std_logic_Vector(6 downto 0) := (others => '0');

begin

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

    cathodes <= not '0' & cathodes_for_digit;
end Behavioral;

```

²Si ricorda che anodi e catodi lavorano in logica negata.

File: anodes_manager.vhd

Codice Anodes Manager

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

entity anodes_manager is
    GENERIC( enable_digit : STD_logic_vector(3 downto 0) := "0001"
            );
    Port ( anodes : out STD_LOGIC_VECTOR (3 downto 0)
           );
end anodes_manager;

architecture Behavioral of anodes_manager is
begin
    anodes <= not enable_digit;
end Behavioral;
```

File: buffer_reg.vhd

Codice Registro Buffer

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

entity buffer_reg is
    Port (
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        load_value : in STD_LOGIC;
        in_value : in STD_LOGIC_VECTOR(3 downto 0);
        value : out STD_LOGIC_VECTOR(3 downto 0)
    );
end buffer_reg;

architecture Behavioral of buffer_reg is
begin
    main: process(clock, reset)
    begin
        if reset = '1' then
            value <= (others => '0');
        elsif clock'event and clock = '1' then          -- ETS
            if load_value = '1' then
                value <= in_value;
            end if;
        end if;
    end process;
end Behavioral;
```

2.4 Schematici

Osservando lo schematico generato automaticamente da ISE è possibile trarre due conclusioni:

1. La descrizione comportamentale adottata per il componente buffer_reg viene tradotta dallo strumento di sintesi in un semplice flip-flop D edge-trigger attivo sul fronte di salita del segnale di clock.

- La dichiarazione delle costanti adottata nel componente `cathodes_manager` viene tradotta dallo strumento di sintesi in una **memoria ROM**, contenente le costanti definite.

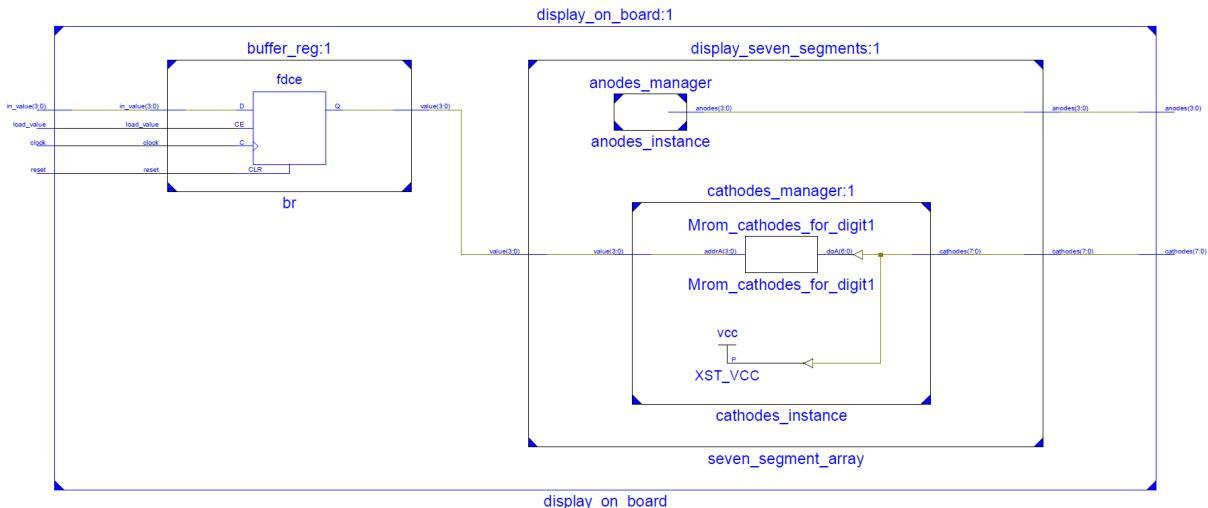


Figura 2.2: Schematico Display On Board

2.5 Sintesi

Come richiesto dalla traccia, è stata effettuata la sintesi del dispositivo sulla board. A tal fine è stato incluso nel progetto un file di configurazione nel quale sono specificati i collegamenti tra i segnali di ingresso-uscita del dispositivo e i diversi componenti della board. In particolare:

- Il segnale di clock è stato associato al clock della scheda.
- I segnali di uscita sono stati associati agli anodi e ai catodi del display presente sulla scheda.
- I bit del segnale di ingresso relativo al valore da visualizzare sono stati associati a 4 degli switch presenti sulla scheda.
- I segnali di reset e caricamento del valore sono stati associati a due pulsanti presenti sulla scheda.

File: Basys_250K.ucf

Codice vincoli board Basys

```
# clock pin for Basys rev E Board
NET "clock"      LOC = "P54";

# onboard 7seg display
NET "cathodes<0>"    LOC = "P25";
NET "cathodes<1>"    LOC = "P16";
NET "cathodes<2>"    LOC = "P23";
NET "cathodes<3>"    LOC = "P21";
NET "cathodes<4>"    LOC = "P20";
NET "cathodes<5>"    LOC = "P17";
NET "cathodes<6>"    LOC = "P83";
NET "cathodes<7>"    LOC = "P22";

NET "anodes<0>"     LOC = "P34";
NET "anodes<1>"     LOC = "P33";
NET "anodes<2>"     LOC = "P32";
NET "anodes<3>"     LOC = "P26";

# Switches
NET "in_value<0>"   LOC = "P38";
NET "in_value<1>"   LOC = "P36";
NET "in_value<2>"   LOC = "P29";
NET "in_value<3>"   LOC = "P24";

# Buttons
NET "reset"          LOC = "P69";
NET "load_value"      LOC = "P48";
```

2.6 Simulazione

Sebbene il comportamento del dispositivo in simulazione non sia facilmente verificabile attraverso la visualizzazione dei segnali di ingresso e uscita, riportati in Figura 2.3, è stato possibile verificare il corretto funzionamento del display utilizzando il costrutto assert.

File: display_tb.vhd

Codice Testbench Display 7 Segmenti

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY display_tb IS
END display_tb;

ARCHITECTURE behavior OF display_tb IS
    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT display_seven_segments
    PORT(
        value : IN std_logic_vector(3 DOWNTO 0);
        anodes : OUT std_logic_vector(3 DOWNTO 0);
        cathodes : OUT std_logic_vector(7 DOWNTO 0)
    );
    END COMPONENT;

    --Inputs
    signal value : std_logic_vector(3 DOWNTO 0) := (others => '0');

    --Outputs
    signal anodes : std_logic_vector(3 DOWNTO 0);
    signal cathodes : std_logic_vector(7 DOWNTO 0);

BEGIN
    -- Instantiate the Unit Under Test (UUT)
```

```

uut: display_seven_segments PORT MAP (
    value => value,
    anodes => anodes,
    cathodes => cathodes
);

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100 ns.
    wait for 100 ns;

    -- insert stimulus here
    value <= x"A";

    wait for 10 ns;

    assert anodes = not "0001" and cathodes = "10001000"
        report "errorel"
        severity failure;

    value <= x"8";

    wait for 10 ns;

    assert anodes = not "0001" and cathodes = "10000000"
        report "errorel"
        severity failure;

    value <= x"5";

    wait for 10 ns;

    assert anodes = not "0001" and cathodes = "10010010"
        report "errorel"
        severity failure;

    wait;
end process;

END;

```

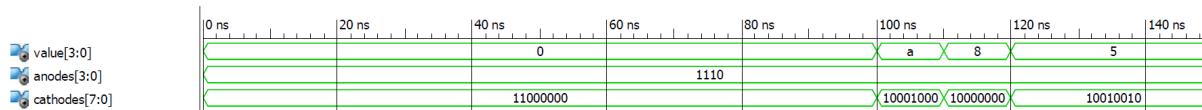


Figura 2.3: Simulazione Display a 7 Segmenti

Capitolo 3

Esercizio 3

3.1 Traccia

Implementare in VHDL e simulare un flip-flop D edge-triggered e master-slave secondo i due differenti modelli visti a lezione (Mazzeo e Sami).

3.2 Introduzione

Per la risoluzione dell'esercizio proposto è stato necessario un profondo studio dei bistabili, elementi di memoria fondamentali. Sono state poi identificate le principali differenze e analogie tra le soluzioni proposte dai professori Mazzeo e Sami nella realizzazione di flip-flop D edge-triggered e master-slave.

È bene osservare che i componenti realizzati in questo capitolo non risultano sintetizzabili sulla board di sviluppo. Difatti, lo strumento di sintesi sconsiglia l'utilizzo di latch, in quanto l'FPGA presente sulla board possiede come elementi di memoria dei flip-flop D.

3.3 Flip-Flop D Edge-Triggered Discesa Sami

3.3.1 Soluzione

Secondo la soluzione adottata da Sami, un flip-flop D edge triggered può essere realizzato mediante un latch D il cui segnale di clock è idealmente impulsivo. Per rendere il clock impulsivo si è posto tale segnale in ingresso ad un blocco derivatore.

È stato quindi realizzato un flip-flop D edge-triggered attivo sul fronte di discesa, seguendo un'architettura multi-livello, come riportato in Figura 3.5. Di seguito sono riportati i diversi livelli dal più interno al più esterno, secondo un approccio di tipo *bottom-up*:

- Il primo livello è costituito dal componente RS_latch (**behavioural**).
- Il secondo livello è costituito dai componenti:

- D_latch (**structural**), ottenuto a partire dal Latch RS, connettendo opportunamente i segnali di ingresso e uscita.
- Derivatore (**data-flow**).
- Il terzo livello è costituito dal componente D_ETd_Sami (**structural**, *top-level-module*) realizzato per composizione dei dispositivi precedenti.

File: RS_latch.vhd

Si è scelto di utilizzare come componente di memoria elementare un latch RS abilitato (o a sincronizzazione esterna), in modo da risolvere il problema delle *alee*, tipico delle macchine sequenziali asincrone.

Di seguito viene riportata in Tabella 3.1 la tabella delle transizioni di stato del componente RS non abilitato e in Figura 3.1 l'automa dello stesso, dal quale si evince che la macchina in questione è una *macchina di Moore*.

q/RS	00	01	11	10	y
q_0	q_0	q_1	-	q_0	0
q_1	q_1	q_1	-	q_0	1

Tabella 3.1: Tabella delle transizioni di stato RS non abilitato

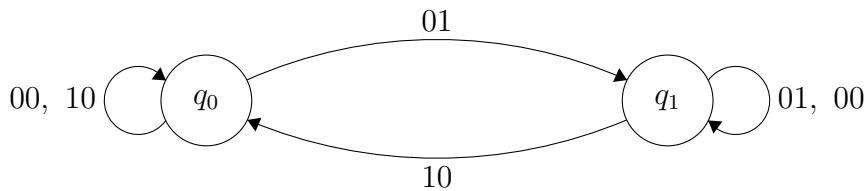


Figura 3.1: Automa RS non abilitato

Il componente RS_latch è stato implementato utilizzando il costrutto `process`. Sono stati inseriti nella *sensitivity list* i segnali di ingresso R (*reset*), S (*set*) ed il segnale di abilitazione E1 (*enable*), in modo da attivare il processo in corrispondenza di una variazione di uno dei tre segnali.

È necessario inserire nella sensitivity list tutti e tre i segnali di ingresso dal momento che il comportamento desiderato del dispositivo è di tipo latch. Se infatti fosse presente solo il segnale di *enable*, il processo verrebbe eseguito solo in corrispondenza delle sue variazioni. Di conseguenza, se i segnali R e S variassero mentre E1 è alto, tali variazioni non causerebbero la ri-esecuzione del processo e la conseguente variazione dell'uscita. Si è scelto inoltre di utilizzare il costrutto `case-when` dal momento che i casi possibili sono mutuamente esclusivi. In alternativa, sarebbe stato comunque possibile utilizzare il costrutto `if-then`.

Codice Latch RS

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RS_latch is
    port(    R      : in STD_LOGIC;
              S      : in STD_LOGIC;
              E1     : in STD_LOGIC;
              Q1     : out STD_LOGIC;
              nQ1    : out STD_LOGIC
            );
end RS_latch;

architecture behavioural of RS_latch is

    signal RS      : STD_LOGIC_VECTOR(1 downto 0);
    RS <= R & S;
    signal TEMP : STD_LOGIC;

begin
    latch: process(RS, E1)
    begin
        if(E1 = '1') then
            case RS is
                when "00" =>
                    null;
                when "01" =>
                    TEMP <= '1';
                when "10" =>
                    TEMP <= '0';
                when "11" =>
                    TEMP <= 'X';
                when others =>
                    TEMP <= 'X';
            end case;
        end if;
    end process;

    -- In alternativa: costrutto if-then
    -- signal TEMP : STD_LOGIC;

    -- latch: process(R, S, E1)
    -- begin
    --     if(E1 = '1') then
    --         if(S = '1' AND R = '0') then
    --             TEMP <= '1';
    --         elsif(S = '0' AND R = '1') then
    --             TEMP <= '0';
    --         elsif(S = '0' AND R = '0') then
    --             null;
    --         elsif(S = '1' AND R = '1') then
    --             TEMP <= 'X';
    --         end if;
    --     end if;
    -- end process;

    Q1      <= TEMP;
    nQ1    <= NOT TEMP;
end behavioural;

```

File: D_latch.vhd

Si è scelto di realizzare un latch D a partire da un latch RS, connettendo i segnali come riportato in Figura 3.5.

Di seguito viene riportata in Tabella 3.2 la tabella delle transizioni di stato del latch D e in Figura 3.2 l'automa dello stesso, dal quale si evince che la macchina in questione è una *macchina di Moore*.

q/AD	00	01	11	10	y
q_0	q_0	q_0	q_1	q_0	0
q_1	q_1	q_1	q_1	q_0	1

Tabella 3.2: Tabella delle transizioni di stato Latch D

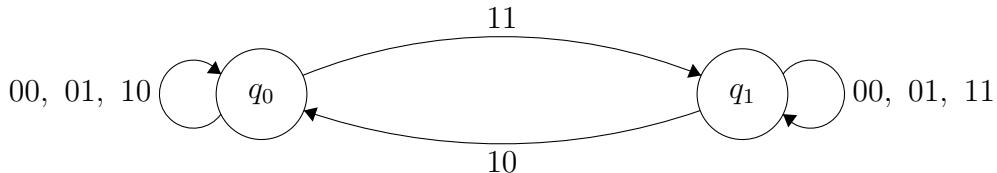


Figura 3.2: Automa Latch D

Codice Latch D

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_latch is
  port(
    D2      : in STD_LOGIC;
    E2      : in STD_LOGIC;
    Q2      : out STD_LOGIC;
    nQ2    : out STD_LOGIC
  );
end D_latch;

architecture structural of D_latch is

  -- Dichiarazione componenti utilizzati nella composizione
  component RS_latch
    port(
      R      : in STD_LOGIC;
      S      : in STD_LOGIC;
      E1     : in STD_LOGIC;
      Q1    : out STD_LOGIC;
      nQ1   : out STD_LOGIC
    );
  end component;

  signal nD2 : STD_LOGIC;

begin
  nD2 <= NOT D2;

  RS_latch0 : RS_latch
    port map(
      R      => nD2,
      S      => D2,
      E1     => E2,
      Q1    => Q2,
      nQ1   => nQ2
    );
end structural;
  
```

File: Derivatore.vhd

Il blocco Derivatore è stato realizzato descrivendo in VHDL il componente omonimo riportato in Figura 3.5. Il dispositivo non fa altro che generare un segnale idealmente impulsivo in corrispondenza del fronte di discesa del segnale di clock, come riportato in Figura 3.3.

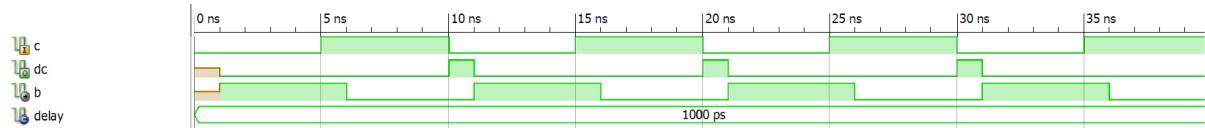


Figura 3.3: Simulazione Derivatore

Codice Derivatore

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

entity Derivatore is
    generic(    delay : TIME := 1 ns);
    port(        C      : in STD_LOGIC;
                dC     : out STD_LOGIC
            );
end Derivatore;

architecture dataflow of Derivatore is

    signal B : STD_LOGIC;

begin
    B      <= NOT C after delay;
    dC     <= NOT (C OR B);
end dataflow;
```

File: D_ETd_Sami.vhd

Si è scelto di realizzare il flip-flop D edge-triggered ponendo l'uscita del derivatore in ingresso al latch D, come riportato in Figura 3.5.

Di seguito viene riportata in Tabella 3.3 la tabella delle transizioni di stato del flip-flop D edge-triggered attivo sul fronte di discesa e in Figura 3.4 l'automa dello stesso, dal quale si evince che la macchina in questione è una *macchina di Moore*.

q/AD	00	01	11	10	y
q_{00}	q_{00}	q_{00}	q_{01}	q_{00}	0
q_{01}	-	q_{11}	q_{01}	q_{00}	0
q_{11}	q_{11}	q_{11}	q_{11}	q_{10}	1
q_{10}	q_{00}	-	q_{11}	q_{10}	1

Tabella 3.3: Tabella delle transizioni di stato Flip-Flop D Edge Triggered Discesa

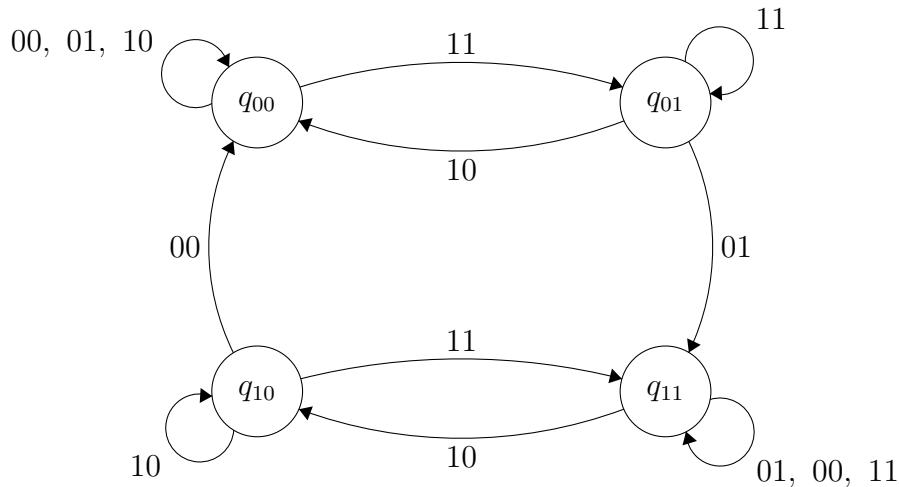


Figura 3.4: Automa Flip-Flop D Edge-Triggered Discesa

Codice Flip-Flop D Edge-Triggered Discesa Sami

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_ETd_Sami is
    port(   D3      : in STD_LOGIC;
            CLK     : in STD_LOGIC;
            Q3      : out STD_LOGIC;
            nQ3    : out STD_LOGIC
        );
end D_ETd_Sami;

architecture structural of D_ETd_Sami is

component D_latch
    port(   D2      : in STD_LOGIC;
            E2      : in STD_LOGIC;
            Q2      : out STD_LOGIC;
            nQ2    : out STD_LOGIC
        );
end component;

component Derivatore
    port(   C       : in STD_LOGIC;
            dC     : out STD_LOGIC
        );
end component;

signal TEMP : STD_LOGIC;

begin
    D_latch0 : D_latch
        port map(   D2      => D3,
                    E2      => TEMP,
                    Q2      => Q3,
                    nQ2    => nQ3
                );
    Derivatore0 : Derivatore
        port map(   C       => CLK,
                    dC     => TEMP
                );
end structural;

```

3.3.2 Schematici

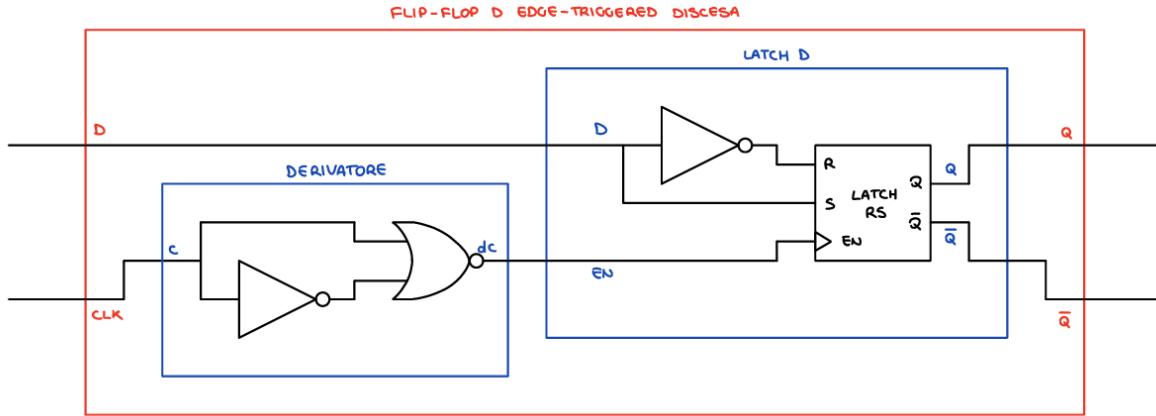


Figura 3.5: Schematico Flip-Flop D Edge-Triggered Discesa Sami

3.3.3 Simulazione

Nel testbench del componente `D_ETd_Sami` è stato evidenziato un caso in cui si manifesta il comportamento non ideale della soluzione proposta da Sami. Va infatti considerato che, non potendo generare fisicamente un impulso ideale, il segnale in uscita dal derivatore è comunque un impulso rettangolare avente una propria durata T ns¹. Durante la durata dell'impulso il dispositivo sarà quindi trasparente, seguendo il comportamento di un latch, piuttosto che di un flip-flop edge-triggered. Per evidenziare tale anomalia, si è fatto quindi variare il dato D da 1 a 0 dopo 0.5 T dal fronte di discesa del clock. Secondo il comportamento ideale, il flip-flop dovrebbe ignorare la variazione del dato e continuare a presentare in uscita il valore 1, campionato sul fronte di discesa del clock. Tuttavia, come riportato in Figura 3.6, l'uscita del dispositivo segue la variazione del dato, comportandosi di fatto come un latch.

File: `D_ETd_Sami_tb.vhd`

Codice Testbench Flip-Flop D Edge-Triggered Discesa Sami

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity D_ETd_Sami_tb is
end D_ETd_Sami_tb;

architecture behavioural of D_ETd_Sami_tb is

component D_ETd_Sami
    port(
        D3      : in STD_LOGIC;
        CLK     : in STD_LOGIC;
        Q3      : out STD_LOGIC;
        nQ3    : out STD_LOGIC
    );
end component;
```

¹Si è supposto $T = 1$ ns.

```

signal D3_tb      : STD_LOGIC;
signal CLK_tb     : STD_LOGIC;
signal Q3_tb      : STD_LOGIC;
signal nQ3_tb     : STD_LOGIC;

constant CLK_period : time := 10 ns;

begin
    UUT : D_ETd_Sami
        port map(
            D3      => D3_tb,
            CLK     => CLK_tb,
            Q3      => Q3_tb,
            nQ3    => nQ3_tb
        );
    CLK_proc: process
    begin
        CLK_tb    <= '0';
        wait for CLK_period/2;
        CLK_tb    <= '1';
        wait for CLK_period/2;
    end process;
    stim_proc: process
    begin
        wait for CLK_period/4;
        D3_tb      <= '1';
        wait until(CLK_tb'EVENT AND CLK_tb = '0');
        wait for 1 ns;
        assert Q3_tb = '1' AND nQ3_tb = '0'
        report "errore0"
        severity failure;
        D3_tb      <= '0';
        wait until(CLK_tb'EVENT AND CLK_tb = '0');
        wait for 1 ns;
        assert Q3_tb = '0' AND nQ3_tb = '1'
        report "errore1"
        severity failure;
        D3_tb      <= '1';
        wait until(CLK_tb'EVENT AND CLK_tb = '0');
        wait for 0.5 ns;    -- Attesa inferiore ad 1 ns
                            -- dopo il fronte di discesa di CLK
        assert Q3_tb = '1' AND nQ3_tb = '0'
        report "errore2"
        severity failure;
        D3_tb      <= '0';
        wait until(CLK_tb'EVENT AND CLK_tb = '0');
        wait for 1 ns;
        assert Q3_tb = '0' AND nQ3_tb = '1'    -- Errore dovuto
                                                -- alla non idealità
        report "errore2"
        severity failure;
        wait;
    end process;
end behavioural;

```

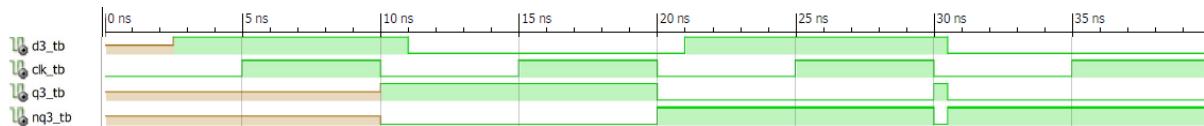


Figura 3.6: Simulazione Flip-Flop D Edge-Triggered Discesa Sami

3.4 Flip-Flop D Edge-Triggered Discesa Mazzeo

3.4.1 Soluzione

Secondo la soluzione adottata da Mazzeo, viene realizzato un flip-flop D edge-triggered attivo sul fronte di discesa effettuando la sintesi della rete corrispondente alla specifica formale espressa mediante la Tabella 3.3.

Per prima cosa è necessario effettuare la codifica degli stati, ottenendo la tabella in codice riportata in Tabella 3.4.

$s_0 s_1 / AD$	00	01	11	10	y
00	00	00	01	00	0
01	-	11	01	00	0
11	11	11	11	10	1
10	00	-	11	10	1

Tabella 3.4: Tabella in codice Flip Flop D Edge Triggered Discesa

A partire dalla tabella in codice si derivano quindi le mappe di Karnaugh per la sintesi delle variabili di stato secondarie S_0 e S_1 , riportate in Tabella 3.5.

Mappa di Karnaugh per S_0						Mappa di Karnaugh per S_1					
$s_0 s_1 / AD$	00	01	11	10	y	$s_0 s_1 / AD$	00	01	11	10	y
00	0	0	0	0	0	00	0	0	1	0	0
01	-	1	0	0	0	01	-	1	1	0	0
11	1	1	1	1	1	11	1	1	1	0	1
10	0	-	1	1	1	10	0	-	1	0	1

Tabella 3.5: Mappe di Karnaugh variabili di stato secondarie

Effettuando il processo di minimizzazione, si ricavano per S_0 e S_1 le seguenti equazioni:

$$S_0 = \bar{A} \cdot s_1 + A \cdot s_0 \quad (3.1)$$

$$S_1 = A \cdot D + \bar{A} \cdot s_1 \quad (3.2)$$

Esprimendo le Equazioni 3.1 e 3.2 in forma di sole porte NOR, è possibile osservare che un flip-flop D edge-triggered attivo sul fronte di discesa può essere ottenuto per composizione di 3 latch RS non abilitati realizzati con porte NOR. Per ulteriori approfondimenti si rimanda a [1].

L'architettura del flip-flop realizzato è stata quindi sviluppata su due livelli, come riportato in Figura 3.7. Il primo livello è costituito da due RS, connessi in modo che l'RS più in basso riceva come segnale di reset la OR tra il segnale di clock e l'uscita del componente più in alto. Il secondo livello è costituito invece da un unico RS, opportunamente pilotato dai dispositivi del livello precedente.

File: RS.vhd

È bene osservare che il componente RS progettato non ha un comportamento ideale, piuttosto si comporta come un circuito reale. Difatti, se entrambi gli ingressi S e R sono alti, il dispositivo restituisce entrambe le uscite Q e nQ basse.

Codice RS Asincrono

```
use IEEE.STD_LOGIC_1164.ALL;

entity RS is
    port(
        R      : in STD_LOGIC;
        S      : in STD_LOGIC;
        Q1    : out STD_LOGIC;
        nQ1   : out STD_LOGIC
    );
end RS;

architecture behavioural of RS is

    signal TEMP : STD_LOGIC;

begin
    latch: process(R, S)
    begin
        if(S = '1' AND R = '0') then
            Q1    <= '1';
            nQ1   <= '0';
        elsif(S = '0' AND R = '1') then
            Q1    <= '0';
            nQ1   <= '1';
        elsif(S = '0' AND R = '0') then
            null;
        elsif(S = '1' AND R = '1') then
            Q1    <= '0';
            nQ1   <= '0';
        end if;
    end process;
end behavioural;
```

File: D_ETd_Mazzeo.vhd

Codice Flip-Flop D Edge-Triggered Discesa Mazzeo

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

entity D_ETd_Mazzeo is
    port (
        D      : in STD_LOGIC;
        CLK    : in STD_LOGIC;
        Q2    : out STD_LOGIC;
        nQ2   : out STD_LOGIC
    );
end D_ETd_Mazzeo;

architecture structural of D_ETd_Mazzeo is

    component RS
        port(
            R      : in STD_LOGIC;
            S      : in STD_LOGIC;
            Q1    : out STD_LOGIC;
            nQ1   : out STD_LOGIC
        );
    end component;

    signal TEMP_OR : STD_LOGIC;
    signal TEMP_R1 : STD_LOGIC;
    signal TEMP_R2 : STD_LOGIC;
    signal TEMP_S3 : STD_LOGIC;

begin
    TEMP_OR <= CLK OR TEMP_R1;

    RS1 : RS
        port map(
            R      => TEMP_OR,
            S      => D,
```

```

Q1      => TEMP_S3,
nQ1    => TEMP_R2
);

RS2 : RS
port map(   R      => TEMP_R2,
              S      => CLK,
              nQ1   => TEMP_R1
);

RS3 : RS
port map(   R      => TEMP_R1,
              S      => TEMP_S3,
              Q1    => Q2,
              nQ1   => nQ2
);
end structural;

```

3.4.2 Schematici

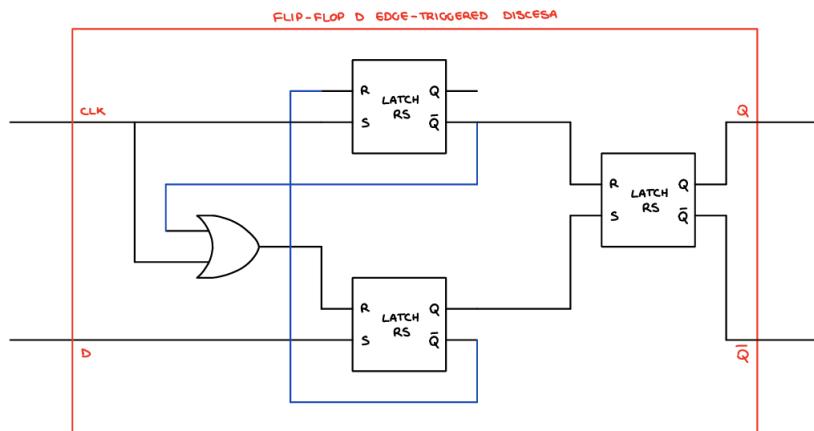


Figura 3.7: Schematico Flip-Flop D Edge-Triggered Discesa Mazzeo

3.4.3 Simulazione

File: D_ETd_Mazzeo_tb.vhd

Codice Testbench Flip-Flop D Edge-Triggered Discesa Mazzeo

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY D_ETd_Mazzeo_tb IS
END D_ETd_Mazzeo_tb;

ARCHITECTURE behavioural OF D_ETd_Mazzeo_tb IS

COMPONENT D_ETd_Mazzeo
PORT(
    D          : IN  std_logic;
    CLK        : IN  std_logic;
    Q2         : OUT std_logic;
    nQ2        : OUT std_logic
);
END COMPONENT;

signal D_tb     : std_logic := '0';
signal CLK_tb  : std_logic := '0';

```

```

signal Q2_tb : std_logic;
signal nQ2_tb : std_logic;

constant CLK_period : time := 10 ns;

BEGIN

  uut: D_ETd_Mazzeo PORT MAP (
    D      => D_tb,
    CLK    => CLK_tb,
    Q2     => Q2_tb,
    nQ2   => nQ2_tb
  );

  Clk_process :process
  begin
    CLK_tb <= '0';
    wait for CLK_period/2;
    CLK_tb <= '1';
    wait for CLK_period/2;
  end process;

  stim_proc:process
  begin
    wait for CLK_period*10;
    wait for 2.5 ns;
    D_tb <= '1';
    wait for CLK_period;
    D_tb <= '0';
    wait for 1 ns;
    D_tb <= '1';
    wait for 3 ns;
    D_tb <= '1';
    wait for 4 ns;
    D_tb <= '0';
    wait;
  end process;
END behavioural;

```

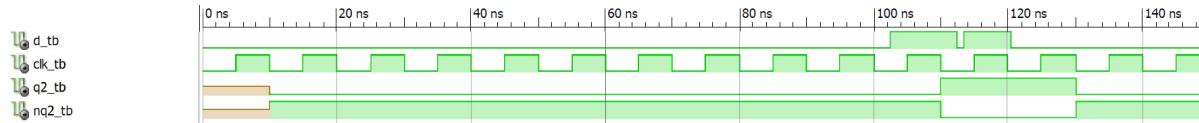


Figura 3.8: Simulazione Flip-Flop D Edge-Triggered Discesa Mazzeo

3.5 Flip-Flop D Master-Slave non ideale

3.5.1 Soluzione

Secondo la soluzione adottata da Mazzeo e Sami, un flip-flop D master-slave viene realizzato utilizzando due latch in cascata, il cui segnale di sincronismo è in contrapposizione di fase.

È stato quindi realizzato un flip-flop D master-slave, seguendo un'architettura multi-livello, come riportato in Figura 3.10. Di seguito sono riportati i diversi livelli dal più interno al più esterno, secondo un approccio di tipo *bottom-up*:

- Il primo livello è costituito da due componenti RS_latch (**behavioural**).

- Il secondo livello è costituito da due componenti D_latch (**structural**).
- Il terzo livello è costituito dal componente D_MS_non_ideale (**structural**, *top-level-module*) realizzato per composizione dei dispositivi precedenti.

File: RS_latch.vhd

Il codice relativo al componente RS_latch è uguale a quello riportato nel Paragrafo 3.3.1.

File: D_latch.vhd

Il codice relativo al componente D_latch è uguale a quello riportato nel Paragrafo 3.3.1.

File: D_MS_non_ideale.vhd

Per realizzare un flip-flop D master-slave sarebbe necessario implementare la macchina sequenziale descritta dalla tabella delle transizioni di stato, riportata in Tabella 3.6, e dal relativo automa riportato in Figura 3.9. Tuttavia la soluzione proposta da Sami e Mazzeo rappresenta un'approssimazione di questo modello ideale, come evidenziato nel Paragrafo 3.5.3.

q/AD	00	01	11	10	y
q_0	q_0	q_0	q_0c_1	q_0c_0	0
q_0c_0	q_0	q_0	q_0c_0	q_0c_0	0
q_0c_1	q_1	q_1	q_0c_1	q_0c_1	0
q_1	q_1	q_1	q_1c_1	q_1c_0	1
q_1c_0	q_0	q_0	q_1c_0	q_1c_0	1
q_1c_1	q_1	q_1	q_1c_1	q_1c_1	1

Tabella 3.6: Tabella delle transizioni di stato Flip Flop Master-Slave non ideale

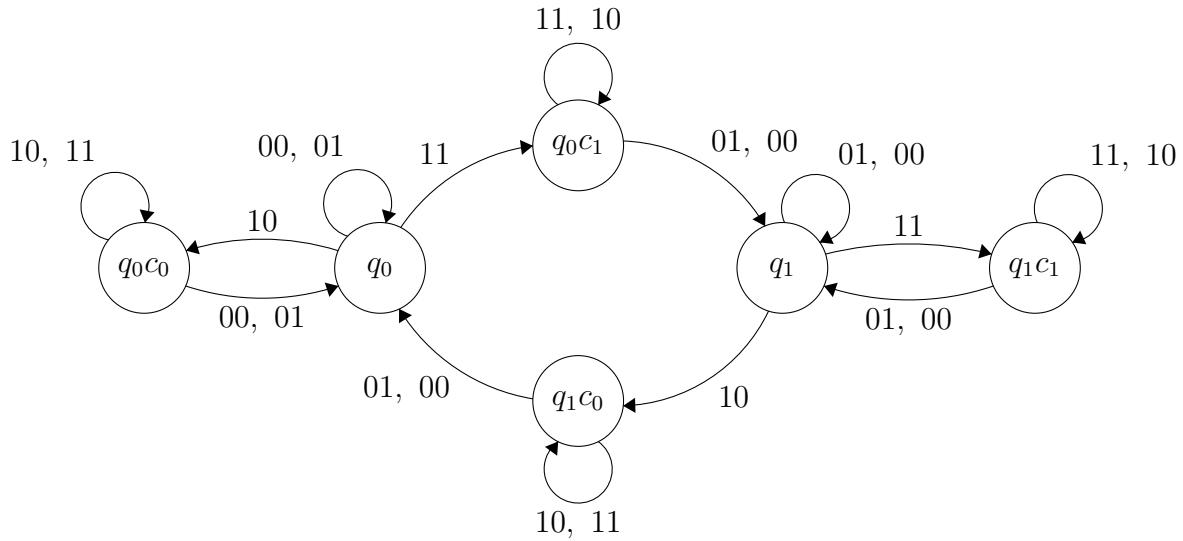


Figura 3.9: Automa Flip Flop Master-Slave non ideale

Codice Flip-Flop D Master-Slave non ideale

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_MS_non_ideale is
    port(
        D3      : in STD_LOGIC;
        CLK     : in STD_LOGIC;
        Q3      : out STD_LOGIC;
        nQ3     : out STD_LOGIC
    );
end D_MS_non_ideale;

architecture structural of D_MS_non_ideale is

component D_latch
    port(
        D2      : in STD_LOGIC;
        E2      : in STD_LOGIC;
        Q2      : out STD_LOGIC;
        nQ2     : out STD_LOGIC
    );
end component;

signal nCLK      : STD_LOGIC;
signal TEMP      : STD_LOGIC;
signal nTEMP     : STD_LOGIC;

begin
    nCLK <= NOT CLK;

    D_master : D_latch
        port map(
            D2      => D3,
            E2      => CLK,
            Q2      => TEMP,
            nQ2     => nTEMP
        );

    D_slave : D_latch
        port map(
            D2      => TEMP,
            E2      => nCLK,
            Q2      => Q3,
            nQ2    => nQ3
        );
end structural;

```

3.5.2 Schematici

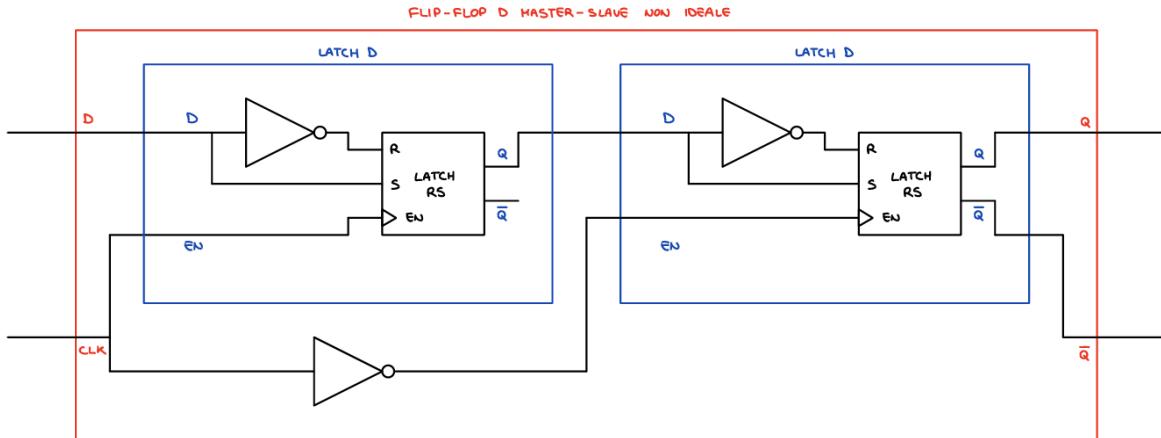


Figura 3.10: Schematico Flip-Flop D Master-Slave non ideale

3.5.3 Simulazione

Nel testbench è stato evidenziato un caso in cui si manifesta il comportamento non ideale presente nella soluzione realizzata con due latch D. Per evidenziare tale anomalia, si è fatto quindi variare il dato D da 0 a 1 mentre il segnale di clock è alto. Secondo il comportamento del flip-flop master slave ideale, il dispositivo dovrebbe ignorare la variazione del dato e presentare sul fronte di discesa in uscita il valore 0, campionato sul fronte di salita. Tuttavia, come riportato in Figura 3.11, l'uscita del dispositivo segue la variazione del dato, comportandosi di fatto come un latch nella frazione di tempo in cui il clock è alto.

File: D_MS_non_ideale_tb.vhd

Codice Testbench Flip-Flop D Master-Slave non ideale

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity D_MS_non_ideale_tb is
end D_MS_non_ideale_tb;

architecture behavioural of D_MS_non_ideale_tb is

component D_MS_non_ideale
    port(   D3      : in STD_LOGIC;
            CLK     : in STD_LOGIC;
            Q3      : out STD_LOGIC;
            nQ3    : out STD_LOGIC
        );
end component;

signal D3_tb : STD_LOGIC;
signal CLK_tb : STD_LOGIC;
signal Q3_tb : STD_LOGIC;
signal nQ3_tb : STD_LOGIC;

constant CLK_period : time := 10 ns;
```

```

begin
UUT : D_MS_non_ideale
  port map(    D3 => D3_tb,
                CLK => CLK_tb,
                Q3 => Q3_tb,
                nQ3 => nQ3_tb
              );
begin
  CLK_tb      <= '0';
  wait for CLK_period/2;
  CLK_tb      <= '1';
  wait for CLK_period/2;
end process;

stim_proc: process
begin
  wait for CLK_period/4;
  D3_tb      <= '1';
  wait until(CLK_tb'EVENT AND CLK_tb = '0');
  wait for 1 ns;
  assert Q3_tb = '1' AND nQ3_tb = '0'
  report "errore0"
  severity failure;

  D3_tb      <= '0';
  wait until(CLK_tb'EVENT AND CLK_tb = '0');
  wait for 1 ns;
  assert Q3_tb = '0' AND nQ3_tb = '1'
  report "errore1"
  severity failure;

  D3_tb      <= '1';
  wait until(CLK_tb'EVENT AND CLK_tb = '0');
  wait for 1 ns;
  assert Q3_tb = '1' AND nQ3_tb = '0'
  report "errore2"
  severity failure;

  D3_tb      <= '0';
  wait until(CLK_tb'EVENT AND CLK_tb = '1');    -- Il flip-flop campiona
                                                 -- 0 sul fronte di salita
  wait for 1 ns;
  D3_tb      <= '1';    -- Il dato varia mentre CLK e' alto
  wait until(CLK_tb'EVENT AND CLK_tb = '0');
  wait for 1 ns;
  assert Q3_tb = '1' AND nQ3_tb = '0'      -- Non idealita': il flip-flop
                                             -- presenta 1 sul fronte
                                             -- di discesa
  report "errore2"
  severity failure;
  wait;
end process;
end behavioural;

```



Figura 3.11: Simulazione Flip-Flop Master-Slave non ideale

3.6 Flip-Flop D MS ideale

3.6.1 Soluzione

Si vuole infine presentare un'ultima soluzione per la realizzazione di un filp-flop D master-slave. A differenza del caso precedente, il componente è stato realizzato utilizzando due flip-flop D edge-triggered attivi sul fronte di discesa, il cui segnale di sincronismo è ancora una volta in contrapposizione di fase. In tal caso, il comportamento risultante è quello di un master-slave ideale, tuttavia la macchina implementata non ne è una realizzazione esatta. Difatti, i due flip-flop D edge-triggered connessi in cascata determinano una macchina avente $4 \times 4 = 16$ stati, mentre la macchina ideale ne possiede soltanto 6.

È stato quindi realizzato un flip-flop D master-slave seguendo un'architettura multi-livello, come riportato in Figura 3.12. Di seguito sono riportati i diversi livelli dal più interno al più esterno, secondo un approccio di tipo *bottom-up*:

- Il primo livello è costituito da un totale di 6 dispositivi RS (**behavioural**), descritti nel Paragrafo 3.4.1.
- Il secondo livello è costituito da due componenti D_ETd_Mazzeo (**structural**), descritti nel Paragrafo 3.4.1.
- Il terzo livello è costituito dal componente D_MS_ideale (**structural**, *top-level-module*) realizzato per composizione dei due flip-flop precedenti.

È bene osservare che per realizzare un flip-flop D master-slave ideale si hanno a disposizione diverse soluzioni:

1. Utilizzare due flip-flop D edge-triggered attivi sullo stesso fronte, a venti segnali di sincronismo opposti (soluzione scelta).
2. Utilizzare due flip-flop D edge-triggered attivi su fronti opposti, a venti stesso segnale di sincronismo.
3. Utilizzare un flip-flop D edge-triggered attivo sul fronte di salita ed un latch D, a venti segnali di sincronismo opposto.

Per superare la non idealità descritta nel Paragrafo 3.5.3 è infatti sufficiente fare in modo che il primo componente non segua la variazione del dato mentre l'abilitazione è alta.

File: RS.vhd

Il codice relativo al componente RS è uguale a quello riportato nel Paragrafo 3.4.1.

File: D_ETd_Mazzeo.vhd

Il codice relativo al componente D_ETd_Mazzeo è uguale a quello riportato nel Paragrafo 3.4.1.

File: Flip-Flop_D_MS_ideale.vhd

Codice Flip-Flop D Master-Slave ideale

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

entity D_MS_ideale is
    port ( D3      : in  STD_LOGIC;
           CLK3     : in  STD_LOGIC;
           Q3      : out STD_LOGIC;
           nQ3     : out STD_LOGIC
         );
end D_MS_ideale;

architecture structural of D_MS_ideale is

component D_ETd_Mazzeo
    port ( D2      : in  STD_LOGIC;
           CLK2     : in  STD_LOGIC;
           Q2      : out STD_LOGIC;
           nQ2     : out STD_LOGIC
         );
end component;

signal TEMP : STD_LOGIC;

begin
    -- Istanziazione e mapping dei componenti
    D_ETd0 : D_ETd_Mazzeo
        port map(      D2      => D3,
                      CLK2    => NOT(CLK3),
                      Q2      => TEMP,
                      -- nQ2
                    );
    D_ETd1 : D_ETd_Mazzeo
        port map(      D2      => TEMP,
                      CLK2    => CLK3,
                      Q2      => Q3,
                      nQ2     => nQ3
                    );
end structural;
```

3.6.2 Schematici

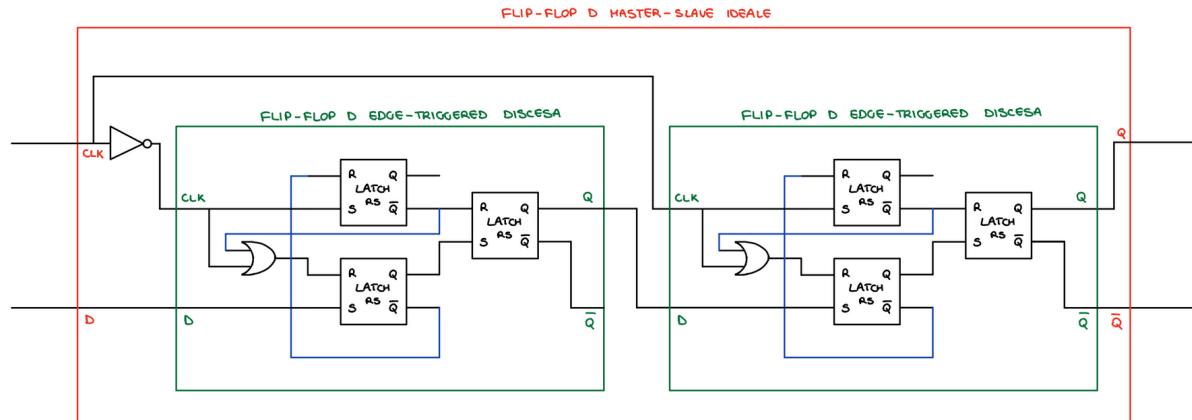


Figura 3.12: Schematico Flip-Flop D Master-Slave ideale

3.6.3 Simulazione

File: **Flip-Flop_D_MS_ideale_tb**

Osservando la simulazione in Figura 3.13 si evince che il comportamento della macchina realizzata coincide con quello del master-slave ideale. È necessario precisare tuttavia che nei casi in cui i componenti master e slave lavorino in contrapposizione di fase sfruttando un *inverter*, in realtà non sono perfettamente sfasati di 180° . L'inverter presenta infatti un proprio ritardo T che crea una *finestra di trasparenza*, la quale, seppur di breve durata, lascia abilitati sia il master che lo slave, cosa che potrebbe creare anomalie rispetto al funzionamento teorico del master-slave.

Codice Testbench Flip-Flop D Master-Slave Mazzeo

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY D_MS_ideale_tb IS
END D_MS_ideale_tb;

ARCHITECTURE behavioural OF D_MS_ideale_tb IS

COMPONENT D_MS_ideale
PORT(
    D3      : IN  std_logic;
    Clk3    : IN  std_logic;
    Q3      : OUT std_logic;
    nQ3    : OUT std_logic
);
END COMPONENT;

signal D3_tb      : std_logic;
signal Clk3_tb    : std_logic;
signal Q3_tb      : std_logic;
signal nQ3_tb     : std_logic;
constant CLK_period : time := 10 ns;
BEGIN
    uut: D_MS_ideale PORT MAP (

```

```

D3      => D3_tb,
CLK3    => CLK3_tb,
Q3      => Q3_tb,
nQ3     => nQ3_tb
);

Clk_process :process
begin
  CLK3_tb <= '0';
  wait for CLK_period/2;
  CLK3_tb <= '1';
  wait for CLK_period/2;
end process;

stim_proc:process
begin
  wait for Clk_period*10;
  wait for 2.5 ns;
  D3_tb <= '1';
  wait for Clk_period;
  D3_tb <= '0';
  wait for 1 ns;
  D3_tb <= '1';
  wait for 3 ns;
  D3_tb <= '1';
  wait for 4 ns;
  D3_tb <= '0';

  wait;
end process;
END behavioural;

```

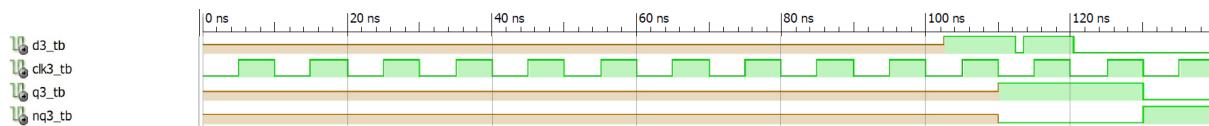


Figura 3.13: Simulazione Flip-Flop Master-Slave ideale

Capitolo 4

Esercizio 4

4.1 Traccia

Progettare ed implementare in VHDL un sommatore a propagazione dei riporti per stringhe di 4 bit, ed utilizzarlo successivamente per realizzare un sommatore di stringhe di 8 bit.

Sintetizzare sulla board il sommatore di stringhe di 8 bit, utilizzando gli switch e i bottoni per inserire le stringhe di input, due cifre del display a 7 segmenti per visualizzare l'output su 8 bit, e un led per segnalare la condizione di overflow.

Nota: i due addendi devono essere acquisiti in due tempi diversi, secondo una tecnica a scelta dello studente.

4.2 Introduzione

Essendo il sommatore *Ripple-Carry Adder* (RCA) una macchina aritmetica puramente combinatoria, non è stato necessario decomporre l'architettura del dispositivo in parte operativa e parte di controllo. Ad ogni modo, per la realizzazione del dispositivo è stato seguito un approccio di tipo strutturale. Il primo passo ha previsto quindi lo sviluppo di un full-adder elementare. Successivamente sono stati realizzati i sommatori di maggiori dimensioni a partire dai componenti di base.

4.3 Soluzione

È stato realizzato un sommatore RCA per numeri naturali codificati su 8 bit seguendo un'architettura multi-livello come riportato in Figura 4.1. Al livello più alto si trova il *top-level-module*, ovvero il modulo `RCA_8bit` (**structural**). Tale componente riceve in ingresso due operandi di 8 bit ed un eventuale riporto entrante di un bit. In uscita presenta invece la somma dei due operandi ed un eventuale riporto uscente, utilizzato per segnalare la condizione di **overflow**. I componenti appartenenti al livello inferiore sono:

- `RCA_4bit` (**structural**): sommatore RCA per stringhe di 4 bit. Ricevuti in ingresso due operandi di 4 bit ed un eventuale riporto entrante, restituisce in uscita la somma dei due operandi ed un eventuale riporto uscente.

All'interno del *top-level-module* sono istanziati due sommatori di questo tipo, in modo che il riporto uscente dal primo venga posto in ingresso al secondo. Inoltre, il primo sommatore riceve i 4 bit meno significativi degli operandi del componente del livello superiore, mentre il secondo riceve i 4 bit più significativi.

Ciascun RCA è composto a sua volta da quattro full-adder elementari:

- **full_adder (dataflow)**: componente in grado di effettuare la somma di due bit ed un eventuale riporto entrante, restituendo in uscita un bit di somma ed un eventuale riporto uscente. Tale componente è stato realizzato descrivendo la relazione ingresso-uscita ottenuta dalla tabella di verità riportata in Tabella 4.1.

Ingressi			Uscite	
X	Y	C_{in}	Z	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabella 4.1: Tabella di verità Full-Adder

A partire dalla tabella di verità sono state ottenute le mappe di Karnaugh delle due uscite del full-adder, riportate in Tabella 4.2.

Mappa di Karnaugh Z		Mappa di Karnaugh C_{out}	
XY/C_{in}	0	1	0
00	0	1	0
01	1	0	1
11	0	1	1
10	1	0	1

Tabella 4.2: Mappe di Karnaugh uscite Full-Adder

Da esse sono state ricavate le seguenti relazioni ingresso-uscita:

$$\begin{aligned}
 Z &= X \cdot \bar{Y} \cdot \bar{C}_{in} + \bar{X} \cdot \bar{Y} \cdot C_{in} + \bar{X} \cdot Y \cdot \bar{C}_{in} + X \cdot Y \cdot C_{in} \\
 &= \bar{Y} \cdot (X \cdot \bar{C}_{in} + \bar{X} \cdot C_{in}) + Y \cdot (X \cdot \bar{C}_{in} + \bar{X} \cdot C_{in}) \\
 &= \bar{Y} \cdot (X \oplus C_{in}) + Y \cdot (X \oplus C_{in}) \\
 &= Y \oplus X \oplus C_{in}
 \end{aligned} \tag{4.1}$$

$$\begin{aligned}
 C_{out} &= X \cdot Y \cdot \bar{C}_{in} + X \cdot Y \cdot C_{in} + \bar{X} \cdot Y \cdot C_{in} + X \cdot \bar{Y} \cdot C_{in} \\
 &= C_{in} \cdot (\bar{X} \cdot Y + \bar{Y} \cdot X) + (X \cdot Y) \cdot (C_{in} + \bar{C}_{in}) \\
 &= (X \cdot Y) + C_{in} \cdot (X \oplus Y)
 \end{aligned} \tag{4.2}$$

File: RCA_8bit.vhd

Codice RCA su 8 bit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RCA_8bit is
    port(
        A1      : in STD_LOGIC_VECTOR (7 downto 0);
        B1      : in STD_LOGIC_VECTOR (7 downto 0);
        cin1   : in STD_LOGIC;
        S1      : out STD_LOGIC_VECTOR (7 downto 0);
        cout1  : out STD_LOGIC
    );
end RCA_8bit;

architecture structural of RCA_8bit is

component RCA_4bit
    port(
        A      : in STD_LOGIC_VECTOR (3 downto 0);
        B      : in STD_LOGIC_VECTOR (3 downto 0);
        cin   : in STD_LOGIC;
        S      : out STD_LOGIC_VECTOR (3 downto 0);
        cout  : out STD_LOGIC
    );
end component;

signal c1 : STD_LOGIC;

begin
    RCA_4bit_0 : RCA_4bit
        port map(
            A      => A1(3 downto 0),
            B      => B1(3 downto 0),
            cin   => cin1,
            S      => S1(3 downto 0),
            cout  => c1
        );
    RCA_4bit_1 : RCA_4bit
        port map(
            A      => A1(7 downto 4),
            B      => B1(7 downto 4),
            cin   => c1,
            S      => S1(7 downto 4),
            cout  => cout1
        );
end structural;

```

File: RCA_4bit.vhd

Codice RCA su 4 bit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RCA_4bit is
    port(
        A      : in STD_LOGIC_VECTOR (3 downto 0);
        B      : in STD_LOGIC_VECTOR (3 downto 0);
        cin   : in STD_LOGIC;
        S      : out STD_LOGIC_VECTOR (3 downto 0);
        cout  : out STD_LOGIC
    );
end RCA_4bit;

architecture structural of RCA_4bit is

component full_adder
    port(x      : in STD_LOGIC;

```

```

        y      : in STD_LOGIC;
        c_in   : in STD_LOGIC;
        z      : out STD_LOGIC;
        c_out  : out STD_LOGIC
    );
end component;

constant n : POSITIVE := 4;
signal c : STD_LOGIC_VECTOR (n-2 downto 0);

begin
full_adder_all: for i in 0 to n-1 generate
    l: if i = 0 generate
        least: full_adder port map(x      => A(i),
                                     y      => B(i),
                                     c_in  => cin,
                                     z      => S(i),
                                     c_out  => c(i))
    );
end generate;
r: if i > 0 AND i < n-1 generate
    rest: full_adder port map(x      => A(i),
                               y      => B(i),
                               c_in  => c(i-1),
                               z      => S(i),
                               c_out  => c(i))
    );
end generate;
m: if i = n-1 generate
    most: full_adder port map(x      => A(i),
                               y      => B(i),
                               c_in  => c(i-1),
                               z      => S(i),
                               c_out  => cout)
    );
end generate;
end generate;
end structural;

```

File: full_adder.vhd

Codice Full-Adder

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder is
    port(x      : in STD_LOGIC;
          y      : in STD_LOGIC;
          c_in   : in STD_LOGIC;
          z      : out STD_LOGIC;
          c_out  : out STD_LOGIC
    );
end full_adder;

architecture dataflow of full_adder is
begin
    z <= x XOR y XOR c_in;
    c_out <= (x AND y) OR (c_in AND (x XOR y));
end dataflow;

```

4.4 Schematici

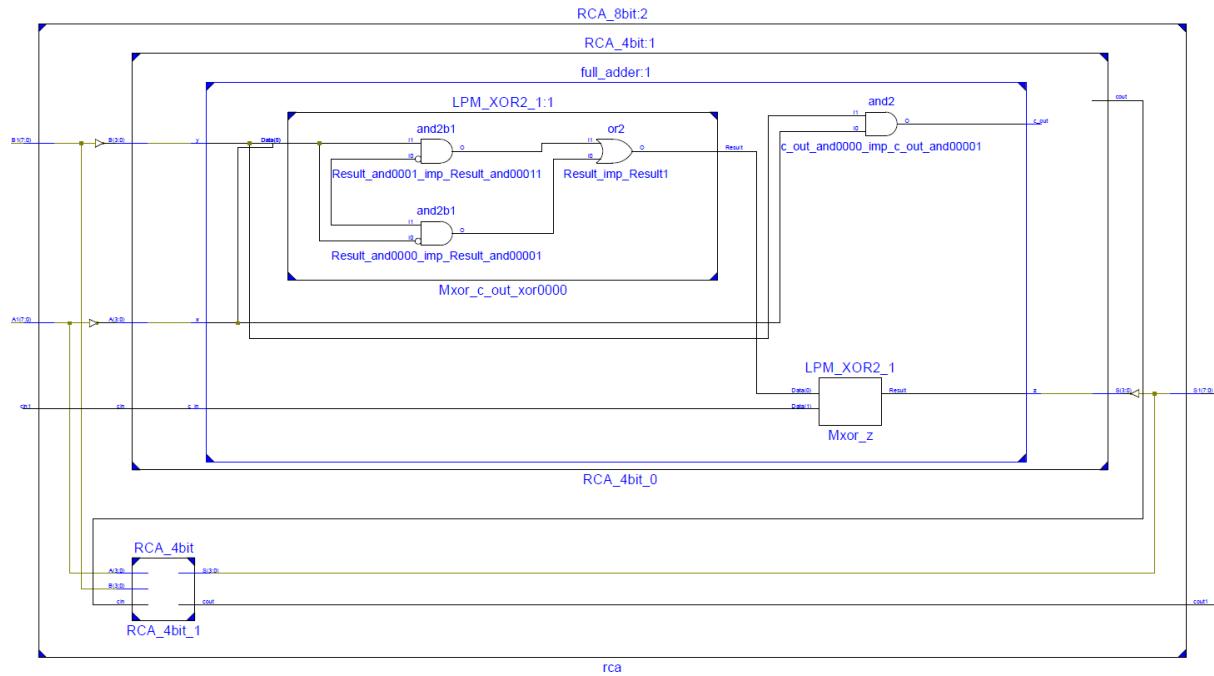


Figura 4.1: Schematico RCA su 8 bit

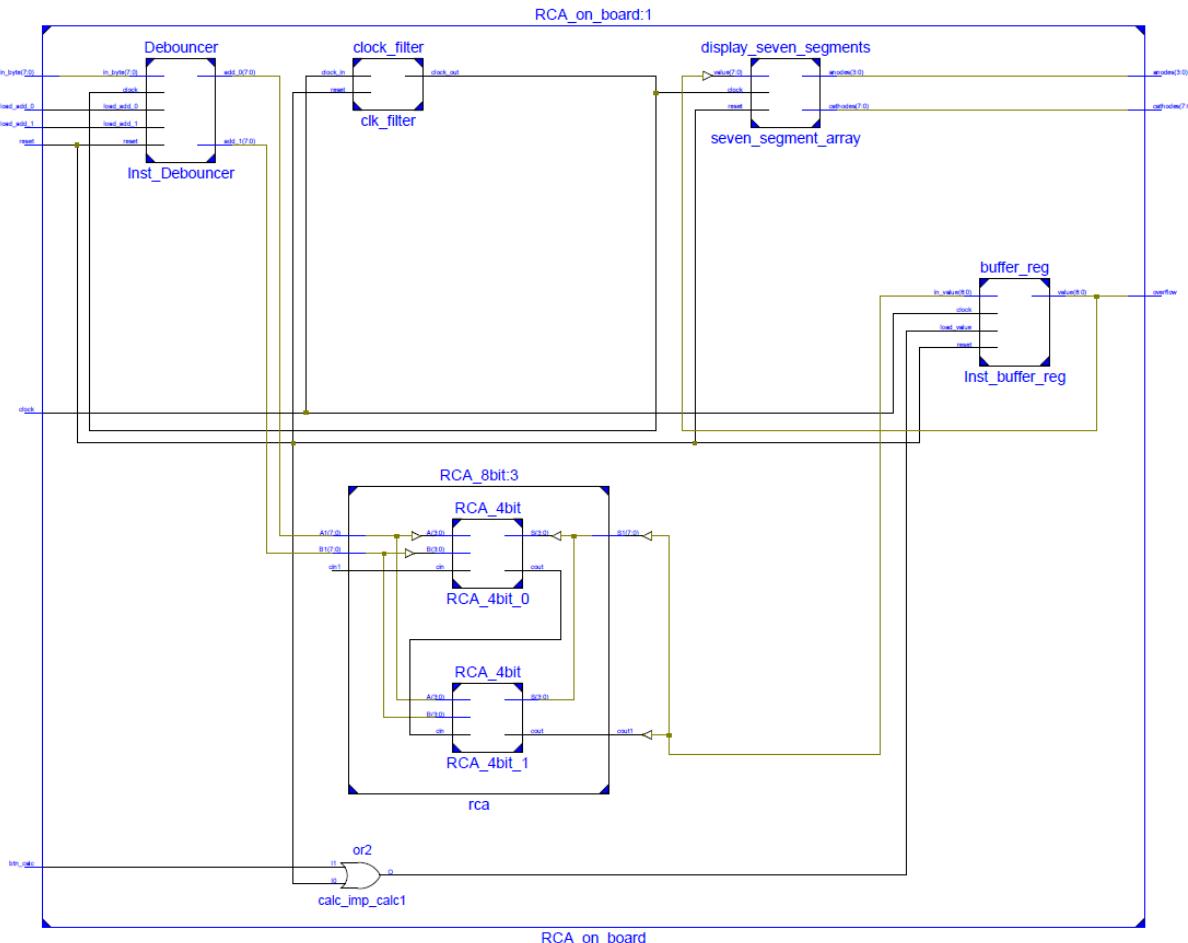


Figura 4.2: Schematico RCA On Board

4.5 Sintesi

Come richiesto dalla traccia, è stata effettuata la sintesi del dispositivo sulla board. A tal fine sono stati prodotti ulteriori moduli VHDL.

Il dispositivo sintetizzato sulla board presenta ancora una volta una struttura multilivello, come riportato in Figura 4.2. Al livello più alto si trova il *top-level-module*, ovvero **RCA_on_board** (**structural**). Tale componente definisce i segnali di ingresso e uscita del dispositivo e descrive le connessioni tra i componenti appartenenti al livello inferiore:

- **RCA_8bit (structural)**: sulla base di quanto esposto in precedenza, rappresenta un sommatore RCA per stringhe di 8 bit.
- **clock_filter (behavioural)**: componente responsabile della divisione in frequenza del segnale di clock, del tutto identico a quello descritto nel Paragrafo 5.5.
- **Debouncer (behavioural)**: ricevuti in ingresso i segnali di clock, reset e due segnali di abilitazione alla lettura dagli switch, gestisce l'acquisizione degli addendi. Tale componente è analogo a quello descritto nel Paragrafo 5.5.

- `display_seven_segments` (**structural**): riceve in ingresso un valore numerico rappresentato su 8 bit e restituisce in uscita le configurazioni di anodi e catodi tali da mostrare su due cifre del display la codifica in esadecimale il risultato della somma. Tale componente è a sua volta costituito da:
 - `cathodes_manager` (**behavioural**): componente responsabile della gestione dei catodi.
 - `anodes_manager` (**behavioural**): componente responsabile della gestione degli anodi.
 - `counter_mod2` (**behavioural**): contatore modulo 2.
- `buffer_reg` (**behavioural**): registro di 9 bit inserito al fine di bufferizzare i segnali di uscita del sommatore. Gli 8 bit meno significativi corrispondono al risultati della somma, il bit più significativo è associato invece al riporto uscente.

File: RCA_on_board.vhd

Si evidenziano di seguito alcune delle connessioni tra i componenti istanziati nel *top-level module*:

- `RCA_8bit`: riceve come operandi i valori in uscita dal Debouncer. I segnali di uscita sono invece posti in ingresso al componente `buffer_reg`.
- Debouncer: riceve come segnale di clock l'uscita del componente `clock_filter`. I segnali di caricamento degli addendi sono associati a due pulsanti presenti sulla scheda.
- `buffer_reg`: gli 8 bit meno significativi del segnale di uscita di tale componente sono posti in ingresso al modulo `display_seven_segments`; il bit più significativo è invece associato ad uno dei led della board.

Codice RCA On Board

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RCA_on_board is
  Port(
    clock      : in STD_LOGIC;
    reset      : in STD_LOGIC;
    load_add_0 : in STD_LOGIC;
    load_add_1 : in STD_LOGIC;
    in_byt     : in STD_LOGIC_VECTOR(7 downto 0);
    btn_calc   : in STD_LOGIC;
    anodes     : out STD_LOGIC_VECTOR (3 downto 0);
    cathodes   : out STD_LOGIC_VECTOR (7 downto 0);
    overflow   : out STD_LOGIC
  );
end RCA_on_board;

architecture structural of RCA_on_board is

COMPONENT display_seven_segments
  PORT(
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    value : in STD_LOGIC_VECTOR (7 downto 0);
    anodes : out STD_LOGIC_VECTOR (3 downto 0);
    cathodes : out STD_LOGIC_VECTOR (7 downto 0)
  );
END COMPONENT;

```

```

COMPONENT Debouncer
  PORT(
    clock : IN std_logic;
    reset : IN std_logic;
    load_add_0 : IN std_logic;
    load_add_1 : IN std_logic;
    in_byte : IN std_logic_vector(7 downto 0);
    add_0 : OUT std_logic_vector(7 downto 0);
    add_1 : OUT std_logic_vector(7 downto 0)
  );
END COMPONENT;

COMPONENT clock_filter
  GENERIC(
    clock_frequency_in : integer := 50000000;
    clock_frequency_out : integer := 500
  );
  PORT(
    clock_in      : IN STD_LOGIC;
    reset         : IN STD_LOGIC;
    clock_out     : OUT STD_LOGIC
  );
END COMPONENT;

COMPONENT RCA_8bit
  port(
    A1          : in STD_LOGIC_VECTOR (7 downto 0);
    B1          : in STD_LOGIC_VECTOR (7 downto 0);
    cin1        : in STD_LOGIC;
    S1          : out STD_LOGIC_VECTOR (7 downto 0);
    cout1       : out STD_LOGIC
  );
END COMPONENT;

COMPONENT buffer_reg
  PORT(
    clock : IN std_logic;
    reset : IN std_logic;
    load_value : IN std_logic;
    in_value : IN std_logic_vector(8 downto 0);
    value : OUT std_logic_vector(8 downto 0)
  );
END COMPONENT;

signal carry_in : std_logic := '0';
signal buf_in : std_logic_vector(8 downto 0);
signal buf_out : std_logic_vector(8 downto 0);
signal s_temp : std_logic_vector(7 downto 0);
signal c_out : std_logic;
signal add_0_temp : std_logic_vector(7 downto 0);
signal add_1_temp : std_logic_vector(7 downto 0);
signal calc : std_logic;
signal clock_fx : std_logic;

begin
  calc <= reset or btn_calc;
  buf_in <= c_out & s_temp;
  seven_segment_array: display_seven_segments
    PORT MAP (
      clock => clock_fx,
      reset => reset,
      value => buf_out(7 downto 0),
      anodes => anodes,
      cathodes => cathodes
    );
  Inst_Debouncer: Debouncer PORT MAP (
    clock => clock_fx,
    reset => reset,
    load_add_0 => load_add_0,
    load_add_1 => load_add_1,
    in_byte => in_byte,
    add_0 => add_0_temp,
    add_1 => add_1_temp
  );
  clk_filter: clock_filter
    GENERIC MAP (
      clock_frequency_in => 50000000,
      clock_frequency_out => 500
    )
    PORT MAP (
      clock_in => clock,
      reset => reset,

```

```

        clock_out => clock_fx
    );
rca: RCA_8bit PORT MAP (
    A1      => add_0_temp,
    B1      => add_1_temp,
    cinl    => carry_in,
    S1      => s_temp,
    cout1   => c_out
);
Inst_buffer_reg: buffer_reg PORT MAP (
    clock => clock,
    reset => reset,
    load_value => calc,
    in_value => buf_in,
    value => buf_out
);
overflow <= buf_out(8);
end structural;

```

File: clock_filter.vhd

Il codice relativo al componente `clock_filter` è uguale a quello riportato nel Paragrafo 5.5.

File: Debouncer.vhd

Codice Debouncer

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Debouncer is
    Port (
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        load_add_0 : in STD_LOGIC;
        load_add_1 : in STD_LOGIC;
        in_byte : in STD_LOGIC_VECTOR(7 downto 0);
        add_0 : out STD_LOGIC_VECTOR(7 downto 0);
        add_1 : out STD_LOGIC_VECTOR(7 downto 0)
    );
end Debouncer;

architecture behavioral of Debouncer is

signal add_0_temp : std_logic_vector(7 downto 0) := (others => '0');
signal add_1_temp : std_logic_vector(7 downto 0) := (others => '0');

begin

add_0 <= add_0_temp;
add_1 <= add_1_temp;

main: process(clock, reset)
begin

if reset = '1' then
    add_1_temp <= (others => '0');
    add_0_temp <= (others => '0');
elsif rising_edge(clock) then
    if load_add_0 = '1' then
        add_0_temp <= in_byte;
    elsif load_add_1 = '1' then
        add_1_temp <= in_byte;
    end if;
end if;

end process;
end behavioral;

```

File: buffer_reg.vhd

Codice Buffer Register

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

entity buffer_reg is
  Port (
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    load_value : in STD_LOGIC;
    in_value : in STD_LOGIC_VECTOR(8 downto 0);
    value : out STD_LOGIC_VECTOR(8 downto 0)
  );
end buffer_reg;

architecture Behavioral of buffer_reg is

begin

  main: process(clock, reset)
  begin
    if reset = '1' then
      value <= (others => '0');
    elsif rising_edge(clock) then
      if load_value = '1' then
        value <= in_value;
      end if;
    end if;
  end process;
end Behavioral;
```

File: display_seven_segments.vhd

Il codice relativo al componente display_seven_segments è uguale a quello riportato nel Paragrafo 6.3.3.

File: cathodes_manager.vhd

Il codice relativo al componente cathodes_manager è uguale a quello riportato nel Paragrafo 6.3.3.

File: anodes_manager.vhd

Il codice relativo al componente anodes_manager è uguale a quello riportato nel Paragrafo 6.3.3.

File: counter_mod2.vhd

Il codice relativo al componente counter_mod2 è uguale a quello riportato nel Paragrafo 6.3.3.

File: BasysRevEGeneral.ucf

Codice vincoli board Basys

```
# clock pin for Basys rev E Board
NET "clock"      LOC = "P54";

# onboard 7seg display
NET "cathodes<0>"    LOC = "P25";
NET "cathodes<1>"    LOC = "P16";
NET "cathodes<2>"    LOC = "P23";
NET "cathodes<3>"    LOC = "P21";
NET "cathodes<4>"    LOC = "P20";
NET "cathodes<5>"    LOC = "P17";
NET "cathodes<6>"    LOC = "P83";
NET "cathodes<7>"    LOC = "P22";

NET "anodes<0>"     LOC = "P34";
NET "anodes<1>"     LOC = "P33";
NET "anodes<2>"     LOC = "P32";
NET "anodes<3>"     LOC = "P26";

# Leds
NET "overflow"      LOC = "P15";

# Switches
NET "in_byte<0>"   LOC = "P38";
NET "in_byte<1>"   LOC = "P36";
NET "in_byte<2>"   LOC = "P29";
NET "in_byte<3>"   LOC = "P24";
NET "in_byte<4>"   LOC = "P18";
NET "in_byte<5>"   LOC = "P12";
NET "in_byte<6>"   LOC = "P10";
NET "in_byte<7>"   LOC = "P6";

# Buttons
NET "load_add_0"    LOC = "P69";
NET "load_add_1"    LOC = "P48";
NET "btn_calc"       LOC = "P47";
NET "reset"          LOC = "P41";
```

4.6 Simulazione

File: RCA_8bit_tb.vhd

Codice Testbench RCA su 8 bit

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY RCA_8bit_tb IS
END RCA_8bit_tb;

ARCHITECTURE behavior OF RCA_8bit_tb IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT RCA_8bit
        PORT(
            A1 : IN std_logic_vector(7 downto 0);
            B1 : IN std_logic_vector(7 downto 0);
            cin1 : IN std_logic;
            enable : IN std_logic;
            clock : IN std_logic;
            S1 : OUT std_logic_vector(7 downto 0);
            cout1 : OUT std_logic
        );
    END COMPONENT;
    -- Inputs
    signal A1 : std_logic_vector(7 downto 0) := (others => '0');
    signal B1 : std_logic_vector(7 downto 0) := (others => '0');
    signal cin1 : std_logic := '0';
    signal enable : std_logic := '0';
    signal clock : std_logic := '0';
```

```

--Outputs
signal S1 : std_logic_vector(7 downto 0);
signal cout1 : std_logic;

-- Clock period definitions
constant clock_period : time := 10 ns;

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: RCA_8bit PORT MAP (
    A1 => A1,
    B1 => B1,
    cin1 => cin1,
    enable => enable,
    clock => clock,
    S1 => S1,
    cout1 => cout1
);

-- Clock process definitions
clock_process :process
begin
    clock <= '0';
    wait for clock_period/2;
    clock <= '1';
    wait for clock_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 5 ns.

    wait for 5 ns;

    -- insert stimulus here

    -- A1 = a7 a6 a5 a4 a3 a2 a1 a0
    -- B1 = b7 b6 b5 b4 b3 b2 b1 b0
    -- S1 = s7 s6 s5 s4 s3 s2 s1 s0
    -- cin1
    -- cout1

    A1          <= "00000000";
    B1          <= "00000000";
    cin1        <= '0';
    enable      <= '1';
    wait for 9 ns;

    assert S1 = "00000000" AND cout1 = '0'
    report "errore0"
    severity failure;

    enable      <= '0';
    wait for 1 ns;

    A1          <= "00011000";
    B1          <= "11000000";
    cin1        <= '0';
    wait for 9 ns;

    assert S1 = "00000000" AND cout1 = '0'
    report "errore1"
    severity failure;

    enable <= '1';
    wait for 1 ns;

    wait for 10 ns;

    assert S1 = "11011000" AND cout1 = '0'
    report "errore2"
    severity failure;

    A1          <= "00100001";
    B1          <= "10001000";
    cin1        <= '1';
    wait for 10 ns;

    wait for 1 ns;

    assert S1 = "10101010" AND cout1 = '0'

```

```

report "errore3"
severity failure;

wait for 1 ns;

enable <= '0';
wait for 1 ns;

A1      <= "10000001";
B1      <= "10000010";
cin1    <= '1';
wait for 16 ns;

enable <= '1';
wait for 2 ns;

assert S1 = "00000100" AND cout1 = '1'
report "errore4"
severity failure;

wait;
end process;

END;

```

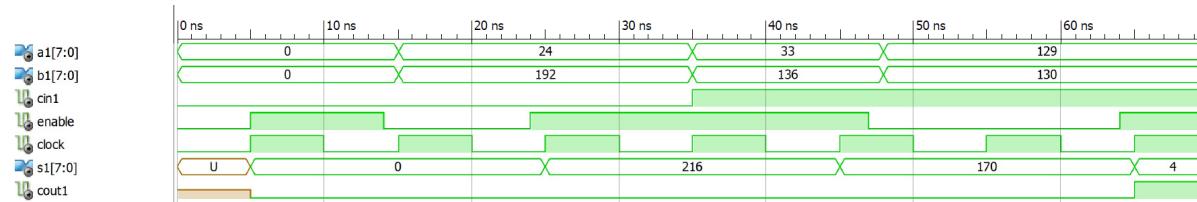


Figura 4.3: Simulazione RCA su 8 bit

Capitolo 5

Esercizio 5

5.1 Traccia

Progettare ed implementare in VHDL un registro a scorrimento circolare di 8 bit utilizzando i flip-flop D edge triggered implementati nell'esercizio 3 (si scelga una delle realizzazioni).

Sintetizzare sulla board il componente utilizzando gli switch per acquisire il dato iniziale, un bottone per acquisire il segnale di shift e i led per visualizzare il contenuto del registro in ogni istante.

5.2 Introduzione

Per la realizzazione del registro a scorrimento circolare è stato seguito un approccio di tipo strutturale. Sono stati prima definiti i componenti di memoria elementari e a partire da essi è stato ottenuto il registro. Si è scelto come componente di base un flip-flop D edge-triggered attivo sul fronte di discesa.

5.3 Soluzione

È stato implementato un registro parallelo-parallelo di 8 bit in grado di operare come registro a scorrimento, seguendo un'architettura multi-livello, come riportato in Figura 5.1. Al livello più alto si trova il *top-level-module*, ovvero il modulo `ShiftRegister (structural)`. Tale componente riceve in ingresso, oltre ai segnali di abilitazione e reset, un valore numerico di 8 bit ed un segnale di caricamento. In questo modo è possibile caricare nel registro il valore desiderato. In uscita viene invece presentato il valore corrente contenuto nel registro. I componenti appartenenti al livello inferiore sono:

- `FF_D (behavioural)`: flip-flop edge-triggered attivo sul fronte di discesa del segnale di clock.
- `mux_2_1 (data-flow)`: multiplexer 2:1, permettono di selezionare tra il caricamento del dato in parallelo e lo scorrimento circolare.

File: ShiftRegister.vhd

Il primo multiplexer riceve come possibile ingresso l'uscita dell'ultimo flip-flop del registro, in modo da realizzare il comportamento circolare. I restanti, ricevono invece l'uscita del flip-flop precedente. Tutti i multiplexer ricevono poi come segnale di abilitazione il segnale di load del registro.

Ogni flip-flop del registro riceve come valore di ingresso l'uscita del rispettivo multiplexer e come segnale di clock il segnale di shift del registro.

Codice Shift Register

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ShiftRegister is
    generic(
        N : POSITIVE := 8
    );
    port(
        X           : in STD_LOGIC_VECTOR(N-1 downto 0);
        ENABLE_SH   : in STD_LOGIC;
        RESET       : in STD_LOGIC;
        LOAD        : in STD_LOGIC;
        Y           : out STD_LOGIC_VECTOR(N-1 downto 0)
    );
end ShiftRegister;

architecture Structural of ShiftRegister is

COMPONENT MUX_2_1_Nbit
GENERIC( N : integer );
PORT (
    X0 : IN STD_LOGIC_VECTOR(N-1 downto 0);
    X1 : IN STD_LOGIC_VECTOR(N-1 downto 0);
    S : IN std_logic;
    Y : OUT STD_LOGIC_VECTOR(N-1 downto 0)
);
END COMPONENT;

component FF_D
port(
    D      : in STD_LOGIC;
    CLK   : in STD_LOGIC;
    RESET : in STD_LOGIC;
    Q     : out STD_LOGIC
);
end component;

signal MUX_OUT: STD_LOGIC_VECTOR(N-1 downto 0);
signal REG_VAL: STD_LOGIC_VECTOR(N-1 downto 0);

begin

mux_2_1_all: for i in 0 to N-1 generate
    m: if i = 0 generate
        most: MUX_2_1_Nbit
        GENERIC MAP ( N => 1 )
        PORT MAP (
            X0(0) => REG_VAL(N-1),
            X1(0) => X(i),
            S => LOAD,
            Y(0) => MUX_OUT(i)
        );
    end generate;
    r: if i > 0 and i <= N-1 generate
        rest: MUX_2_1_Nbit
        GENERIC MAP ( N => 1 )
        PORT MAP (
            X0(0) => REG_VAL(i-1),
            X1(0) => X(i),
            S => LOAD,
            Y(0) => MUX_OUT(i)
        );
    end generate;
end generate;

FF_D_all: for i in 0 to N-1 generate
    FF_D_i: FF_D

```

```

PORT MAP (
    D      => MUX_OUT(i),
    CLK   => ENABLE_SH,
    RESET => RESET,
    Q      => REG_VAL(i)
);
end generate;

Y <= REG_VAL;

end Structural;

```

File: FF_D.vhd

Invece di riutilizzare uno dei dispositivi realizzati nel Capitolo 3, si è scelto di implementare nuovamente il flip-flop D edge-triggered mediante il costrutto process. Il dispositivo è abilitato sul fronte di discesa del segnale di clock, avendo utilizzato la funzione falling_edge.

Si è scelta tale soluzione in modo da rendere il dispositivo sintetizzabile. Difatti, in questo modo il modulo realizzato viene riconosciuto dallo strumento di sintesi e associato al flip-flop D disponibile sulla scheda.

Codice Flip-Flop D Edge-Triggered Discesa

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FF_D is
    Port ( D : in STD_LOGIC;
           CLK : in STD_LOGIC;
           RESET: in STD_LOGIC;
           Q : out STD_LOGIC
    );
end FF_D;

architecture Behavioral of FF_D is

begin
    FF_D_proc : process(CLK, RESET)
    begin
        if RESET = '1' then
            Q <= '0';
        elsif falling_edge(CLK) then
            Q <= D;
        end if;
    end process;
end Behavioral;

```

File: MUX_2_1_Nbit.vhd

Il codice relativo al componente mux_2_1 è uguale a quello riportato nel Paragrafo 1.3.

5.4 Schematici

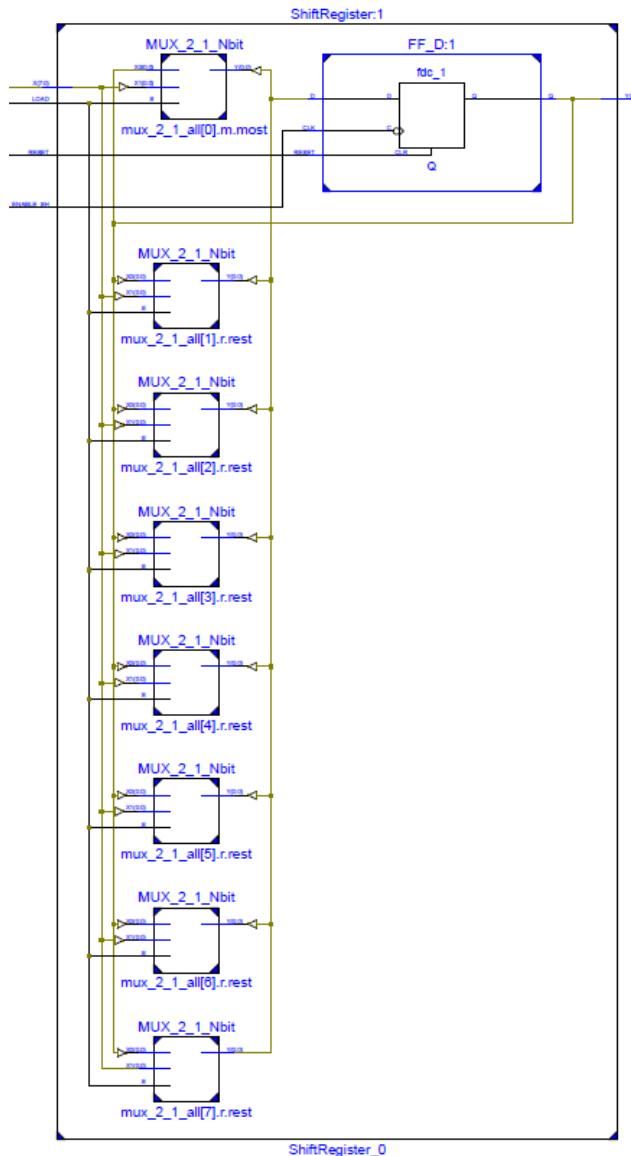


Figura 5.1: Schematico Shift Register

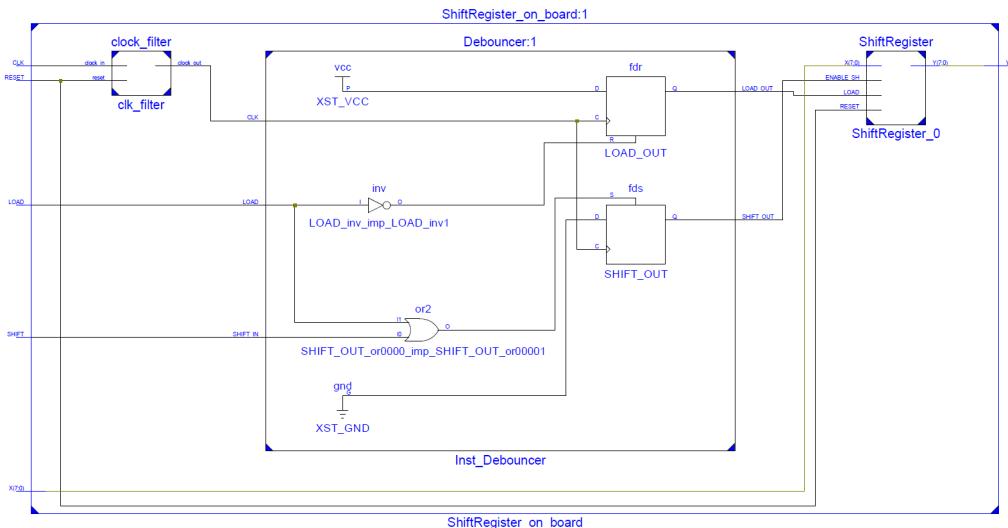


Figura 5.2: Schematico Shift Register On Board

5.5 Sintesi

Come richiesto dalla traccia, è stata effettuata la sintesi del dispositivo sulla board. A tal fine sono stati prodotti ulteriori moduli VHDL.

Il dispositivo sintetizzato sulla board presenta ancora una volta una struttura multi-livello, come riportato in Figura 5.2. Al livello più alto si trova il *top-level-module*, ovvero `ShiftRegister_on_board` (**structural**). Tale componente definisce i segnali di ingresso e uscita del dispositivo e descrive le connessioni tra i componenti appartenenti al livello inferiore:

- **ShiftRegister (structural):** sulla base di quanto esposto in precedenza, rappresenta un registro a scorrimento circolare parallelo-parallelo.
- **clock_filter (behavioural):** componente responsabile della divisione in frequenza del segnale di clock. Preso in ingresso il clock fornito dalla board alla frequenza di 50 MHz, fornisce in uscita un segnale alto dopo un tempo pari a $T_{ck_out} = 10^5 T_{ck_in}$, il che equivale ad avere un segnale alto ogni 10^{-5} s.
È bene osservare che il *duty cycle* del segnale in uscita è pari allo 0.2%.
- **Debouncer (behavioural):** ricevuti in ingresso, oltre che al segnale di clock, i segnali di load e shift, fornisce in uscita due segnali che abilitano l'aggiornamento dei valori del registro. In particolare, se il segnale di load è alto, viene abilitato il caricamento parallelo; viceversa, se il segnale di shift è alto, viene abilitato lo scorrimento circolare.

File: ShiftRegister_on_board.vhd

Si evidenziano di seguito alcune delle connessioni tra i componenti istanziati nel *top-level-module*:

- ShiftRegister: riceve come segnali di shift e caricamento i segnali di shift e load in uscita dal Debouncer. I valori di ingresso e uscita sono invece associati rispettivamente agli switch e ai led presenti sulla board.
- Debouncer: riceve come segnale di clock l'uscita del componente `clock_filter`. I segnali di shift e load in ingresso sono invece associati a due dei pulsanti presenti sulla scheda.

Codice Shift Register On Board

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ShiftRegister_on_board is
    port(   X      : in STD_LOGIC_VECTOR(7 downto 0);
            CLK     : in STD_LOGIC;
            SHIFT   : in STD_LOGIC;
            RESET   : in STD_LOGIC;
            LOAD    : in STD_LOGIC;
            Y       : out STD_LOGIC_VECTOR(7 downto 0)
        );
end ShiftRegister_on_board;

architecture Behavioral of ShiftRegister_on_board is

COMPONENT ShiftRegister
GENERIC(
    N : POSITIVE := 8
);
PORT(
    X          : in STD_LOGIC_VECTOR(N-1 downto 0);
    ENABLE_SH  : in STD_LOGIC;
    RESET      : in STD_LOGIC;
    LOAD       : in STD_LOGIC;
    Y          : out STD_LOGIC_VECTOR(N-1 downto 0)
);
END COMPONENT;

COMPONENT Debouncer
PORT(
    CLK : IN std_logic;
    SHIFT_IN : IN std_logic;
    LOAD : IN std_logic;
    SHIFT_OUT : OUT std_logic;
    LOAD_OUT : OUT std_logic
);
END COMPONENT;

COMPONENT clock_filter
GENERIC(
    clock_frequency_in      : integer := 50000000;
    clock_frequency_out     : integer := 500
);
PORT(
    clock_in     : IN std_logic;
    reset        : in STD_LOGIC;
    clock_out    : OUT std_logic
);
END COMPONENT;

signal SHIFT_TEMP : STD_LOGIC;
signal LOAD_TEMP  : STD_LOGIC;
signal CLOCK_FX   : STD_LOGIC;

begin

    ShiftRegister_0: ShiftRegister
        port map(
            X      => X,
            ENABLE_SH => SHIFT_TEMP,
            RESET      => RESET,
            LOAD       => LOAD_TEMP,
            Y          => Y
        );

    Inst_Debouncer: Debouncer
        port map(
            CLK      => CLOCK_FX,
            SHIFT_IN => SHIFT,
            ...
        );

```

```

LOAD          => LOAD,
SHIFT_OUT     => SHIFT_TEMP,
LOAD_OUT      => LOAD_TEMP
);

clk_filter: clock_filter
PORT MAP(
    clock_in      => CLK,
    reset         => RESET,
    clock_out     => CLOCK_FX
);

end Behavioral;

```

File: clock_filter.vhd

Si è scelto di implementare il componente `clock_filter` mediante un contatore ed un comparatore, come riportato in Figura 5.3.

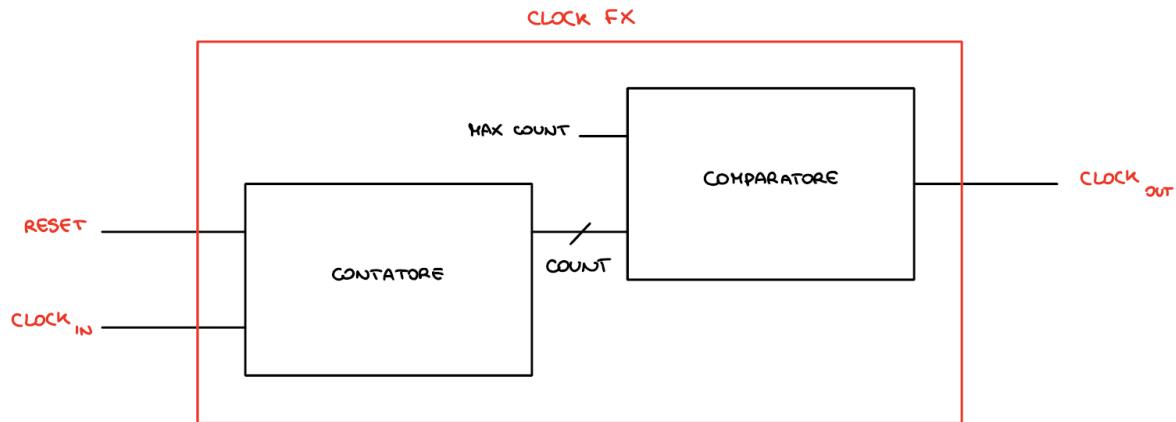


Figura 5.3: Clock Filter Soluzione 1

In alternativa, si potrebbe realizzare lo stesso dispositivo utilizzando un registro a scorrimento, come illustrato in Figura 5.4. Il vantaggio di tale configurazione consiste nella possibilità di poter modificare agevolmente il *duty cycle* del segnale in uscita, modificando opportunamente il contenuto del registro: aumentando il numero di bit alti aumenta di conseguenza il duty cycle.

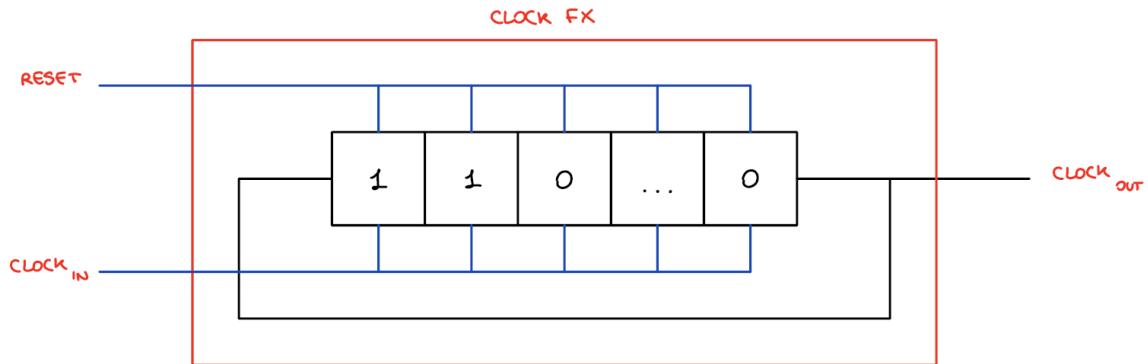


Figura 5.4: Clock Filter Soluzione 2

Codice Clock Filter

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity clock_filter is
    generic(
        clock_frequency_in : integer := 50000000;
        clock_frequency_out : integer := 500
    );
    Port ( clock_in : in STD_LOGIC;
            reset : in STD_LOGIC;
            clock_out : out STD_LOGIC);
end clock_filter;

architecture Behavioral of clock_filter is

    signal clockfx : std_logic := '0';
    constant count_max_value : integer := clock_frequency_in/(clock_frequency_out)-1;
begin
    clock_out <= clockfx;

    count_for_division: process(clock_in, reset)
        variable counter : integer range 0 to count_max_value := 0;
    begin
        if reset = '1' then
            counter := 0;
            clockfx <= '0';
        elsif clock_in'event and clock_in = '1' then
            if counter = count_max_value then
                clockfx <= '1';
                counter := 0;
            else
                clockfx <= '0';
                counter := counter + 1;
            end if;
        end if;
    end process;
end Behavioral;

```

File: Debouncer.vhd

Il componente Debouncer svolge un ruolo fondamentale nella gestione dell'acquisizione dei segnali di ingresso provenienti dai pulsanti della board. In particolare, si fa carico della risoluzione del problema del *bouncing* (rimbalzo), fenomeno riportato in Figura 5.5,

tipico dei pulsanti presenti su una board. Tale componente permette di campionare i segnali associati alla pressione del pulsante solo quando risultano stabili.

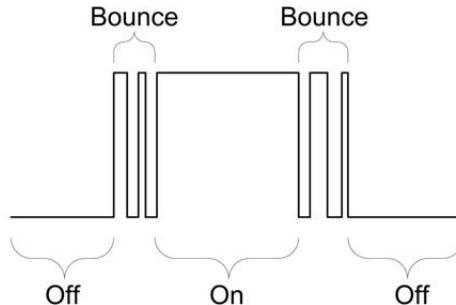


Figura 5.5: Fenomeno del Bouncing

Avendo utilizzato la funzione `rising_edge`, il componente Debouncer è abilitato sul fronte di salita del segnale di clock. Si ricorda che i flip-flop del registro a scorrimento sono invece abilitati sul fronte di discesa del segnale di shift. In questo modo, in caso di caricamento (`LOAD = '1'`), il contenuto del registro sarà aggiornato quando entrambi i segnali di load e shift in uscita dal Debouncer saranno stabili.

Codice Debouncer

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

entity Debouncer is
  Port ( CLK : in STD_LOGIC;
         SHIFT_IN : in STD_LOGIC;
         LOAD : in STD_LOGIC;
         SHIFT_OUT : out STD_LOGIC;
         LOAD_OUT : out STD_LOGIC
        );
end Debouncer;

architecture behavioral of Debouncer is

begin

  cu_proc : process(CLK)
  begin
    if rising_edge(CLK) then
      SHIFT_OUT <= '0';
      LOAD_OUT <= '0';

      if LOAD = '1' then
        LOAD_OUT <= '1';
        SHIFT_OUT <= '1';
      end if;
      if SHIFT_IN = '1' then
        SHIFT_OUT <= '1';
      end if;
    end if;
  end process;

end behavioral;
```

File: Basys_250K.ucf

Codice vincoli board Basys

```
# clock pin for Basys rev E Board
NET "CLK"      LOC = "P54";
# Leds
NET "Y<0>"    LOC = "P15";
NET "Y<1>"    LOC = "P14";
NET "Y<2>"    LOC = "P8";
NET "Y<3>"    LOC = "P7";
NET "Y<4>"    LOC = "P5";
NET "Y<5>"    LOC = "P4";
NET "Y<6>"    LOC = "P3";
NET "Y<7>"    LOC = "P2";
# Switches
NET "X<0>"    LOC = "P38";
NET "X<1>"    LOC = "P36";
NET "X<2>"    LOC = "P29";
NET "X<3>"    LOC = "P24";
NET "X<4>"    LOC = "P18";
NET "X<5>"    LOC = "P12";
NET "X<6>"    LOC = "P10";
NET "X<7>"    LOC = "P6";
# Buttons
NET "RESET"     LOC = "P69";
NET "LOAD"       LOC = "P48";
NET "SHIFT"      LOC = "P47";
```

5.6 Simulazione

File: ShiftRegister_on_board_tb.vhd

Codice Testbench Shift Register On Board

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ShiftRegiste_on_board_tb IS
END ShiftRegiste_on_board_tb;

ARCHITECTURE behavior OF ShiftRegiste_on_board_tb IS
-- Component Declaration for the Unit Under Test (UUT)
COMPONENT ShiftRegister_on_board
PORT(
    X : IN std_logic_vector(7 downto 0);
    CLK : IN std_logic;
    SHIFT : IN std_logic;
    RESET : IN std_logic;
    LOAD : IN std_logic;
    Y : OUT std_logic_vector(7 downto 0)
);
END COMPONENT;

--Inputs
signal X : std_logic_vector(7 downto 0) := (others => '0');
signal CLK : std_logic := '0';
signal SHIFT : std_logic := '0';
signal RESET : std_logic := '0';
signal LOAD : std_logic := '0';

--Outputs
signal Y : std_logic_vector(7 downto 0);

-- Clock period definitions
constant CLK_period : time := 20 ns;

BEGIN
```

```
-- Instantiate the Unit Under Test (UUT)
uut: ShiftRegister_on_board PORT MAP (
    X => X,
    CLK => CLK,
    SHIFT => SHIFT,
    RESET => RESET,
    LOAD => LOAD,
    Y => Y
);

-- Clock process definitions
CLK_process :process
begin
    CLK <= '0';
    wait for CLK_period/2;
    CLK <= '1';
    wait for CLK_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100 ns.
    wait for 100 ns;

    wait for CLK_period*10;

    -- insert stimulus here
    X <= "00011100";
    LOAD <= '1';

    wait for 20 ms;

    LOAD <= '0';

    wait for 20 ms;

    RESET <= '1';

    wait for 20 ms;

    RESET <= '0';

    wait;
end process;

END;
```

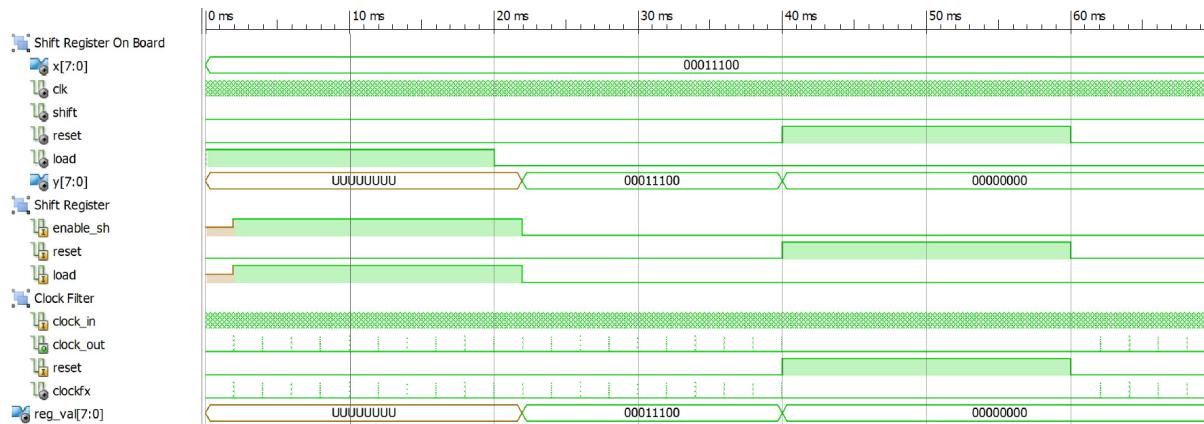


Figura 5.6: Simulazione Shift Register On Board

5.7 High Level Synthesis

Dopo aver realizzato il dispositivo specificato dalla traccia si è deciso di approfondire lo studio dei registri seguendo un approccio alternativo. Esistono infatti strumenti alternativi al VHDL, detti di **High Level Synthesis**, che permettono di descrivere architetture hardware secondo un livello di astrazione più alto. Il vantaggio principale di tali strumenti consiste nel facilitare la realizzazione di prototipi, rispetto ai tradizionali linguaggi di descrizione dell'hardware. Tuttavia, inevitabilmente si hanno anche alcuni svantaggi, tra cui la perdita del totale controllo dei componenti.

Per lo sviluppo dei registri è stato utilizzato **Simulink**, il quale permette di descrivere sistemi hardware attraverso la realizzazione di modelli e di simularne il comportamento. Nello specifico, è stato utilizzato il plug-in **HDL Coder**, il quale consente di gestire l'intero ciclo di sviluppo, a partire dalla fase di progettazione fino a quella di sintesi su uno specifico dispositivo. Inoltre, HDL Coder mette a disposizione numerosi componenti elementari, tra cui porte logiche e registri.

5.7.1 Registro Serie-Serie

Il primo dispositivo realizzato è un Registro Serie-Serie di n^1 bit, il quale riceve e restituisce bit in serie. Evidentemente, il processo di sviluppo nel caso di sintesi di alto livello è diverso rispetto quello tradizionale.

Il primo passo ha previsto la definizione del componente da realizzare, rappresentato in verde in Figura 5.7. Tutto ciò che si trova all'esterno del componente è associabile ad un testbench e viene quindi utilizzato per testare il dispositivo. In particolare sono stati utilizzati i seguenti componenti di libreria:

- Pulse Generator: blocchi utilizzati per generare i segnali di ingresso.
- Sum: blocco utilizzato per sommare i singoli impulsi, in modo da ottenere segnali più complessi.
- Data Type Conversion: blocco in grado di convertire i segnali dal tipo `double` al tipo `boolean`.
- Scope: blocco utilizzato per visualizzare i segnali e verificare il corretto funzionamento del dispositivo.

¹Si è supposto $n = 4$.

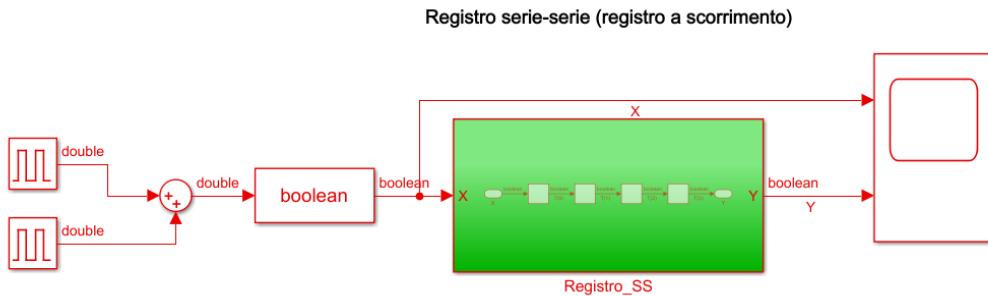


Figura 5.7: Testbench Registro Serie-Serie

Successivamente, è stata descritta l'effettiva architettura del componente. A tale scopo è stato seguito un approccio di tipo strutturale, realizzando quindi il registro come composizione di Flip-Flop D Edge Triggered attivi sul fronte di salita, opportunamente connessi. Come si evince dalla Figura 5.8, il blocco Simulink corrispondente ad un Flip-Flop D ETs è il blocco Unit Delay, ovvero un componente che restituisce in uscita il segnale di ingresso ritardato di un tempo pari ad un periodo di clock. Evidentemente, l'ingresso viene fornito al primo flip-flop e viene prelevata l'uscita dell'ultimo.

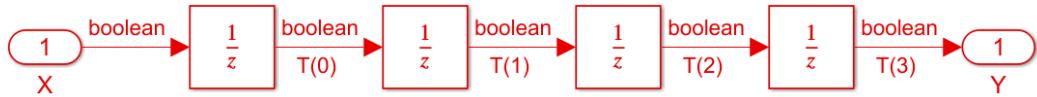


Figura 5.8: Registro Serie-Serie

Attraverso il blocco Scope è possibile verificare il corretto funzionamento del dispositivo. Il segnale di ingresso X viene riportato in uscita dopo n periodi di clock.

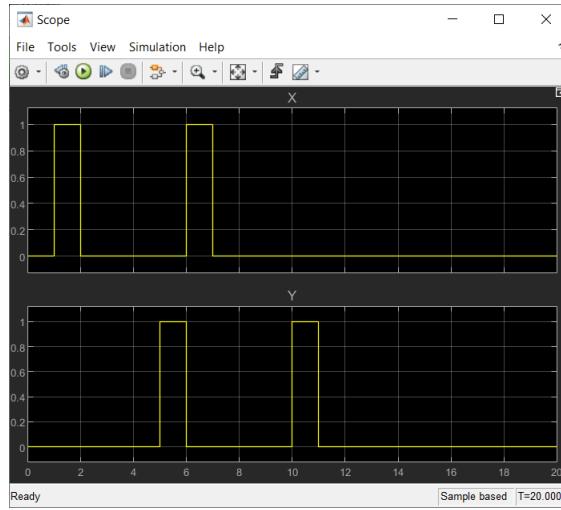


Figura 5.9: Scope Registro Serie-Serie

5.7.2 Registro Serie-Serie Circolare

Seguendo lo stesso approccio è stato realizzato un Registro Serie-Serie Circolare di n bit.

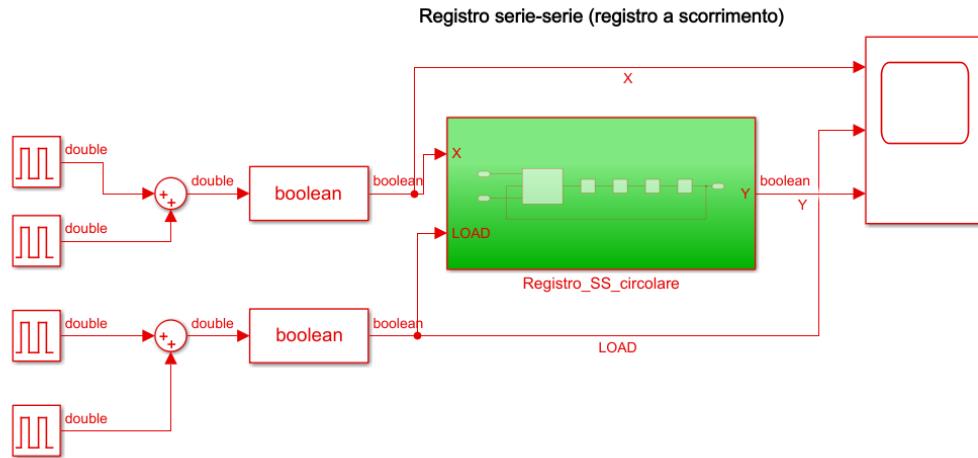


Figura 5.10: Testbench Registro Serie-Serie Circolare

Per realizzare il comportamento di un registro a scorrimento con caricamento, è stato introdotto un blocco aggiuntivo, rappresentato dal multiplexer 2:1 riportato in Figura 5.12. Come si evince da Figura 5.13, se il segnale di LOAD è alto, viene dato in ingresso il valore X, altrimenti, viene riportata in ingresso l'uscita dell'ultimo flip-flop.

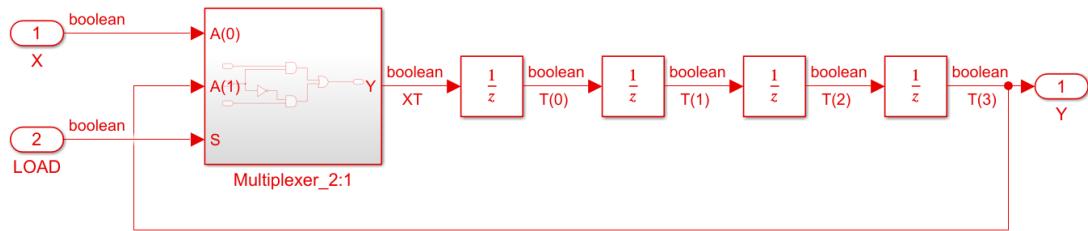


Figura 5.11: Registro Serie-Serie Circolare

Essendo una macchina combinatoria, il multiplexer è stato realizzato utilizzando i componenti di libreria corrispondenti alle tradizionali porte logiche AND-OR-NOT.

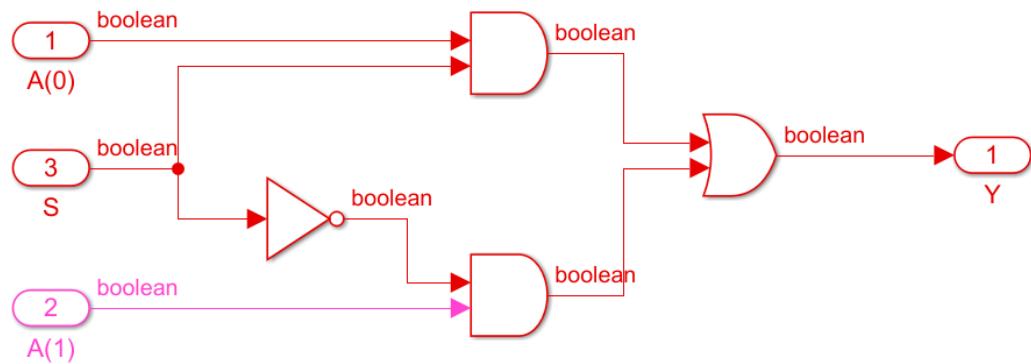


Figura 5.12: Multiplexer 2:1

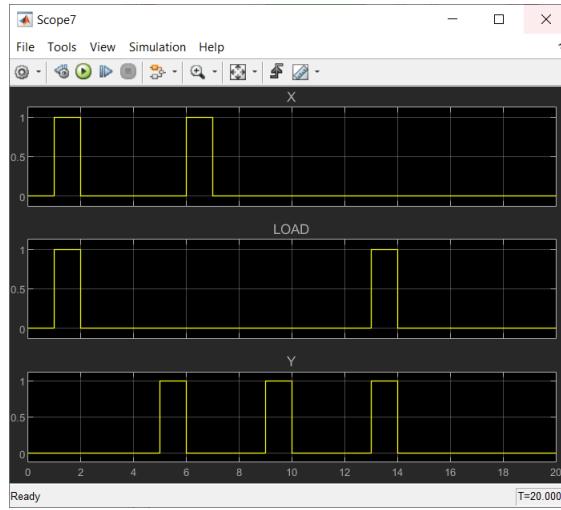


Figura 5.13: Scope Registro Serie-Serie Circolare

5.7.3 Registro Serie-Parallelo

Seguendo lo stesso approccio è stato realizzato un Registro Serie-Parallelo di n bit.

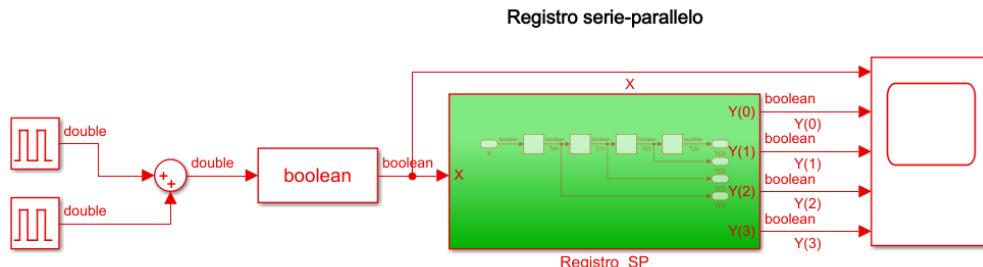


Figura 5.14: Testbench Registro Serie-Parallelo

Lo schema a blocchi del componente è analogo al caso di Registro Serie-Serie, con l'unica differenza di prelevare i segnali in uscita da ciascun flip-flop.

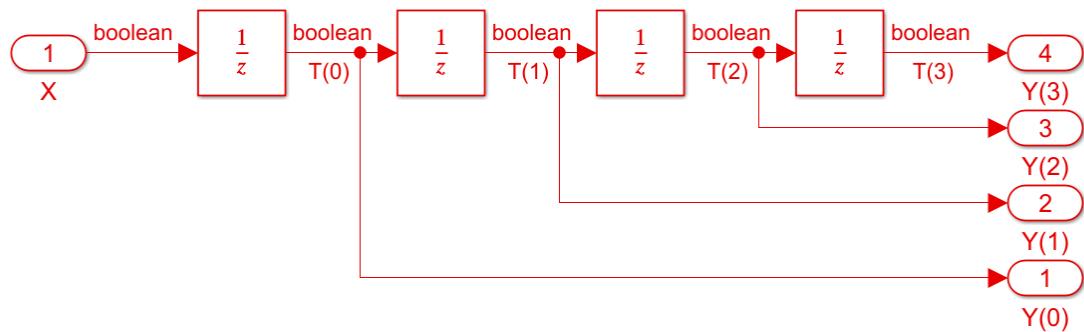


Figura 5.15: Registro Serie-Parallelo

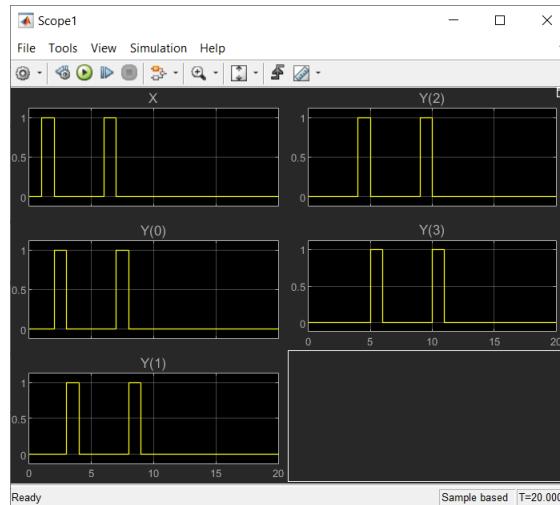


Figura 5.16: Scope Registro Serie-Parallelo

5.7.4 Registro Parallelo-Parallelo

Seguendo lo stesso approccio è stato realizzato un Registro Parallelo-Parallelo di n bit.

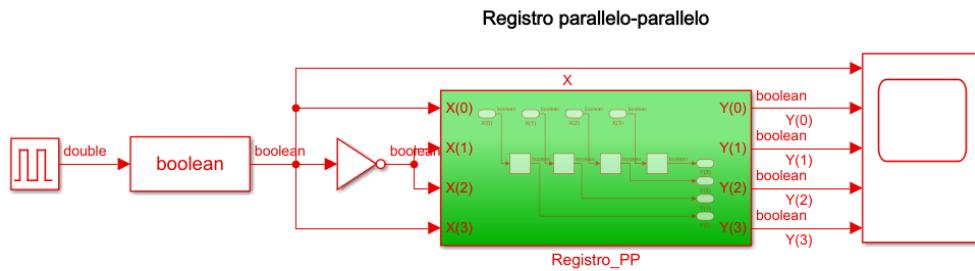


Figura 5.17: Testbench Registro Parallelo-Parallelo

In questo caso, è possibile fornire l'ingresso a ciascun flip-flop del registro e, ancora una volta, viene prelevata l'uscita da ognuno di essi.

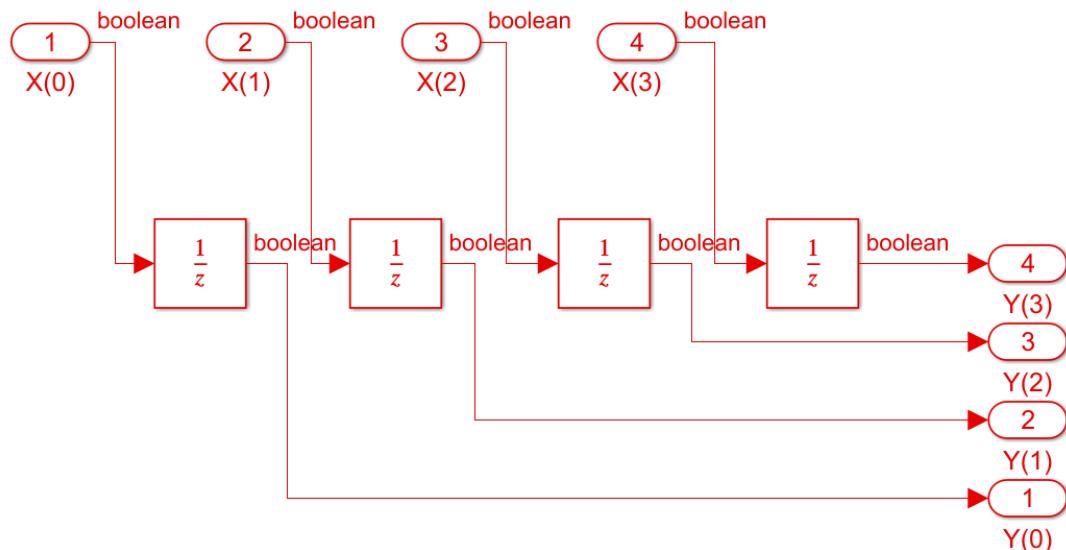


Figura 5.18: Registro Parallelo-Parallelo

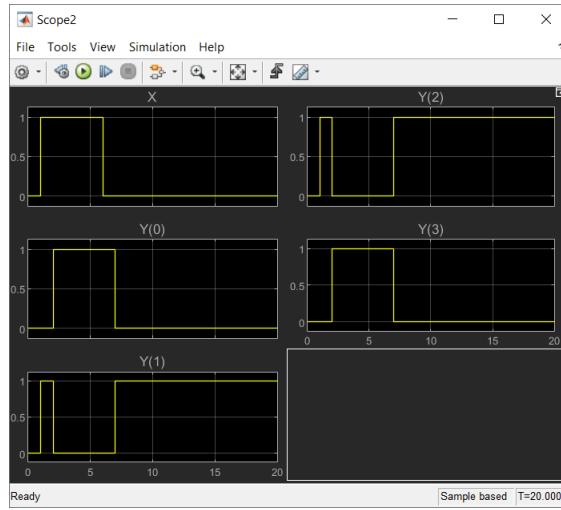


Figura 5.19: Scope Registro Parallelo-Parallelo

5.7.5 Registro Parallelo-Serie

Seguendo lo stesso approccio è stato realizzato un Registro Parallelo-Serie di n bit.

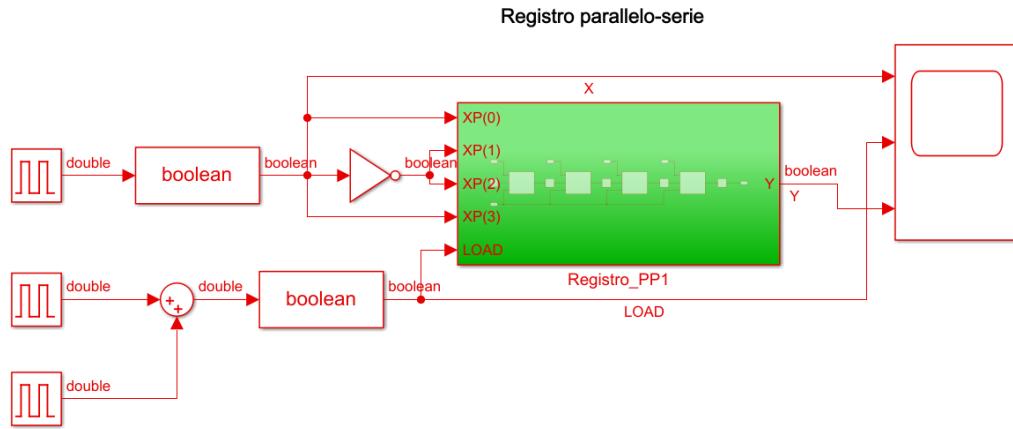


Figura 5.20: Testbench Registro Parallelo-Serie

Per realizzare il caricamento dei dati in parallelo è stato aggiunto un multiplexer per ogni flip-flop. I diversi multiplexer sono pilotati dallo stesso segnale di LOAD e ciascuno di essi permette di selezionare tra un bit del segnale di ingresso XP e l'uscita del flip-flop precedente. Ovviamente, l'uscita viene prelevata esclusivamente dall'ultimo flip-flop.

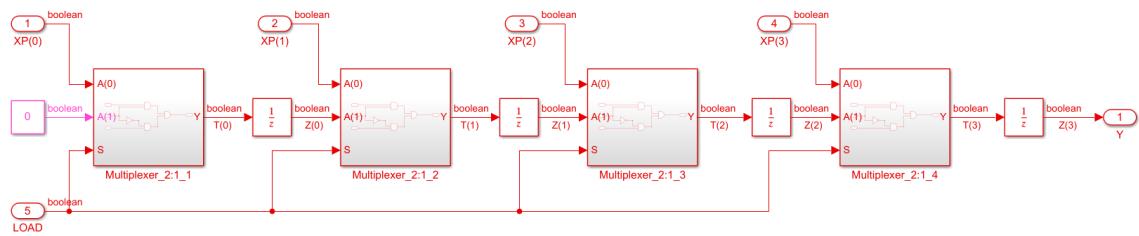


Figura 5.21: Registro Parallelo-Serie

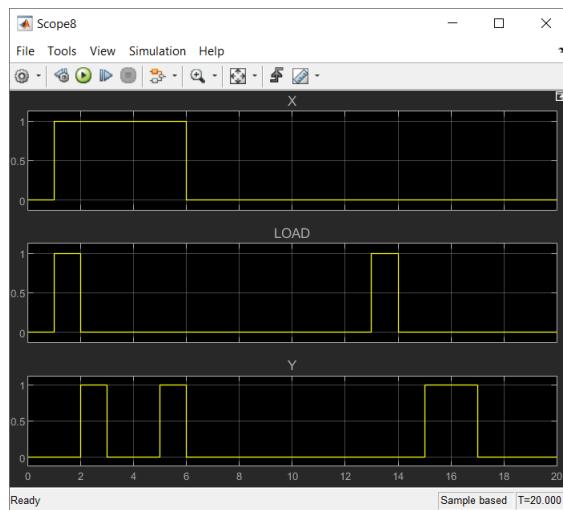


Figura 5.22: Scope Registro Parallelo-Serie

Capitolo 6

Esercizio 6

6.1 Traccia

Progettare ed implementare in VHDL un contatore mod-16 nella modalità serie e parallelo. Sintetizzare sulla board il componente (nella sola modalità serie) utilizzando un bottone per acquisire il segnale di conteggio e 2 cifre del display per visualizzare il contenuto del registro.

6.2 Introduzione

Per la realizzazione dei contatori serie e parallelo è stato seguito un approccio di tipo strutturale. Sono stati prima definiti i componenti elementari e a partire da essi sono stati ottenuti i contatori. Si è scelto in entrambi i casi come componente di base un flip-flop T, in quanto rappresenta il contatore modulo 2. La differenza tra le due soluzioni è legata a come sono connessi i contatori elementari.

6.3 Contatore Seriale

6.3.1 Soluzione

È stato realizzato un contatore modulo 16 seriale utilizzando il livello di astrazione **structural**. L'architettura del dispositivo è composta da 4 flip-flop T edge-triggered attivi sul fronte di discesa, descritti tramite il livello di astrazione **behavioural**.

I diversi flip-flop sono connessi tra di loro come illustrato in Figura 6.1. Il primo riceve in ingresso un segnale di conteggio, i restanti ricevono in ingresso l'uscita del precedente. Ciascun flip-flop è inoltre dotato di un segnale di reset asincrono che permette di inizializzarne il valore.

È bene osservare che in questo tipo di architettura si hanno problemi dovuti a fenomeni di propagazione.

File: Flip_Flop_T_etd.vhd

Codice Flip-Flop T Edge-Triggered Discesa

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Flip_Flop_T_etd is
    Port ( T : in STD_LOGIC;
           Reset : in std_logic;
           Q : out STD_LOGIC
    );
end Flip_Flop_T_etd;

architecture Behavioral of Flip_Flop_T_etd is
    signal Qint : std_logic;
begin
    ffT : process(T, Reset)
    begin
        if( Reset = '1') then
            Qint <= '0';
        elsif(falling_edge(T)) then
            Qint <= not Qint;
        end if;
    end process ffT;
    Q <= Qint;
end Behavioral;

```

File: Contatore_mod16_seriale.vhd

Codice Contatore Modulo 16 Seriale

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Contatore_mod16_seriale is
    generic(n : POSITIVE := 4);
    port(count : in STD_LOGIC;
          Reset : in STD_LOGIC;
          value : out STD_LOGIC_VECTOR(n-1 downto 0)
    );
end Contatore_mod16_seriale;

architecture structural of Contatore_mod16_seriale is

    component Flip_Flop_T_etd
        Port ( T : in STD_LOGIC;
               Reset : in std_logic;
               Q : out STD_LOGIC
        );
    end component;
    signal Qtemp : std_logic_vector (n-1 downto 0);

begin
    Flip_Flop_T_all: for i in 0 to n-1 generate
        l: if i = 0 generate
            least: Flip_Flop_T_etd port map(      T => count,
                                                Reset => Reset,
                                                Q => Qtemp(0)
                                         );
        end generate;
        r: if i > 0 and i <= n-1 generate
            rest: Flip_Flop_T_etd port map(      T => Qtemp(i-1),
                                                Reset => Reset,
                                                Q => Qtemp(i)
                                         );
        end generate;
    end generate;

```

```
value <= Qtemp;
end structural;
```

6.3.2 Schematici

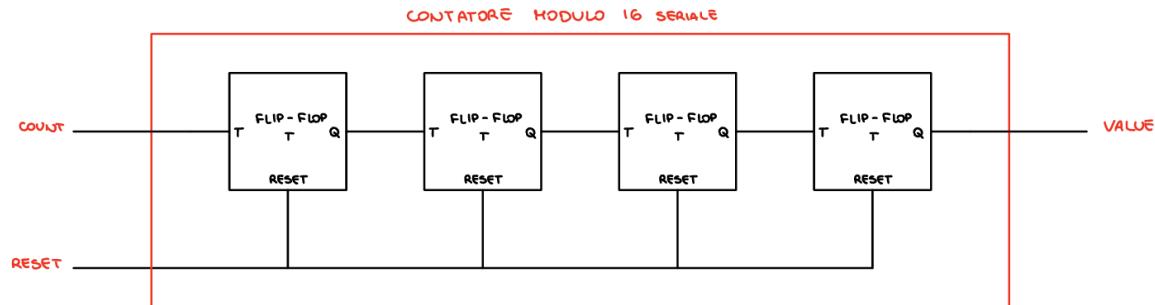


Figura 6.1: Schematico Contatore Modulo 16 Seriale

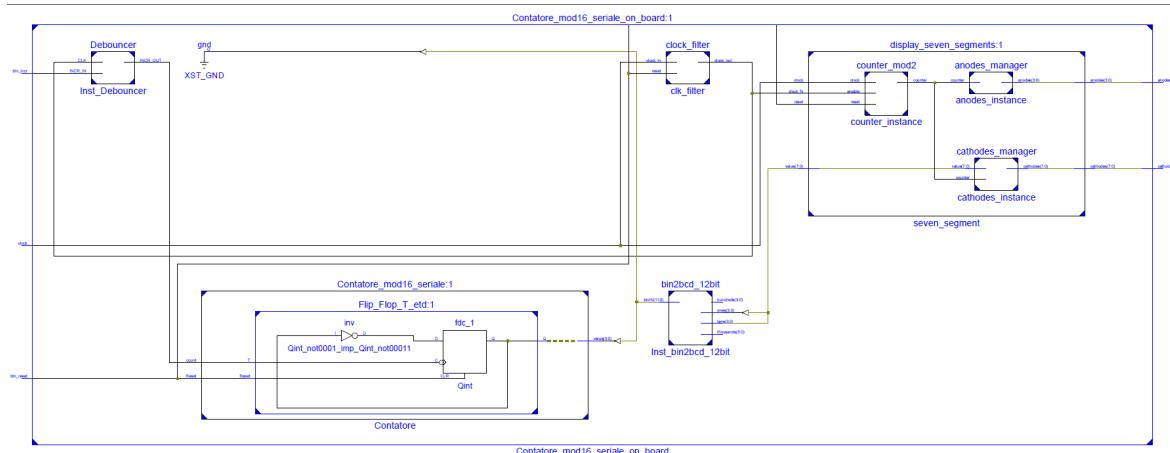


Figura 6.2: Schematico Contatore Modulo 16 Seriale On Board

6.3.3 Sintesi

Come richiesto dalla traccia, è stata effettuata la sintesi del dispositivo sulla board. A tal fine sono stati prodotti ulteriori moduli VHDL.

Il dispositivo sintetizzato sulla board presenta ancora una volta una struttura multi-livello, come riportato in Figura 6.2. Al livello più alto si trova il *top-level-module*, ovvero **Contatore_mod16_seriale_on_board (structural)**. Tale componente definisce i segnali di ingresso e uscita del dispositivo e descrive le connessioni tra i componenti appartenenti al livello inferiore:

- **Contatore_mod16_serial (structural)**: sulla base di quanto esposto in precedenza, rappresenta un contatore seriale
- **clock_filter (behavioural)**: componente responsabile della divisione in frequenza del segnale di clock, del tutto identico a quello descritto nel Paragrafo 5.5.
- **Debouncer (behavioural)**: ricevuti in ingresso i segnali di clock e conteggio, gestisce l'acquisizione del segnale di conteggio proveniente da uno dei pulsanti della board.
Tale componente è analogo a quello descritto nel Paragrafo 5.5.
- **display_seven_segments (structural)**: riceve in ingresso un valore numerico rappresentato su 8 bit e restituisce in uscita le configurazioni di anodi e catodi tali da mostrare su due cifre del display la codifica in *Binary-Coded Decimal* (BCD) del segnale in uscita dal contatore. Nel formato BCD ogni cifra di un numero decimale è codificata in binario su 4 bit. Tale componente è a sua volta costituito da:
 - **cathodes_manager (behavioural)**: componente responsabile della gestione dei catodi.
 - **anodes_manager (behavioural)**: componente responsabile della gestione degli anodi.
 - **counter_mod2 (behavioural)**: contatore modulo 2.

- **bin2bcd_12bit (behavioural)**: ricevuto in ingresso un valore rappresentato in binario su 12 bit ne restituisce in uscita la codifica in formato BCD, eseguendo l'algoritmo *Double Dabble*. Tale componente è stato reperito in rete [2] e, nel contesto dell'esercizio, è stato utilizzato seguendo un approccio *black box*.
La conversione da binario a BCD è necessaria in quanto l'uscita del componente **Contatore_mod16_serial** è codificata in binario, ma il conteggio deve essere visualizzato sul display in formato BCD.

File: Contatore_mod16_serial_on_board.vhd

Si evidenziano di seguito alcune delle connessioni tra i componenti istanziati nel *top-level module*:

- **Contatore_mod16_serial**: riceve come segnale di conteggio il segnale in uscita dal **Debouncer**. Il segnale di uscita viene invece posto in ingresso al componente **bin2bcd_12bit**.
- **Debouncer**: riceve come segnale di clock l'uscita del componente **clock_filter**. Il segnale di conteggio in ingresso è invece associato ad uno dei pulsanti presenti sulla scheda.
- **bin2bcd_12bit**: l'uscita di tale componente viene posta in ingresso al modulo **display_seven_segments**.

Codice Contatore Modulo 16 Seriale On Board

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Contatore_mod16_seriale_on_board is
    Port(
        clock          : in STD_LOGIC;
        btn_reset      : in STD_LOGIC;
        btn_incr       : in STD_LOGIC;
        anodes         : out STD_LOGIC_VECTOR (3 downto 0);
        cathodes       : out STD_LOGIC_VECTOR (7 downto 0)
    );
end Contatore_mod16_seriale_on_board;

architecture Structural of Contatore_mod16_seriale_on_board is

component clock_filter
    generic(
        clock_frequency_in : integer      := 50000000;
        clock_frequency_out : integer      := 500
    );
    Port (
        clock_in       : in STD_LOGIC;
        reset          : in STD_LOGIC;
        clock_out      : out STD_LOGIC);
end component;

COMPONENT Debouncer
PORT(
    CLK : IN std_logic;
    INCR_IN : IN std_logic;
    INCR_OUT : OUT std_logic
);
END COMPONENT;

component Contatore_mod16_seriale
    generic( n : POSITIVE := 4);
    port( count       : in STD_LOGIC;
          Reset       : in STD_LOGIC;
          value       : out STD_LOGIC_VECTOR(n-1 downto 0)
    );
end component;

component display_seven_segments
    Port ( clock          : in STD_LOGIC;
           reset         : in STD_LOGIC;
           value         : in STD_LOGIC_VECTOR (7 downto 0);
           anodes        : out STD_LOGIC_VECTOR (3 downto 0);
           cathodes      : out STD_LOGIC_VECTOR (7 downto 0)
    );
end component;

component bin2bcd_12bit
    Port ( binIN : in STD_LOGIC_VECTOR (11 downto 0);
           ones : out STD_LOGIC_VECTOR (3 downto 0);
           tens : out STD_LOGIC_VECTOR (3 downto 0);
           hundreds : out STD_LOGIC_VECTOR (3 downto 0);
           thousands : out STD_LOGIC_VECTOR (3 downto 0)
    );
end component;

signal clock_fx_temp : std_logic := '0';
signal incr_out_temp : std_logic := '0';
signal value_temp : std_logic_vector (3 downto 0) := (others => '0');
signal value_out_temp : std_logic_vector (7 downto 0) := (others => '0');
signal value_temp_1 : std_logic_vector (11 downto 0) := (others => '0');

begin

seven_segment : display_seven_segments
    Port map( clock      => clock_fx_temp,
              reset      => btn_reset,
              value      => value_out_temp,
              anodes     => anodes,
              cathodes   => cathodes
    );

clk_filter: clock_filter
    Port map (    clock_in      => clock,
                  reset          => btn_reset,
                  clock_out      => clock_fx_temp
    );

```

```

Inst_Debouncer: Debouncer
  PORT MAP (
    CLK => clock_fx_temp,
    INCR_IN => btn_incr,
    INCR_OUT => incr_out_temp
  );
  Contatore : Contatore_mod16_seriale
  Port map(
    count => incr_out_temp,
    Reset => btn_reset,
    value => value_temp
  );
  value_temp_1 <= "00000000" & value_temp;
  Inst_bin2bcd_12bit: bin2bcd_12bit
  PORT MAP(
    binIN => value_temp_1,
    ones => value_out_temp (3 downto 0),
    tens => value_out_temp (7 downto 4)
  );
end Structural;

```

File: Clock_filter.vhd

Il codice relativo al componente *clock_filter* è uguale a quello riportato nel Paragrafo 5.5.

File: Debouncer.vhd**Codice Debouncer**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Debouncer is
  Port ( CLK : in STD_LOGIC;
         INCR_IN : in STD_LOGIC;
         INCR_OUT : out STD_LOGIC
       );
end Debouncer;

architecture behavioral of Debouncer is
begin
  input: process(CLK)
  begin
    if rising_edge(CLK) then
      if INCR_OUT <= '0';
        if INCR_IN = '1' then
          INCR_OUT <= '1';
        end if;
      end if;
    end process input;
  end behavioral;

```

File: seven_segment_array.vhd

Tale componente è analogo a quello descritto nel Paragrafo 2.3, tuttavia presenta alcune differenze dovute al fatto di voler abilitare la visualizzazione su due cifre del display. In aggiunta ai moduli per la gestione di anodi e catodi presenta infatti un contatore modulo 2. Il segnale di conteggio in uscita dal contatore viene utilizzato dai componenti *cathodes_manager* e *anodes_manager* per determinare la cifra del display da visualizzare.

Codice Display 7 Segmenti

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity display_seven_segments is
    Port ( clock      : in STD_LOGIC;
            reset      : in STD_LOGIC;
            value      : in STD_LOGIC_VECTOR (7 downto 0);
            anodes     : out STD_LOGIC_VECTOR (3 downto 0);
            cathodes   : out STD_LOGIC_VECTOR (7 downto 0)
        );
end display_seven_segments;

architecture Structural of display_seven_segments is
    signal counter : std_logic;

COMPONENT counter_mod2
    PORT(
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        counter : out STD_LOGIC
    );
END COMPONENT;

COMPONENT cathodes_manager
    PORT(
        counter      : IN std_logic;
        value       : IN std_logic_vector(7 downto 0);
        cathodes    : OUT std_logic_vector(7 downto 0)
    );
END COMPONENT;

COMPONENT anodes_manager
    PORT(
        counter : IN STD_LOGIC;
        anodes  : OUT std_logic_vector(3 downto 0)
    );
END COMPONENT;

begin
    counter_instance: counter_mod2 port map(
        clock => clock,
        reset => reset,
        counter => counter
    );
    cathodes_instance: cathodes_manager port map(
        counter => counter,
        value => value,
        cathodes => cathodes
    );
    anodes_instance: anodes_manager port map(
        counter => counter,
        anodes => anodes
    );
end Structural;

```

File: Counter_instance.vhd

Ricevuto in ingresso il segnale in uscita dal componente `clock_filter`, realizza il comportamento di un contatore modulo 2, incrementando il conteggio in corrispondenza del fronte di salita.

Codice Contatore Modulo 2

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter_mod2 is
    Port ( clock : in STD_LOGIC;
            reset : in STD_LOGIC;

```

```

        counter : out STD_LOGIC
    );
end counter_mod2;

architecture Behavioral of counter_mod2 is
begin
    signal c : std_logic := '0';

    begin
        counter <= c;
        counter_process: process(clock, reset)
        begin
            if reset = '1' then
                c <= '0';
            elsif rising_edge(clock)then
                c <= not c;
            end if;
        end process;
    end Behavioral;

```

File: cathodes_manager.vhd

Dal momento che si vuole visualizzare un numero intero compreso tra 0 e 15 in codifica BCD vengono abilitate solo due delle quattro cifre disponibili sul display della board. Tale componente si fa quindi carico di selezionare in modo alternato la configurazione dei catodi associata a ciascuna delle due cifre da visualizzare.

Codice Cathodes Manager

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity cathodes_manager is
    Port ( counter      : in STD_LOGIC;
           value         : in STD_LOGIC_VECTOR (7 downto 0);
           cathodes     : out STD_LOGIC_VECTOR (7 downto 0));
end cathodes_manager;

architecture Behavioral of cathodes_manager is

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

alias digit_0 is value (3 downto 0);
alias digit_1 is value (7 downto 4);

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

begin
    digit_switching: process(counter, value)
    begin
        case counter is
            when '0' =>

```

```

        nibble <= digit_0;
      when '1' =>
        nibble <= digit_1;
      when others =>
        nibble <= (others => '0');
    end case;
end process;

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

cathodes <= not '0' & cathodes_for_digit;
end Behavioral;

```

File: anodes_instance.vhd

Dal momento che si vuole visualizzare un numero intero compreso tra 0 e 15 in codifica BCD vengono abilitate solo due delle quattro cifre disponibili sul display della board. Tale componente si fa quindi carico di attivare alternativamente uno dei due anodi associati alle cifre da abilitare, in modo da fornire all'osservatore l'illusione che entrambe le cifre siano accese.

Codice Anodes Manager

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity anodes_manager is
  Port ( counter : in STD_LOGIC;
         anodes : out STD_LOGIC_VECTOR (3 downto 0)
       );
end anodes_manager;

architecture Behavioral of anodes_manager is
  signal anodes_switching : std_logic_vector(3 downto 0) := (others => '0');

```

```

begin
anodes <= not anodes_switching or not "0011";
anodes_process: process(counter)
begin
  case counter is
    when '0' =>
      anodes_switching <= x"1";
    when '1' =>
      anodes_switching <= x"2";
    when others =>
      anodes_switching <= (others => '0');
  end case;
end process;
end Behavioral;

```

File: bin2bcd_12bit.vhd

Codice Convertitore Binario-BCD su 12 bit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity bin2bcd_12bit is
  Port ( binIN : in STD_LOGIC_VECTOR (11 downto 0);
         ones : out STD_LOGIC_VECTOR (3 downto 0);
         tens : out STD_LOGIC_VECTOR (3 downto 0);
         hundreds : out STD_LOGIC_VECTOR (3 downto 0);
         thousands : out STD_LOGIC_VECTOR (3 downto 0)
        );
end bin2bcd_12bit;

architecture Behavioral of bin2bcd_12bit is
begin
bcd1: process(binIN)
  -- temporary variable
  variable temp : STD_LOGIC_VECTOR (11 downto 0);
  -- variable to store the output BCD number
  -- organized as follows
  -- thousands = bcd(15 downto 12)
  -- hundreds = bcd(11 downto 8)
  -- tens = bcd(7 downto 4)
  -- units = bcd(3 downto 0)
  variable bcd : UNSIGNED (15 downto 0) := (others => '0');
  -- by
  -- https://en.wikipedia.org/wiki/Double_dabble

begin
  -- zero the bcd variable
  bcd := (others => '0');

  -- read input into temp variable
  temp(11 downto 0) := binIN;

  -- cycle 12 times as we have 12 input bits
  -- this could be optimized, we do not need to check and add 3 for the
  -- first 3 iterations as the number can never be >4
  for i in 0 to 11 loop
    if bcd(3 downto 0) > 4 then
      bcd(3 downto 0) := bcd(3 downto 0) + 3;
    end if;

    if bcd(7 downto 4) > 4 then
      bcd(7 downto 4) := bcd(7 downto 4) + 3;
    end if;

    if bcd(11 downto 8) > 4 then
      bcd(11 downto 8) := bcd(11 downto 8) + 3;
    end if;
  end loop;
end process;
end Behavioral;

```

```
-- thousands can't be >4 for a 12-bit input number
-- so don't need to do anything to upper 4 bits of bcd
-- shift bcd left by 1 bit, copy MSB of temp into LSB of bcd
bcd := bcd(14 downto 0) & temp(11);
-- shift temp left by 1 bit
temp := temp(10 downto 0) & '0';
end loop;
-- set outputs
ones <= STD_LOGIC_VECTOR(bcd(3 downto 0));
tens <= STD_LOGIC_VECTOR(bcd(7 downto 4));
hundreds <= STD_LOGIC_VECTOR(bcd(11 downto 8));
thousands <= STD_LOGIC_VECTOR(bcd(15 downto 12));
end process bcd1;
end Behavioral;
```

6.3.4 Simulazione

File: Contatore_mod16_seriale_tb.vhd

Codice Testbench Contatore Modulo 16 seriale

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Contatore_mod16_seriale_tb IS
END Contatore_mod16_seriale_tb;

ARCHITECTURE behavior OF Contatore_mod16_seriale_tb IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT Contatore_mod16_seriale
        PORT (
            count : IN      std_logic;
            Reset : IN      std_logic;
            value : OUT     std_logic_vector(3 downto 0)
        );
    END COMPONENT;
    --Inputs
    signal count : std_logic := '0';
    signal Reset : std_logic := '0';
    --Outputs
    signal value : std_logic_vector(3 downto 0);
    constant count_period : time := 10 ns;
BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: Contatore_mod16_seriale PORT MAP (
        count => count,
        Reset => Reset,
        value => value
    );
    -- Clock process definitions
    count_process :process
    begin
        count <= '0';
        wait for count_period/2;
        count <= '1';
        wait for count_period/2;
    end process;
    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 15 ns.
        wait for 15 ns;
        Reset <= '1';
    end process;
```

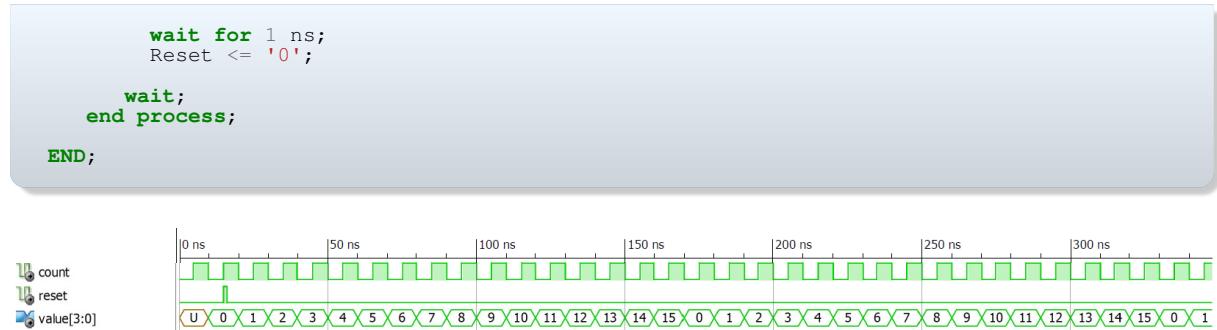


Figura 6.3: Simulazione Contatore Modulo 16 Seriale

6.4 Contatore Parallello

6.4.1 Soluzione

È stato realizzato un contatore modulo 16 parallelo utilizzando il livello di astrazione **structural**. L'architettura del dispositivo è composta da 4 Flip Flop T edge-triggered attivi sul fronte di discesa, descritti tramite il livello di astrazione **behavioural**. I diversi flip-flop sono connessi tra di loro come illustrato in Figura 6.4. Tutti i flip-flop ricevono in ingresso un segnale in uscita da una AND tra il segnale di conteggio e le uscite dei flip-flop precedenti. Ciascun flip-flop è inoltre dotato di un segnale di reset asincrono che permette di inizializzarne il valore.

È bene osservare che in questo tipo di architettura **non** si hanno problemi dovuti a fenomeni di propagazione.

File: Flip_Flop_T_ etd.vhd

Il codice relativo al componente `Flip_Flop_T` è uguale a quello riportato nel Paragrafo 6.3.1.

File: Contatore_mod16_parallello.vhd

Codice Contatore Modulo 16 Parallello

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Contatore_mod16_parallello is
  port(count      : in STD_LOGIC;
       Reset      : in STD_LOGIC;
       value      : out STD_LOGIC_VECTOR(3 downto 0)
      );
end Contatore_mod16_parallello;

architecture Structural of Contatore_mod16_parallello is
  component Flip_Flop_T_ etd
    Port ( T      : in STD_LOGIC;
           Reset   : in std_logic;
           Q      : out STD_LOGIC
          );
  end component;

  signal Temp : std_logic_vector(2 downto 0) := (others => '0');
  signal Qtemp : std_logic_vector(3 downto 0) := (others => '0');

```

```

begin
    Temp(0) <= count and Qtemp(0);
    Temp(1) <= Temp(0) and Qtemp(1);
    Temp(2) <= Temp(1) and Qtemp(2);

    FF0: Flip_Flop_T_etd port map(      T => count,
                                            Reset => Reset,
                                            Q => Qtemp(0)
                                              );
    FF1: Flip_Flop_T_etd port map(      T => Temp(0),
                                            Reset => Reset,
                                            Q => Qtemp(1)
                                              );
    FF2: Flip_Flop_T_etd port map(      T => Temp(1),
                                            Reset => Reset,
                                            Q => Qtemp(2)
                                              );
    FF3: Flip_Flop_T_etd port map(      T => Temp(2),
                                            Reset => Reset,
                                            Q => Qtemp(3)
                                              );
    value <= Qtemp;
end Structural;

```

6.4.2 Schematici

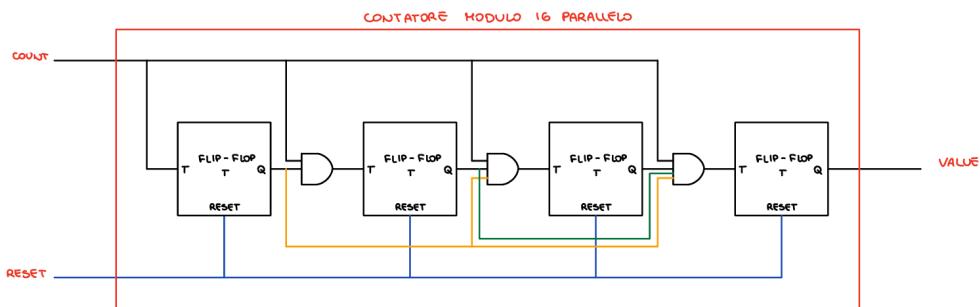


Figura 6.4: Schematico Contatore Modulo 16 Parallelo

6.4.3 Simulazione

File: Contatore_mod16_parallel_tb.vhd

Codice Testbench Contatore Modulo 16 Parallelo

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Contatore_mod16_parallel_tb IS
END Contatore_mod16_parallel_tb;

ARCHITECTURE behavior OF Contatore_mod16_parallel_tb IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT Contatore_mod16_parallel
        PORT(
            count : IN std_logic;
            Reset : IN std_logic;
            value : OUT std_logic_vector(3 downto 0)
        );
    END COMPONENT;
    --Inputs

```

```

signal count : std_logic := '0';
signal Reset : std_logic := '0';

--Outputs
signal value : std_logic_vector(3 downto 0);

constant count_period : time := 10 ns;

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: Contatore_mod16_parallello PORT MAP (
    count => count,
    Reset => Reset,
    value => value
);

-- Clock process definitions
count_process :process
begin
    count <= '0';
    wait for count_period/2;
    count <= '1';
    wait for count_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 10 ns.
    wait for 10 ns;

    Reset <= '1';
    wait for 1 ns;
    Reset <= '0';

    wait;
end process;

END;

```

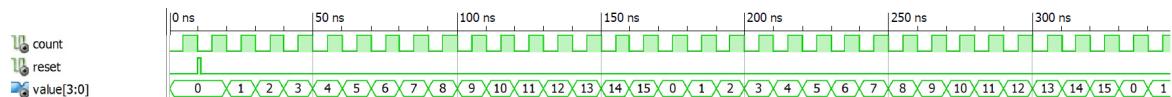


Figura 6.5: Simulazione Contatore Modulo 16 Parallelo

Capitolo 7

Esercizio 7

7.1 Traccia

Progettare ed implementare in VHDL un “arbitro 2 su 3”, ossia un componente che, presi due input binari in ingresso, fornisce in uscita un valore binario pari a quello che compare almeno 2 volte su 3 in ingresso.

Sintetizzare sulla board il componente utilizzando gli switch per acquisire gli ingressi e un led per visualizzare il risultato.

7.2 Introduzione

Data la semplicità dell’arbitro 2 su 3, è stato sufficiente implementare in un singolo modulo VHDL la funzione ingresso-uscita del dispositivo.

7.3 Soluzione

È stato realizzato l’arbitro 2 su 3 utilizzando il livello di astrazione **data-flow**, essendo il livello di astrazione più appropriato per la descrizione di una macchina combinatoria. Di seguito è riportata in Tabella 7.1 la tabella di verità del componente:

Ingressi			Uscite
A1	A2	A3	U
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabella 7.1: Tabella di verità Arbitro 2 su 3

File: arbitro.vhd

Codice Arbitro 2 su 3

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

entity arbitro is
    port ( A1      : in STD_LOGIC;
           A2      : in STD_LOGIC;
           A3      : in STD_LOGIC;
           U       : out STD_LOGIC
        );
end arbitro;

architecture dataflow of arbitro is
begin
    U <= (A2 AND A3) OR (A1 AND (NOT A2) AND A3)
        OR (A1 AND A2 AND (NOT A3));
end dataflow;
```

7.4 Schematici

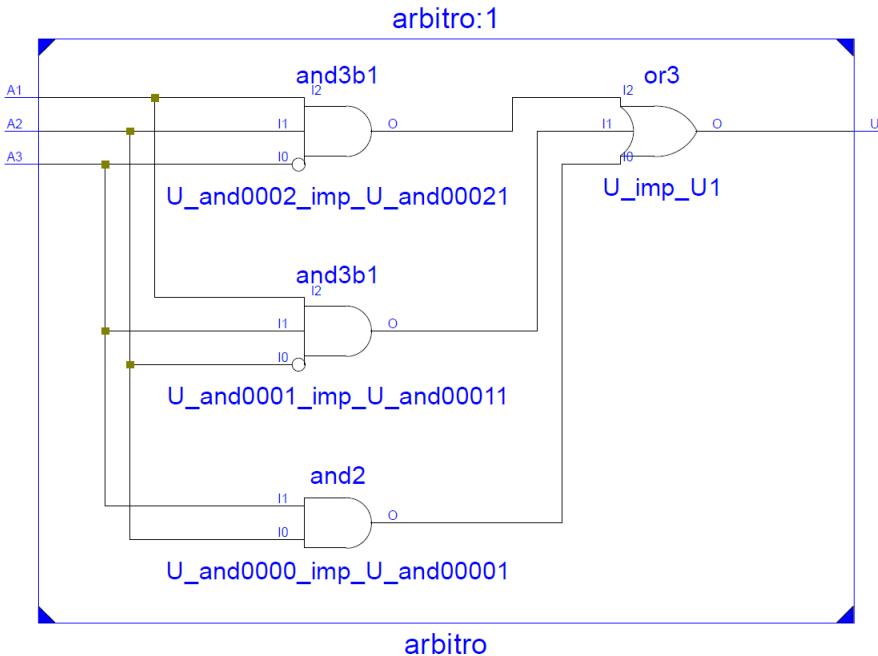


Figura 7.1: Schematico Arbitro 2 su 3

7.5 Sintesi

Come richiesto dalla traccia, è stata effettuata la sintesi del dispositivo sulla board. A tal fine è stato incluso nel progetto un file di configurazione nel quale sono specificati i collegamenti tra i segnali di ingresso-uscita del dispositivo e i diversi componenti della board. In particolare:

- Il segnale di uscita è stato associato ad uno dei led disponibili.
- I tre segnali di ingresso sono stati associati a tre degli switch presenti sulla scheda.

File: Basys_250K.ucf

Codice vincoli board Basys

```
# Leds
NET "U"      LOC = "P2"; # LED7

# Switches
NET "A1"      LOC = "P38"; # SWITCH0
NET "A2"      LOC = "P36"; # SWITCH1
NET "A3"      LOC = "P29"; # SWITCH2
```

7.6 Simulazione

File: arbitro_tb.vhd

Codice Testbench Arbitro 2 su 3

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity arbitro_tb is
end arbitro_tb;

architecture behavioral of arbitro_tb is

component arbitro is
port( A1 : in STD_LOGIC;
      A2 : in STD_LOGIC;
      A3 : in STD_LOGIC;
      U : out STD_LOGIC
    );
end component;

signal a1 : STD_LOGIC:='0';
signal a2 : STD_LOGIC:='0';
signal a3 : STD_LOGIC:='0';
signal u: STD_LOGIC:='0';

begin

uut: arbitro Port map(
    A1 => a1,
    A2 => a2,
    A3 => a3,
    U => u
  );

stim_proc: process
begin

  wait for 10 ns;
  a1 <= '1';
  a2 <= '1';
  a3 <='0';
  wait for 20 ns;

  assert u='1'
  report "errore0"
  severity failure;

  a1 <= '0';
  a2 <= '1';
  a3 <='0';
  wait for 20 ns;

  assert u='0'
  report "errore1"

end process;
end;
```

```
severity failure;  
  
a1 <= '1';  
a2 <= '0';  
a3 <='0';  
wait for 20 ns;  
  
assert u='0'  
report "errore2"  
severity failure;  
wait;  
end process;  
end;
```

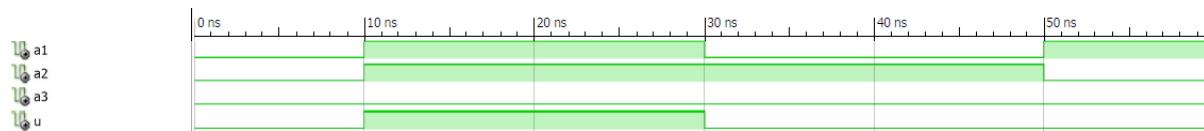


Figura 7.2: Simulazione Arbitro 2 su 3

Capitolo 8

Esercizio 8

8.1 Traccia

Progettare ed implementare un orologio che, a partire da un clock di riferimento che opera da base dei tempi di adeguata precisione, genera mediante uso di contatori il secondo, il minuto e l'ora.

L'orologio deve essere sintetizzato su FPGA e la visualizzazione dell'ora deve sfruttare le 4 cifre del display e i led messi a disposizione dalla board di sviluppo, secondo la seguente modalità:

- I minuti (da 1 a 60) e le ore (da 1 a 24) sono visualizzati in formato 8-4-2-1¹ mediante le due cifre rispettivamente meno e più significative del display a 4 cifre a sette segmenti.
- I secondi (da 1 a 60) sono visualizzati utilizzando i quattro² led di peso meno significativo dell'array di led presenti nella scheda

Il valore del tempo deve poter essere inizializzato acquisendo, in sequenza e tramite gli switch, i valori dell'ora, dei minuti e dei secondi.

8.2 Introduzione

Per la realizzazione di un orologio è stato seguito un approccio di tipo strutturale. Sono stati prima definiti i componenti elementari, contatori modulo M con caricamento, e a partire da essi è stato ottenuto l'orologio.

8.3 Soluzione

È stato realizzato un orologio seguendo un'architettura multi-livello come riportato in Figura 8.1. Al livello più alto si trova il *top-level-module*, ovvero il modulo `Digital_watch (structural)`. Tale componente riceve in ingresso, oltre ai segnali di clock e reset, tre

¹Codifica *Binary-Coded Decimal* (BCD)

²I secondi sono stati codificati in binario su 6 led.

segnali di caricamento e tre valori, associati rispettivamente a secondi, minuti ed ore. In questo modo è possibile impostare l'orario desiderato. In uscita sono invece riportati i valori correnti di secondi, minuti ed ore. I componenti appartenenti al livello inferiore sono:

- **clock_filter (behavioural)**: componente responsabile della divisione in frequenza del segnale di clock, analogo a quello descritto nel Paragrafo 5.5.
Tale componente viene utilizzato come base dei tempi, fornendo in uscita un segnale avente un periodo pari ad 1 secondo.
- **Counter_mod_M_load (behavioural)**: componente in grado di effettuare il conteggio modulo M con caricamento.

All'interno del top-level-module sono istanziati tre contatori, in modo che i primi due, associati a secondi e minuti, contino modulo 60, mentre l'ultimo, associato alle ore, conti modulo 24.

I tre contatori sono organizzati secondo un'**architettura seriale**. Il contatore dei secondi incrementa il conteggio sul fronte di salita del segnale in uscita dal `clock_filter` e, raggiunto il valore massimo, fornisce in uscita un segnale alto. I contatori di minuti e ore incrementano invece il conteggio sul fronte di salita del segnale di uscita del contatore precedente.

File: Digital_watch.vhd

Codice Digital Watch

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Digital_watch is
    Generic( clock_frequency_in : integer := 50000000;
              clock_frequency_out : integer := 1
            );
    Port ( clock          : in STD_LOGIC;
           reset         : in STD_LOGIC;
           load_sec      : in STD_LOGIC;
           load_min      : in STD_LOGIC;
           load_h        : in STD_LOGIC;
           sec_in        : in STD_LOGIC_VECTOR (5 downto 0);
           min_in        : in STD_LOGIC_VECTOR (5 downto 0);
           h_in          : in STD_LOGIC_VECTOR (4 downto 0);
           sec_out       : out STD_LOGIC_VECTOR (5 downto 0);
           min_out       : out STD_LOGIC_VECTOR (5 downto 0);
           h_out          : out STD_LOGIC_VECTOR (4 downto 0));
end Digital_watch;

architecture Structural of Digital_watch is

component Counter_mod_M_load
    generic(   M : integer;
               N : integer
            );
    port( clock          : in STD_LOGIC;
          reset         : in STD_LOGIC;
          load          : in STD_LOGIC;
          data          : in STD_LOGIC_VECTOR (N-1 downto 0);
          y             : out STD_LOGIC_VECTOR (N-1 downto 0);
          enable_out    : out STD_LOGIC
        );
end component;

COMPONENT clock_filter
    GENERIC(
              clock_frequency_in : integer;
              clock_frequency_out : integer
            );
    PORT(

```

```

        clock_in : IN std_logic;
        reset : in STD_LOGIC;
        clock_out : OUT std_logic
    );
END COMPONENT;

signal enable_sec : STD_LOGIC;
signal enable_min : STD_LOGIC;

signal clock_min : STD_LOGIC;
signal clock_h : STD_LOGIC;

signal clock_fx : STD_LOGIC;

begin

clk_filter: clock_filter
    GENERIC MAP(
        clock_frequency_in => clock_frequency_in,
        clock_frequency_out => clock_frequency_out
    )
    PORT MAP(
        clock_in      => clock,
        reset         => reset,
        clock_out     => clock_fx
    );
    clock_min      <= enable_sec;
    clock_h        <= enable_min;

sec_counter: Counter_mod_M_load
    generic map( M => 60,
                 N => 6
    )
    port map(      clock          => clock_fx,
                   reset          => reset,
                   load           => load_sec,
                   data           => sec_in,
                   y              => sec_out,
                   enable_out     => enable_sec
    );
    min_counter: Counter_mod_M_load
    generic map( M => 60,
                 N => 6
    )
    port map(      clock          => clock_min,
                   reset          => reset,
                   load           => load_min,
                   data           => min_in,
                   y              => min_out,
                   enable_out     => enable_min
    );
    h_counter: Counter_mod_M_load
    generic map( M => 24,
                 N => 5
    )
    port map(      clock          => clock_h,
                   reset          => reset,
                   load           => load_h,
                   data           => h_in,
                   y              => h_out
    );
end Structural;

```

File: clock_filter.vhd

Il codice relativo al componente `clock_filter` è uguale a quello riportato nel Paragrafo 5.5.

File: Counter_mod_M_load.vhd

L'utilizzo del costrutto `generic` ha permesso di definire un unico modulo contatore. Evidentemente, il valore dei parametri generici è diverso a seconda del contatore

istanziato.

Il contatore realizzato incrementa il conteggio sul fronte di salita del segnale di clock e permette inoltre caricamento e reset asincroni. In uscita viene riportato, oltre al valore di conteggio, un segnale alto per un periodo di clock nel momento in cui viene raggiunto il valore di conteggio massimo. Tale segnale viene utilizzato nell'architettura complessiva come abilitazione per il contatore successivo.

Codice Contatore Modulo M con Caricamento

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Counter_mod_M_load is
    generic( M : integer := 60;
             N : integer := 6
            );
    port( clock      : in STD_LOGIC;
          reset      : in STD_LOGIC;
          load       : in STD_LOGIC;
          data       : in STD_LOGIC_VECTOR (N-1 downto 0);
          y          : out STD_LOGIC_VECTOR (N-1 downto 0);
          enable_out : out STD_LOGIC
        );
end Counter_mod_M_load;

architecture Behavioral of Counter_mod_M_load is

begin
count: process(clock, reset, load, data)
begin
    if(reset = '1') then
        ty      <= (others => '0');
        enable_out <= '0';
    elsif(load = '1') then
        if(conv_integer(data) > M-1) then
            ty      <= std_logic_vector(to_unsigned(M-1, ty' length));
        else
            ty      <= data;
        end if;
        enable_out <= '0';
    elsif(rising_edge(clock)) then
        if(ty = std_logic_vector(to_unsigned(M-1, ty' length))) then
            ty      <= (others => '0');
            enable_out <= '1';
        else
            ty      <= ty + "1";
            enable_out <= '0';
        end if;
    end if;
end process;

y <= ty;
end Behavioral;

```

8.4 Schematici

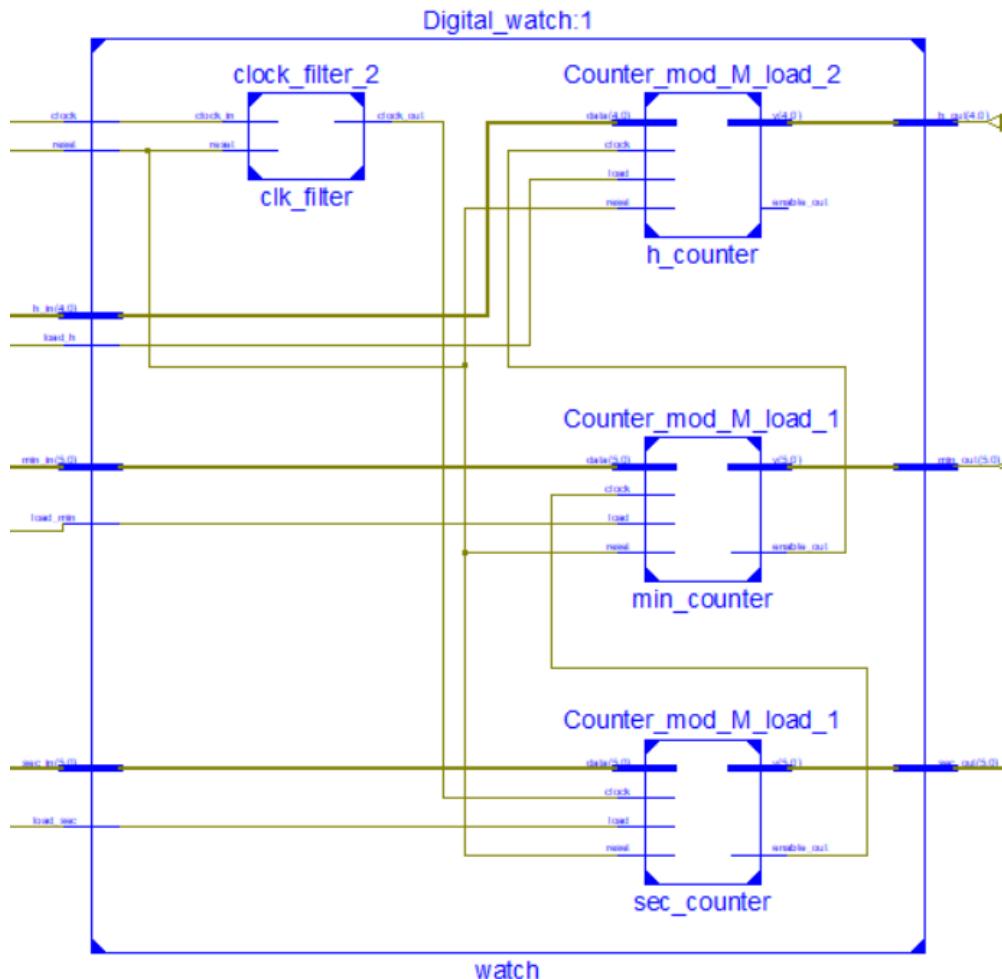


Figura 8.1: Schematico Digital Watch

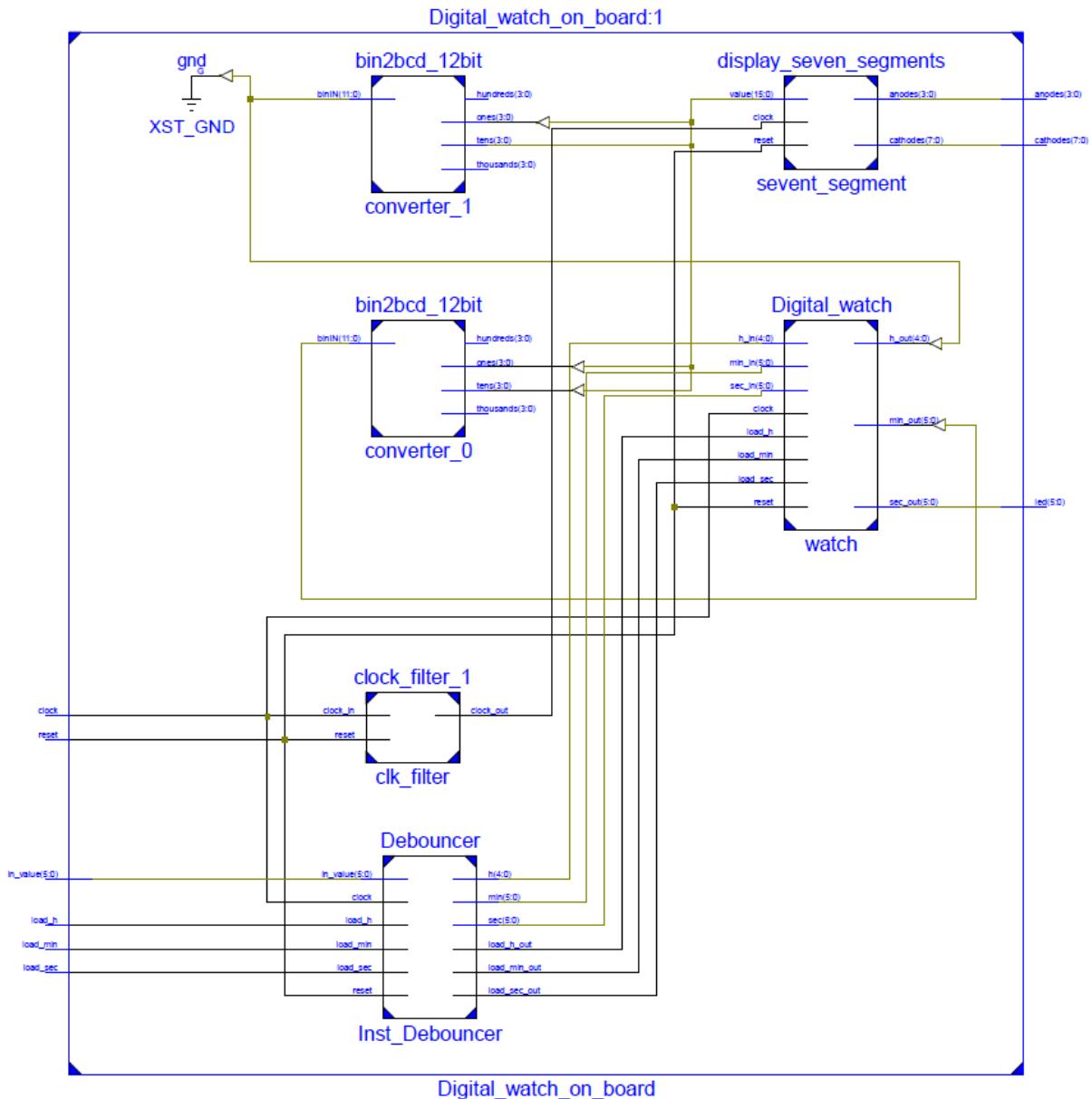


Figura 8.2: Schematico Digital Watch On Board

8.5 Sintesi

Come richiesto dalla traccia, è stata effettuata la sintesi del dispositivo sulla board. A tal fine sono stati prodotti ulteriori moduli VHDL.

Il dispositivo sintetizzato sulla board presenta ancora una volta una struttura multilivello, come riportato in Figura 8.2. Al livello più alto si trova il *top-level-module*, ovvero **Digital_watch_on_board (structural)**. Tale componente definisce i segnali di ingresso e uscita del dispositivo e descrive le connessioni tra i componenti appartenenti al livello inferiore:

- **Digital_watch (structural)**: sulla base di quanto esposto in precedenza, rappresenta un orologio il cui orario può essere all'occorrenza caricato.
- **clock_filter (behavioural)**: componente responsabile della divisione in frequenza del segnale di clock, del tutto identico a quello descritto nel Paragrafo 5.5.
- **Debouncer (behavioural)**: ricevuti in ingresso i segnali di clock, reset e tre segnali di abilitazione alla lettura dagli switch, gestisce l'acquisizione di dei valori di secondi, ore e minuti. In questo modo è possibile impostare l'orario desiderato. Tale componente è analogo a quello descritto nel Paragrafo 5.5.
- **display_seven_segments (structural)**: riceve in ingresso un valore numerico rappresentato su 16 bit e restituisce in uscita le configurazioni di anodi e catodi tali da mostrare su quattro cifre del display la codifica in *Binary-Coded Decimal* (BCD) del valore di ingresso. Tale componente è a sua volta costituito da:
 - **cathodes_manager (behavioural)**: componente responsabile della gestione dei catodi.
 - **anodes_manager (behavioural)**: componente responsabile della gestione degli anodi.
 - **counter_mod4 (behavioural)**: contatore modulo 4.
- **bin2bcd_12bit (behavioural)**: convertitore da binario a BCD, del tutto identico a quello descritto nel Paragrafo 6.3.3.

File: Digital_watch_on_board.vhd

Si evidenziano di seguito alcune delle connessioni tra i componenti istanziati nel *top-level-module*:

- **Digital_watch**: riceve come segnali di caricamento e valori di ingresso per secondi, minuti e ore i segnali in uscita dal Debouncer. Le uscite relative a minuti e ore sono poste in ingresso al componente **bin2bcd_12bit**; le uscite relative ai secondi sono invece associate a sei dei led presenti sulla board.
- **Debouncer**: riceve come segnale di clock l'uscita del componente **clock_filter**. I segnali di caricamento di secondi, minuti e ore sono associati a tre dei pulsanti presenti sulla scheda.
- **bin2bcd_12bit**: l'uscita di tale componente viene posta in ingresso al modulo **display_seven_segments**.

Codice Digital Watch On Board

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

entity Digital_watch_on_board is
  Port ( clock : in STD_LOGIC;
         reset : in STD_LOGIC;
```

```

load_sec      : in STD_LOGIC;
load_min      : in STD_LOGIC;
load_h        : in STD_LOGIC;
in_value      : in STD_LOGIC_VECTOR (5 downto 0);
led           : out STD_LOGIC_VECTOR (5 downto 0);
anodes : out STD_LOGIC_VECTOR (3 downto 0);
cathodes : out STD_LOGIC_VECTOR (7 downto 0)
);
end Digital_watch_on_board;

architecture Behavioral of Digital_watch_on_board is

component display_seven_segments
  Port ( clock : in STD_LOGIC;
         reset : in STD_LOGIC;
         value : in STD_LOGIC_VECTOR (15 downto 0);
         anodes : out STD_LOGIC_VECTOR (3 downto 0);
         cathodes : out STD_LOGIC_VECTOR (7 downto 0)
       );
end component;

COMPONENT Debouncer
PORT(
  clock : IN std_logic;
  reset : IN std_logic;
  load_sec : IN std_logic;
  load_min : IN std_logic;
  load_h : IN std_logic;
  in_value : IN std_logic_vector(5 downto 0);
  sec : OUT std_logic_vector(5 downto 0);
  min : OUT std_logic_vector(5 downto 0);
  h : OUT std_logic_vector(4 downto 0);
  load_sec_out : OUT std_logic;
  load_min_out : OUT std_logic;
  load_h_out : OUT std_logic
);
END COMPONENT;

component Digital_watch is
  Generic( clock_frequency_in : integer;
           clock_frequency_out : integer
         );
  Port ( clock      : in STD_LOGIC;
         reset      : in STD_LOGIC;
         load_sec   : in STD_LOGIC;
         load_min   : in STD_LOGIC;
         load_h     : in STD_LOGIC;
         sec_in     : in STD_LOGIC_VECTOR (5 downto 0);
         min_in     : in STD_LOGIC_VECTOR (5 downto 0);
         h_in       : in STD_LOGIC_VECTOR (4 downto 0);
         sec_out    : out STD_LOGIC_VECTOR (5 downto 0);
         min_out    : out STD_LOGIC_VECTOR (5 downto 0);
         h_out      : out STD_LOGIC_VECTOR (4 downto 0)
       );
end component;

COMPONENT clock_filter
  GENERIC(
    clock_frequency_in : integer;
    clock_frequency_out : integer
  );
  PORT(
    clock_in : IN std_logic;
    reset : in STD_LOGIC;
    clock_out : OUT std_logic
  );
END COMPONENT;

component bin2bcd_12bit is
  Port ( binIN : in STD_LOGIC_VECTOR (11 downto 0);
         ones : out STD_LOGIC_VECTOR (3 downto 0);
         tens : out STD_LOGIC_VECTOR (3 downto 0);
         hundreds : out STD_LOGIC_VECTOR (3 downto 0);
         thousands : out STD_LOGIC_VECTOR (3 downto 0)
       );
end component;

signal clock_fx : STD_LOGIC;
signal value : STD_LOGIC_VECTOR (15 downto 0);
signal temp_sec_in : STD_LOGIC_VECTOR (5 downto 0);
signal temp_min_in : STD_LOGIC_VECTOR (5 downto 0);
signal temp_h_in : STD_LOGIC_VECTOR (4 downto 0);

```

```

signal temp_load_sec : STD_LOGIC;
signal temp_load_min : STD_LOGIC;
signal temp_load_h : STD_LOGIC;

signal temp_sec_out : STD_LOGIC_VECTOR (5 downto 0);
signal temp_min_out : STD_LOGIC_VECTOR (5 downto 0);
signal temp_h_out : STD_LOGIC_VECTOR (4 downto 0);

signal temp_binIN_0 : STD_LOGIC_VECTOR (11 downto 0);
signal temp_binIN_1 : STD_LOGIC_VECTOR (11 downto 0);

begin

sevent_segment: display_seven_segments
port map(
    clock => clock_fx,
    reset => reset,
    value => value,
    anodes => anodes,
    cathodes => cathodes
);

Inst_Debouncer: Debouncer PORT MAP (
    clock => clock,
    reset => reset,
    load_sec => load_sec,
    load_min => load_min,
    load_h => load_h,
    in_value => in_value,
    sec => temp_sec_in,
    min => temp_min_in,
    h => temp_h_in,
    load_sec_out => temp_load_sec,
    load_min_out => temp_load_min,
    load_h_out=> temp_load_h
);

watch: Digital_watch
generic map(
    clock_frequency_in => 50000000,
    clock_frequency_out => 1
)
port map(
    clock => clock,
    reset => reset,
    load_sec => temp_load_sec,
    load_min => temp_load_min,
    load_h => temp_load_h,
    sec_in => temp_sec_in,
    min_in => temp_min_in,
    h_in => temp_h_in,
    sec_out => temp_sec_out,
    min_out => temp_min_out,
    h_out      => temp_h_out
);

clk_filter: clock_filter
generic map(
    clock_frequency_in => 50000000,
    clock_frequency_out => 500
)
PORT MAP (
    clock_in => clock,
    reset => reset,
    clock_out => clock_fx
);

temp_binIN_1 <= "0000000" & temp_h_out;
-- Si concatenano 6 bit '0' essendo il
-- convertitore binario - BCD a 12 bit

converter_1: bin2bcd_12bit
port map (
    binIN => temp_binIN_1,
    ones => value(11 downto 8),
    tens => value(15 downto 12)
);

temp_binIN_0 <= "000000" & temp_min_out;

converter_0: bin2bcd_12bit
port map (
    binIN => temp_binIN_0,
    ones => value(3 downto 0),
    tens => value(7 downto 4)
);

```

```
    );
led <= temp_sec_out;
end Behavioral;
```

File: clock_filter.vhd

Il codice relativo al componente *clock_filter* è uguale a quello riportato nel Paragrafo 5.5.

File: Debouncer.vhd

Codice Debouncer

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

entity Debouncer is
  Port ( clock      : in STD_LOGIC;
         reset      : in STD_LOGIC;
         load_sec   : in STD_LOGIC;
         load_min   : in STD_LOGIC;
         load_h     : in STD_LOGIC;
         in_value   : in STD_LOGIC_VECTOR(5 downto 0);
         sec        : out STD_LOGIC_VECTOR (5 downto 0);
         min        : out STD_LOGIC_VECTOR (5 downto 0);
         h          : out STD_LOGIC_VECTOR (4 downto 0);
         load_sec_out : out STD_LOGIC;
         load_min_out : out STD_LOGIC;
         load_h_out  : out STD_LOGIC
  );
end Debouncer;

architecture behavioral of Debouncer is

signal temp_sec      : STD_LOGIC_VECTOR (5 downto 0) := (others => '0');
signal temp_min      : STD_LOGIC_VECTOR (5 downto 0) := (others => '0');
signal temp_h        : STD_LOGIC_VECTOR (4 downto 0) := (others => '0');

begin

sec <= temp_sec;
min <= temp_min;
h   <= temp_h;

main: process(clock, reset)
begin
  if reset = '1' then
    temp_sec <= (others => '0');
    temp_min <= (others => '0');
    temp_h   <= (others => '0');
  elsif clock'event and clock = '1' then
    if load_sec = '1' then
      temp_sec <= in_value;
      load_sec_out <= '1';
    elsif load_min = '1' then
      temp_min <= in_value;
      load_min_out <= '1';
    elsif load_h = '1' then
      temp_h <= in_value(4 downto 0);
      load_h_out <= '1';
    else
      load_sec_out <= '0';
      load_min_out <= '0';
      load_h_out <= '0';
    end if;
  end if;
end process;

end behavioral;
```

File: display_seven_segments.vhd

Tale componente è analogo a quello descritto nel Paragrafo 6.3.3, con l'unica differenza di prevedere un contatore modulo 4, in quanto si vuole abilitare la visualizzazione su tutte e quattro le cifre del display.

Codice Display 7 Segmenti

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

entity display_seven_segments is
    Port ( clock : in STD_LOGIC;
            reset : in STD_LOGIC;
            value : in STD_LOGIC_VECTOR (15 downto 0);
            anodes : out STD_LOGIC_VECTOR (3 downto 0);
            cathodes : out STD_LOGIC_VECTOR (7 downto 0));
end display_seven_segments;

architecture Structural of display_seven_segments is

signal counter : std_logic_vector(1 downto 0);

COMPONENT counter_mod4
    PORT(
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        counter : out STD_LOGIC_VECTOR (1 downto 0)
    );
END COMPONENT;

COMPONENT cathodes_manager
    PORT(
        counter : IN std_logic_vector(1 downto 0);
        value : IN std_logic_vector(15 downto 0);
        cathodes : OUT std_logic_vector(7 downto 0)
    );
END COMPONENT;

COMPONENT anodes_manager
    PORT(
        counter : IN std_logic_vector(1 downto 0);
        anodes : OUT std_logic_vector(3 downto 0)
    );
END COMPONENT;

begin

counter_instance: counter_mod4 port map(
    clock => clock,
    reset => reset,
    counter => counter
);

cathodes_instance: cathodes_manager port map(
    counter => counter,
    value => value,
    cathodes => cathodes
);

anodes_instance: anodes_manager port map(
    counter => counter,
    anodes => anodes
);
end Structural;
```

File: counter_mod4.vhd

Codice Contatore Modulo 4

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter_mod4 is
    Port ( clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           counter : out STD_LOGIC_VECTOR (1 downto 0));
end counter_mod4;

architecture Behavioral of counter_mod4 is

signal c : std_logic_vector (1 downto 0) := (others => '0');

begin

counter <= c;

counter_process: process(clock, reset)
begin

    if reset = '1' then
        c <= (others => '0');
    elsif rising_edge(clock) then
        c <= std_logic_vector(unsigned(c) + 1);
    end if;

end process;
end Behavioral;

```

File: cathodes_manager.vhd

Codice Cathodes Manager

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity cathodes_manager is
    Port ( counter : in STD_LOGIC_VECTOR (1 downto 0);
           value : in STD_LOGIC_VECTOR (15 downto 0);
           cathodes : out STD_LOGIC_VECTOR (7 downto 0));
end cathodes_manager;

architecture Behavioral of cathodes_manager is

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

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

signal DOT      : STD_LOGIC;
signal cathodes_for_digit : std_logic_vector(6 downto 0) := (others => '0');
signal nibble  : std_logic_vector(3 downto 0) := (others => '0');

```

```

begin
    digit_switching: process(counter, value)
begin
    case counter is
        when "00" =>
            nibble <= digit_0;
            DOT <= '0';
        when "01" =>
            nibble <= digit_1;
            DOT <= '0';
        when "10" =>
            nibble <= digit_2;
            DOT <= '1';
        when "11" =>
            nibble <= digit_3;
            DOT <= '0';
        when others =>
            nibble <= (others => '0');
            DOT <= '0';
    end case;
end process;

seven_segment_decoder_process: process(nibble)
begin
    case nibble is
        when "0000" => cathodes_for_digit <= zero;
        when "0001" => cathodes_for_digit <= one;
        when "0010" => cathodes_for_digit <= two;
        when "0011" => cathodes_for_digit <= three;
        when "0100" => cathodes_for_digit <= four;
        when "0101" => cathodes_for_digit <= five;
        when "0110" => cathodes_for_digit <= six;
        when "0111" => cathodes_for_digit <= seven;
        when "1000" => cathodes_for_digit <= eight;
        when "1001" => cathodes_for_digit <= nine;
        when "1010" => cathodes_for_digit <= a;
        when "1011" => cathodes_for_digit <= b;
        when "1100" => cathodes_for_digit <= c;
        when "1101" => cathodes_for_digit <= d;
        when "1110" => cathodes_for_digit <= e;
        when "1111" => cathodes_for_digit <= f;
        when others => cathodes_for_digit <= (others => '0');
    end case;
end process seven_segment_decoder_process;
cathodes <= not(DOT) & cathodes_for_digit;
end Behavioral;

```

File: anodes_manager.vhd

Codice Anodes Manager

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity anodes_manager is
    Port ( counter : in STD_LOGIC_VECTOR (1 downto 0);
           anodes : out STD_LOGIC_VECTOR (3 downto 0)
         );
end anodes_manager;

architecture Behavioral of anodes_manager is

signal anodes_switching : std_logic_vector(3 downto 0) := (others => '0');

begin
    anodes <= not anodes_switching OR not "1111";

    anodes_process: process(counter)
begin
    case counter is
        when "00" =>
            anodes_switching <= x"1";
        when "01" =>
            anodes_switching <= x"2";
        when "10" =>

```

```

anodes_switching <= x"4";
when "11" =>
    anodes_switching <= x"8";
when others =>
    anodes_switching <= (others => '0');
end case;
end process;

end Behavioral;

```

File: bin2bcd_12bit.vhd

Il codice relativo al componente bin2bcd_12bit è uguale a quello riportato nel Paragrafo 6.3.3.

File: BasysRevEGeneral.ucf

Codice vincoli board Basys

```

# clock pin for Basys rev E Board
NET "clock"      LOC = "P54";

# onboard 7seg display
NET "cathodes<0>"      LOC = "P25";
NET "cathodes<1>"      LOC = "P16";
NET "cathodes<2>"      LOC = "P23";
NET "cathodes<3>"      LOC = "P21";
NET "cathodes<4>"      LOC = "P20";
NET "cathodes<5>"      LOC = "P17";
NET "cathodes<6>"      LOC = "P83";
NET "cathodes<7>"      LOC = "P22";

NET "anodes<0>"      LOC = "P34";
NET "anodes<1>"      LOC = "P33";
NET "anodes<2>"      LOC = "P32";
NET "anodes<3>"      LOC = "P26";

# Leds
NET "led<0>"      LOC = "P15";
NET "led<1>"      LOC = "P14";
NET "led<2>"      LOC = "P8";
NET "led<3>"      LOC = "P7";
NET "led<4>"      LOC = "P5";
NET "led<5>"      LOC = "P4";
#NET "Led<6>"      LOC = "P3";
#NET "Led<7>"      LOC = "P2";

# Switches
NET "in_value<0>"      LOC = "P38";
NET "in_value<1>"      LOC = "P36";
NET "in_value<2>"      LOC = "P29";
NET "in_value<3>"      LOC = "P24";
NET "in_value<4>"      LOC = "P18";
NET "in_value<5>"      LOC = "P12";
#NET "in_value<6>"      LOC = "P10";
#NET "in_value<7>"      LOC = "P6";

# Buttons
NET "reset"      LOC = "P69";
NET "load_sec"     LOC = "P48";
NET "load_min"     LOC = "P47";
NET "load_h"       LOC = "P41";

```

8.6 Simulazione

File: Digital_watch_tb.vhd

Codice del Testbench di Digital Watch

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Digital_watch_tb IS
END Digital_watch_tb;

ARCHITECTURE behavior OF Digital_watch_tb IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT Digital_watch
    PORT(
        clock : IN std_logic;
        reset : IN std_logic;
        load_sec : IN std_logic;
        load_min : IN std_logic;
        load_h : IN std_logic;
        sec_in : IN std_logic_vector(5 downto 0);
        min_in : IN std_logic_vector(5 downto 0);
        h_in : IN std_logic_vector(4 downto 0);
        sec_out : OUT std_logic_vector(5 downto 0);
        min_out : OUT std_logic_vector(5 downto 0);
        h_out : OUT std_logic_vector(4 downto 0)
    );
    END COMPONENT;
    --Inputs
    signal clock : std_logic := '0';
    signal reset : std_logic := '0';
    signal load_sec : std_logic := '0';
    signal load_min : std_logic := '0';
    signal load_h : std_logic := '0';
    signal sec_in : std_logic_vector(5 downto 0) := (others => '0');
    signal min_in : std_logic_vector(5 downto 0) := (others => '0');
    signal h_in : std_logic_vector(4 downto 0) := (others => '0');

    --Outputs
    signal sec_out : std_logic_vector(5 downto 0);
    signal min_out : std_logic_vector(5 downto 0);
    signal h_out : std_logic_vector(4 downto 0);

    -- Clock period definitions
    constant clock_period : time := 10 ns;
BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: Digital_watch PORT MAP (
        clock => clock,
        reset => reset,
        load_sec => load_sec,
        load_min => load_min,
        load_h => load_h,
        sec_in => sec_in,
        min_in => min_in,
        h_in => h_in,
        sec_out => sec_out,
        min_out => min_out,
        h_out => h_out
    );
    -- Clock process definitions
    clock_process :process
    begin
        clock <= '0';
        wait for clock_period/2;
        clock <= '1';
        wait for clock_period/2;
    end process;
    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.

```

```

wait for 100 ns;
-- insert stimulus here
reset <= '1';

wait for 5 ns;
reset <= '0';

wait for clock_period*50;

sec_in <= "111011";
min_in <= "111011";
h_in <= "10111";

load_sec <= '1';
load_min <= '1';
load_h <= '1';

wait for 5 ns;

load_sec <= '0';
load_min <= '0';
load_h <= '0';

wait;
end process;

END;

```

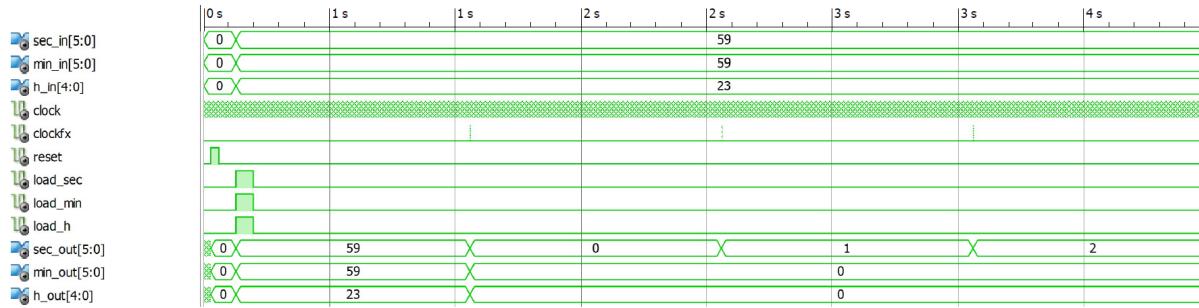


Figura 8.3: Simulazione Digital Watch On Board

Capitolo 9

Esercizio 9

9.1 Traccia

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

- *Carry Lookahead Adder* (CLA), per effettuare la somma di 2 stringhe A e B da 8 bit ciascuna.
- *Carry Save Adder* (CSA), per effettuare la somma di 3 stringhe A, B e C da 8 bit ciascuna.
- *Carry Select Adder*, per effettuare la somma di 2 stringhe A e B da 16 bit ciascuna.
- *Moltiplicatore con somma per righe*, per effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna.
- *Moltiplicatore con somma per diagonali*, per effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna.
- *Moltiplicatore con somma per colonne*, per effettuare il prodotto di 2 stringhe A e B da 6 bit ciascuna.
- *Moltiplicatore a celle MAC*, per effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna.

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

9.2 Introduzione

Tra gli esercizi proposti, si è scelto di realizzare il moltiplicatore a celle MAC (*Multiply-and-Accumulate*). Essendo una macchina aritmetica puramente combinatoria, non è stato necessario decomporre l'architettura del dispositivo in parte operativa e parte di controllo.

Ad ogni modo, per la realizzazione del dispositivo è stato seguito un approccio di tipo strutturale. Dopo aver definito la singola cella MAC, è stato ottenuto il moltiplicatore per composizione.

9.3 Soluzione

Un moltiplicatore a celle MAC è una macchina aritmetica combinatoria in grado di effettuare il prodotto tra numeri naturali, avente l'architettura riportata in Figura 9.2.

La generica cella MAC è un componente puramente combinatorio, ottenuta compiendo un full-adder e una porta AND, come illustrato in Figura 9.1. Ciascuna cella riceve in ingresso:

- I bit x_j e y_i dei due operandi.
- La somma parziale $s_{i,j}$ della cella precedente, situata sulla stessa colonna.
- Il riporto uscente $c_{j,i}$ della cella precedente, situata sulla stessa riga.

Evidentemente, tutte le celle sulla stessa riga ricevono in ingresso lo stesso bit del secondo operando e tutte le celle sulla stessa diagonale ricevono in ingresso lo stesso bit del primo operando. Ciascuna cella calcola quindi:

- Il prodotto parziale tra i bit x_j e y_i dei due operandi, ottenuto mediante una porta AND:

$$x_j \cdot y_i \quad (9.1)$$

- La somma parziale $s_{i+1,j-1}$ tra il prodotto parziale corrente e la somma parziale della cella precedente, situata sulla stessa colonna, tenendo conto del riporto uscente della cella precedente, situata sulla stessa riga.

$$s_{i+1,j-1} = s_{i,j} + x_j \cdot y_i \quad (9.2)$$

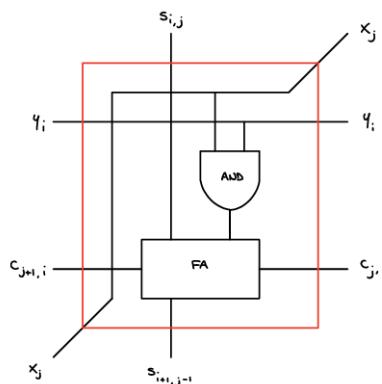


Figura 9.1: Cella MAC

A differenza degli altri moltiplicatori, i quali calcolano i prodotti e le somme parziali in due stadi distinti, il moltiplicatore a celle MAC effettua tali operazioni in un unico stadio. Ciò permette di ottenere dei vantaggi in termini di prestazioni, sebbene l'architettura risenta in ogni caso dei ritardi di propagazione dei riporti.

È stato quindi realizzato un moltiplicatore a celle MAC per numeri naturali codificati su 8 bit, seguendo un'architettura multi-livello. Al livello più alto si trova il *top-level-module*, ovvero il modulo **MAC_multiplier** (**structural**). Tale componente riceve in ingresso due operandi di 8 bit e presenta in uscita il prodotto dei due operandi, espresso su 16 bit. I componenti appartenenti al livello inferiore sono:

- **MAC_cell** (**structural**): rappresenta la singola cella del moltiplicatore MAC. Ciascuna cella MAC è composta a sua volta da un full-adder e una porta and:
 - **full_adder** (**dataflow**)

File: **MAC_multiplier.vhd**

Sfruttando la regolarità dell'architettura è stato possibile descrivere il componente con relativa semplicità.

Il moltiplicatore è stato ottenuto componendo un totale di 64 celle MAC, connesse tra loro come illustrato in Figura 9.2. Sono stati utilizzati due costrutti **for** innestati, rispettivamente per righe e colonne dell'architettura, e, tramite l'uso di due matrici per somme e riporti, sono stati definiti i segnali di interconnessione tra le singole celle.

Codice componente moltiplicatore MAC

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MAC_multiplier is
  Port(   A : in STD_LOGIC_VECTOR (7 downto 0);
          B : in STD_LOGIC_VECTOR (7 downto 0);
          Y : out STD_LOGIC_VECTOR (15 downto 0)
        );
end MAC_multiplier;

architecture Behavioral of MAC_multiplier is

  COMPONENT MAC_cell
    PORT(
      x : IN std_logic;
      s_in : IN std_logic;
      y : IN std_logic;
      c_in : IN std_logic;
      c_out : OUT std_logic;
      s_out : OUT std_logic
    );
  END COMPONENT;

  constant N : positive := 8;
  type matrix is array (N-1 downto 0, N-1 downto 0) of std_logic;
  signal temp_s : matrix;
  signal temp_c : matrix;
  signal temp_Y : STD_LOGIC_VECTOR (15 downto 0);

begin
  MAC_cell_row : for i in 0 to N-1 generate
    l_row: if i=0 generate
      least_row: for j in 0 to N-1 generate
        l_col: if j=0 generate
          least_col: MAC_cell
            PORT MAP (
              x => A(j),
              s_in => '0',
              y => B(i),
              c_in => '0',
              c_out => temp_s(i, j),
              s_out => temp_c(j, i)
            );
        end generate;
      end generate;
    end generate;
  end generate;
  temp_Y <= temp_c;
  Y <= temp_Y;
end Behavioral;

```

```

        c_in => '0',
        c_out => temp_c(i,j),
        s_out => temp_s(i,j)
    );
end generate;

r_col: if j>0 and j<=N-1 generate
    rest_col: MAC_cell
        PORT MAP(
            x => A(j),
            s_in => '0',
            y => B(i),
            c_in => temp_c(i,j-1),
            c_out => temp_c(i,j),
            s_out => temp_s(i,j)
        );
    end generate;
end generate;
end generate;

r_row: if i>0 and i<= N-1 generate
    rest_row: for j in 0 to N-1 generate
        l_col: if j=0 generate
            least_col: MAC_cell
                PORT MAP(
                    x => A(j),
                    s_in => temp_s(i-1,j+1),
                    y => B(i),
                    c_in => '0',
                    c_out => temp_c(i,j),
                    s_out => temp_s(i,j)
                );
        end generate;
        r_col: if j>0 and j<N-1 generate
            rest_col: MAC_cell
                PORT MAP(
                    x => A(j),
                    s_in => temp_s(i-1,j+1),
                    y => B(i),
                    c_in => temp_c(i,j-1),
                    c_out => temp_c(i,j),
                    s_out => temp_s(i,j)
                );
        end generate;
    end generate;
end generate;

m_col: if j=N-1 generate
    most_col: MAC_cell
        PORT MAP(
            x => A(j),
            s_in => temp_c(i-1,j),
            y => B(i),
            c_in => temp_c(i,j-1),
            c_out => temp_c(i,j),
            s_out => temp_s(i,j)
        );
    end generate;
end generate;
end generate;
end generate;

temp_Y(0) <= temp_s(0,0);
temp_Y(1) <= temp_s(1,0);
temp_Y(2) <= temp_s(2,0);
temp_Y(3) <= temp_s(3,0);
temp_Y(4) <= temp_s(4,0);
temp_Y(5) <= temp_s(5,0);
temp_Y(6) <= temp_s(6,0);
temp_Y(7) <= temp_s(7,0);
temp_Y(8) <= temp_s(7,1);
temp_Y(9) <= temp_s(7,2);
temp_Y(10)<= temp_s(7,3);
temp_Y(11)<= temp_s(7,4);
temp_Y(12)<= temp_s(7,5);
temp_Y(13)<= temp_s(7,6);
temp_Y(14)<= temp_s(7,7);
temp_Y(15)<= temp_c(7,7);

y      <= temp_Y;
end Behavioral;
```

File: MAC_cell.vhd

Codice componente cella MAC

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

entity MAC_cell is
    Port ( x : in STD_LOGIC;
           s_in : in STD_LOGIC;
           y : in STD_LOGIC;
           c_in : in STD_LOGIC;
           c_out : out STD_LOGIC;
           s_out : out STD_LOGIC);
end MAC_cell;

architecture Behavioral of MAC_cell is

COMPONENT full_adder
    PORT(
        x : IN std_logic;
        y : IN std_logic;
        c_in : IN std_logic;
        z : OUT std_logic;
        c_out : OUT std_logic
    );
END COMPONENT;

signal temp_y : std_logic;

begin

temp_y <= x and y;

full_adder_0: full_adder PORT MAP (
    x => s_in,
    y => temp_y,
    c_in => c_in,
    z => s_out,
    c_out => c_out
);

end Behavioral;
```

File: full_adder.vhd

Il codice relativo al componente full_adder è uguale a quello riportato nel Paragrafo 4.3

9.4 Schematici

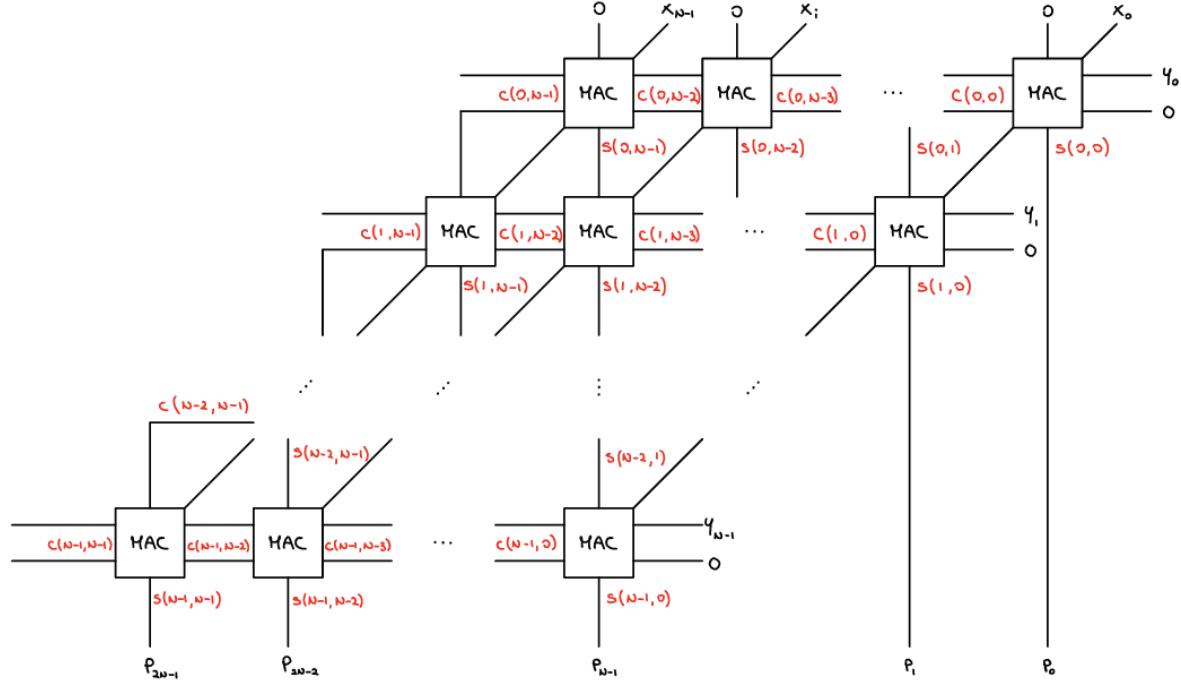


Figura 9.2: Schematico Moltiplicatore MAC

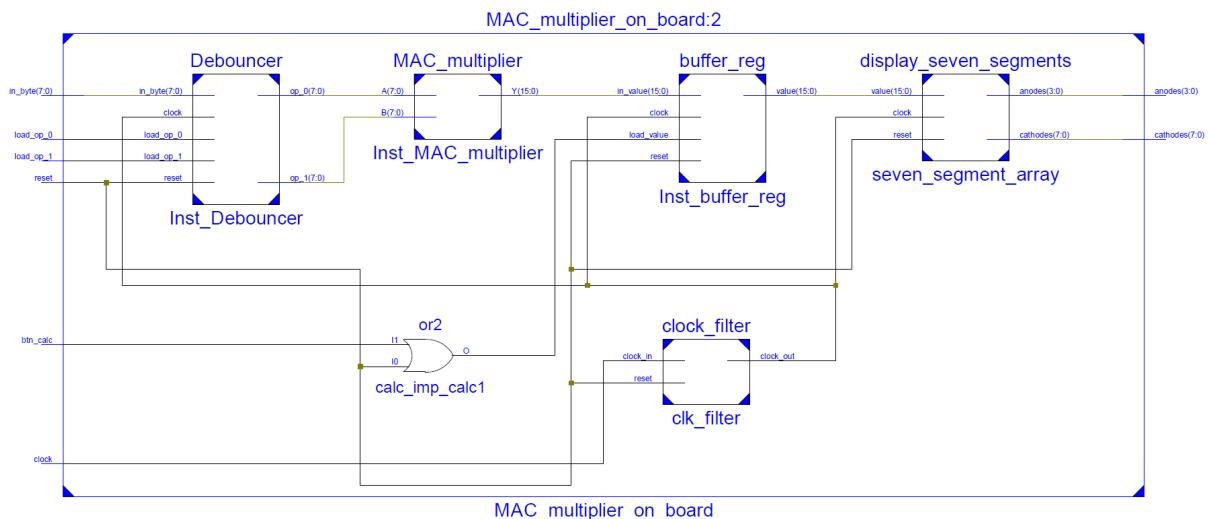


Figura 9.3: Schematico Moltiplicatore MAC On Board

9.5 Sintesi

Come richiesto dalla traccia, è stata effettuata la sintesi del dispositivo sulla board. A tal fine sono stati prodotti ulteriori moduli VHDL.

Il dispositivo sintetizzato sulla board presenta ancora una volta una struttura multi-livello, come riportato in Figura 9.3. Al livello più alto si trova il *top-level-module*, ovvero MAC_multiplier_on_board (**structural**). Tale componente definisce i segnali di ingresso e uscita del dispositivo e descrive le connessioni tra i componenti appartenenti al livello inferiore:

- MAC_multiplier (**structural**): sulla base di quanto esposto in precedenza, rappresenta un moltiplicatore a celle MAC per stringhe di 8 bit.
- clock_filter (**behavioural**): componente responsabile della divisione in frequenza del segnale di clock, del tutto identico a quello descritto nel Paragrafo 5.5.
- Debouncer (**behavioural**): ricevuti in ingresso i segnali di clock, reset e due segnali di abilitazione alla lettura dagli switch, gestisce l'acquisizione degli operandi. Tale componente è analogo a quello descritto nel Paragrafo 5.5.
- display_seven_segments (**structural**): riceve in ingresso un valore numerico rappresentato su 16 bit e restituisce in uscita le configurazioni di anodi e catodi tali da mostrare su quattro cifre del display la codifica in esadecimale del risultato del prodotto. Tale componente è a sua volta costituito da:
 - cathodes_manager (**behavioural**): componente responsabile della gestione dei catodi.
 - anodes_manager (**behavioural**): componente responsabile della gestione degli anodi.
 - counter_mod4 (**behavioural**): contatore modulo 4.
- buffer_reg (**behavioural**): registro di 16 bit inserito al fine di bufferizzare l'uscita del moltiplicatore.

File: MAC_Multiplier_on_board.vhd

Si evidenziano di seguito alcune delle connessioni tra i componenti istanziati nel *top-level-module*:

- MAC_Multiplier: riceve come operandi i valori in uscita dal Debouncer. I segnali di uscita sono invece posti in ingresso al componente buffer_reg.
- Debouncer: riceve come segnale di clock l'uscita del componente clock_filter. I segnali di caricamento degli addendi sono associati a due pulsanti presenti sulla scheda.
- buffer_reg: l'uscita di tale componente viene posta in ingresso al modulo display_seven_segments. In questo modo viene garantita la visualizzazione del risultato corretto.

Codice MAC Multiplier On Board

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity MAC_multiplier_on_board is
  Port(
    clock          : in  STD_LOGIC;
    reset          : in  STD_LOGIC;
    load_op_0      : in  STD_LOGIC;
    load_op_1      : in  STD_LOGIC;
    in_byte        : in  STD_LOGIC_VECTOR(7 downto 0);
    btn_calc       : in  STD_LOGIC;
    anodes         : out STD_LOGIC_VECTOR (3 downto 0);
    cathodes       : out STD_LOGIC_VECTOR (7 downto 0)
  );
end MAC_multiplier_on_board;

architecture structural of MAC_multiplier_on_board is

  COMPONENT display_seven_segments
    PORT(
      clock : in  STD_LOGIC;
      reset : in  STD_LOGIC;
      value : in  STD_LOGIC_VECTOR (15 downto 0);           -- 4 Nibble da mostrare
      anodes : out STD_LOGIC_VECTOR (3 downto 0);
      cathodes : out STD_LOGIC_VECTOR (7 downto 0)
    );
  END COMPONENT;

  COMPONENT Debouncer
    PORT(
      clock : IN std_logic;
      reset : IN std_logic;
      load_op_0 : IN std_logic;
      load_op_1 : IN std_logic;
      in_byte : IN std_logic_vector(7 downto 0);
      op_0 : OUT std_logic_vector(7 downto 0);
      op_1 : OUT std_logic_vector(7 downto 0)
    );
  END COMPONENT;

  COMPONENT clock_filter
    GENERIC(
      clock_frequency_in : integer := 50000000;
      clock_frequency_out : integer := 500
    );
    PORT(
      clock_in      : IN STD_LOGIC;
      reset         : IN STD_LOGIC;
      clock_out     : OUT STD_LOGIC
    );
  END COMPONENT;

  COMPONENT MAC_multiplier
    Port(
      A : in  STD_LOGIC_VECTOR (7 downto 0);
      B : in  STD_LOGIC_VECTOR (7 downto 0);
      Y : out STD_LOGIC_VECTOR (15 downto 0)
    );
  END COMPONENT;

  COMPONENT buffer_reg
    PORT(
      clock : IN std_logic;
      reset : IN std_logic;
      load_value : IN std_logic;
      in_value : IN std_logic_vector(15 downto 0);
      value : OUT std_logic_vector(15 downto 0)
    );
  END COMPONENT;

  signal buf_in : std_logic_vector(15 downto 0);
  signal buf_out : std_logic_vector(15 downto 0);
  signal op_0_temp : std_logic_vector(7 downto 0);
  signal op_1_temp : std_logic_vector(7 downto 0);
  signal calc : std_logic;
  signal clock_fx : std_logic;

begin
  calc <= reset or btn_calc;
  seven_segment_array: display_seven_segments
    PORT MAP(

```

```

        clock => clock_fx,
        reset => reset,
        value => buf_out,
        anodes => anodes,
        cathodes => cathodes
    );
Inst_Debouncer: Debouncer PORT MAP(
    clock => clock_fx,
    reset => reset,
    load_op_0 => load_op_0,
    load_op_1 => load_op_1,
    in_byte => in_byte,
    op_0 => op_0_temp,
    op_1 => op_1_temp
);
clk_filter: clock_filter
GENERIC MAP(
    clock_frequency_in => 50000000,
    clock_frequency_out => 500
)
PORT MAP(
    clock_in => clock,
    reset => reset,
    clock_out => clock_fx
);
Inst_MAC_multiplier: MAC_multiplier
PORT MAP(
    A => op_0_temp,
    B => op_1_temp,
    Y => buf_in
);
Inst_buffer_reg: buffer_reg PORT MAP(
    clock => clock_fx,
    reset => reset,
    load_value => calc,
    in_value => buf_in,
    value => buf_out
);
end structural;

```

File: clock_filter.vhd

Il codice relativo al componente `clock_filter` è uguale a quello riportato nel Paragrafo 5.5.

File: Debouncer.vhd

Codice Debouncer

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Debouncer is
    Port (
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        load_op_0 : in STD_LOGIC;
        load_op_1 : in STD_LOGIC;
        in_byte : in STD_LOGIC_VECTOR(7 downto 0);
        op_0 : out STD_LOGIC_VECTOR(7 downto 0);
        op_1 : out STD_LOGIC_VECTOR(7 downto 0)
    );
begin
end Debouncer;

architecture behavioral of Debouncer is

signal op_0_temp : std_logic_vector(7 downto 0):= (others => '0');
signal op_1_temp : std_logic_vector(7 downto 0):= (others => '0');


```

```

begin
op_0 <= op_0_temp;
op_1 <= op_1_temp;

main: process(clock, reset, load_op_0, load_op_1)
begin
  if reset = '1' then
    op_0_temp <= (others => '0');
    op_1_temp <= (others => '0');
  elsif clock'event and clock = '1' then      -- Attivo sul fronte di salita
    if load_op_0 = '1' then
      op_0_temp <= in_byte;
    elsif load_op_1 = '1' then
      op_1_temp <= in_byte;
    end if;
  end if;
end process;
end behavioral;

```

File: buffer_reg.vhd

Codice Buffer Register

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity buffer_reg is
  Port (
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    load_value : in STD_LOGIC;
    in_value : in STD_LOGIC_VECTOR(15 downto 0);
    value : out STD_LOGIC_VECTOR(15 downto 0)
  );
end buffer_reg;

architecture Behavioral of buffer_reg is
begin
  main: process(clock, reset)
  begin
    if reset = '1' then
      value <= (others => '0');
    elsif rising_edge(clock) then      -- Attivo sul fronte di salita
      if load_value = '1' then
        value <= in_value;
      end if;
    end if;
  end process;
end Behavioral;

```

File: seven_segment_array.vhd

Il codice relativo al componente display_seven_segments è uguale a quello riportato nel Paragrafo 8.5.

File: counter_mod4.vhd

Il codice relativo al componente counter_mod4 è uguale a quello riportato nel Paragrafo 8.5.

File: cathodes_manager.vhd

Il codice relativo al componente `cathodes_manager` è molto simile a quello riportato nel Paragrafo 8.5. L'unica differenza consiste nel non aver abilitato uno dei punti delle cifre.

Codice Cathodes Manager

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity cathodes_manager is
    Port ( counter : in STD_LOGIC_VECTOR (1 downto 0);
           value : in STD_LOGIC_VECTOR (15 downto 0);
           cathodes : out STD_LOGIC_VECTOR (7 downto 0));
end cathodes_manager;

architecture Behavioral of cathodes_manager is

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

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

signal cathodes_for_digit : STD_LOGIC_VECTOR(6 downto 0) := (others => '0');
signal nibble :STD_LOGIC_VECTOR(3 downto 0) := (others => '0');

begin

digit_switching: process(counter, value)
begin
    case counter is
        when "00" =>
            nibble <= digit_0;
        when "01" =>
            nibble <= digit_1;
        when "10" =>
            nibble <= digit_2;
        when "11" =>
            nibble <= digit_3;
        when others =>
            nibble <= (others => '0');
    end case;
end process;

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

```

```

when "1101" => cathodes_for_digit <= d;
when "1110" => cathodes_for_digit <= e;
when "1111" => cathodes_for_digit <= f;
when others => cathodes_for_digit <= (others => '0');
end case;
end process seven_segment_decoder_process;
cathodes <= not "0" & cathodes_for_digit;
end Behavioral;

```

File: anodes_manager.vhd

Il codice relativo al componente anodes_manager è uguale a quello riportato nel Paragrafo 8.5.

File: Basys_250K.ucf

Codice vincoli board Basys

```

# clock pin for Basys rev E Board
NET "clock"      LOC = "P54";

# onboard 7seg display
NET "cathodes<0>"      LOC = "P25";
NET "cathodes<1>"      LOC = "P16";
NET "cathodes<2>"      LOC = "P23";
NET "cathodes<3>"      LOC = "P21";
NET "cathodes<4>"      LOC = "P20";
NET "cathodes<5>"      LOC = "P17";
NET "cathodes<6>"      LOC = "P83";
NET "cathodes<7>"      LOC = "P22";

NET "anodes<0>"      LOC = "P34";
NET "anodes<1>"      LOC = "P33";
NET "anodes<2>"      LOC = "P32";
NET "anodes<3>"      LOC = "P26";

# Switches
NET "in_byte<0>"      LOC = "P38";
NET "in_byte<1>"      LOC = "P36";
NET "in_byte<2>"      LOC = "P29";
NET "in_byte<3>"      LOC = "P24";
NET "in_byte<4>"      LOC = "P18";
NET "in_byte<5>"      LOC = "P12";
NET "in_byte<6>"      LOC = "P10";
NET "in_byte<7>"      LOC = "P6";

# Buttons
NET "reset"      LOC = "P69";
NET "load_op_0"    LOC = "P48";
NET "load_op_1"    LOC = "P47";
NET "btn_calc"    LOC = "P41";

```

9.6 Simulazione

File: MAC_multiplier_tb.vhd

Codice del Testbench di MAC Multiplier

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY MAC_multiplier_tb IS
END MAC_multiplier_tb;

ARCHITECTURE behavior OF MAC_multiplier_tb IS

```

```
-- Component Declaration for the Unit Under Test (UUT)
COMPONENT MAC_multiplier
PORT(
    A : IN  std_logic_vector(7 downto 0);
    B : IN  std_logic_vector(7 downto 0);
    Y : OUT std_logic_vector(15 downto 0)
);
END COMPONENT;

--Inputs
signal A : std_logic_vector(7 downto 0) := (others => '0');
signal B : std_logic_vector(7 downto 0) := (others => '0');

--Outputs
signal Y : std_logic_vector(15 downto 0);

BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: MAC_multiplier PORT MAP (
        A => A,
        B => B,
        Y => Y
    );
    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 30 ns.
        wait for 30 ns;

        -- insert stimulus here
        A <= "000000010";
        B <= "000000010";
        wait for 10 ns;

        assert Y = "00000000000000100"
            report "errore1"
            severity failure;

        A <= "000000001";
        B <= "000000010";
        wait for 10 ns;

        assert Y = "00000000000000010"
            report "errore2"
            severity failure;

        A <= "000000011";
        B <= "000000110";
        wait for 10 ns;

        assert Y = "00000000000010010"
            report "errore3"
            severity failure;

        wait;
    end process;
END;
```



Figura 9.4: Simulazione MAC Multiplier

Capitolo 10

Esercizio 10

10.1 Traccia

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

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

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

10.2 Introduzione

Tra gli esercizi proposti, si è scelto di realizzare il divisore *non-restoring*. Data la complessità dell'esercizio, è stato necessario decomporre l'architettura del dispositivo in parte operativa e parte di controllo. Dopo aver definito gli elementi costituenti l'unità operativa è stata realizzata l'unità di controllo. Dal momento che è possibile considerare l'unità di controllo come un microcontrollore in grado di eseguire l'operazione di divisione, si è scelto di implementarla sia in **logica cablata** che in **logica micropogrammata**.

10.3 Soluzione

È stato realizzato divisore *non-restoring* in grado di effettuare la divisione tra numeri naturali codificati rispettivamente su 4 bit. Tale dispositivo esegue la divisione seguendo l'algoritmo specificato in Figura 10.1.

```

NRDivider:      (in:INBUS; OUT:OUTBUS)
                register S,A[n-1:0],M[n-1:0],Q[n-1:0],COUNT[log2n:0];
                bus INBUS[n-1:0], OUTBUS[n-1:0];
BEGIN:          COUNT:=0;S:=0;
INPUT:           A:=INBUS {carico la prima metà del dividendo D (0 in testa)}
                Q:=INBUS {carico la seconda metà del dividendo D}
                M:=INBUS; {divisore V}

LSHIFT:          S.A.Q[n-1:1]=A.Q; {la prima volta S è 0, e dopo lo shift è ancora 0}

SUB:             if S==0 then
                  S.A:=S.A-M;
else
SUM:             S.A:=S.A+M;
endif

SETq:            Q[0]:=not S;
COUNT:=COUNT+1;

COUNT_TEST:      if COUNT< n then goto LSHIFT;
endif

CORRECTION :    if S==1 then
                  S.A:=S.A+M;
endif

OUTPUT:          OUTBUS:=Q, OUTBUS:=A;
END NRDiver;
```

Figura 10.1: Algoritmo Non Restoring Division

Data la complessità dell'esercizio, è stato necessario suddividere l'architettura del componente in due parti, come illustrato in Figura 10.2: unità operativa e unità di controllo. Le due unità cooperano in modo da realizzare l'algoritmo di divisione non-restoring. In particolare, i segnali di uscita dell'unità di controllo pilotano l'unità operativa.

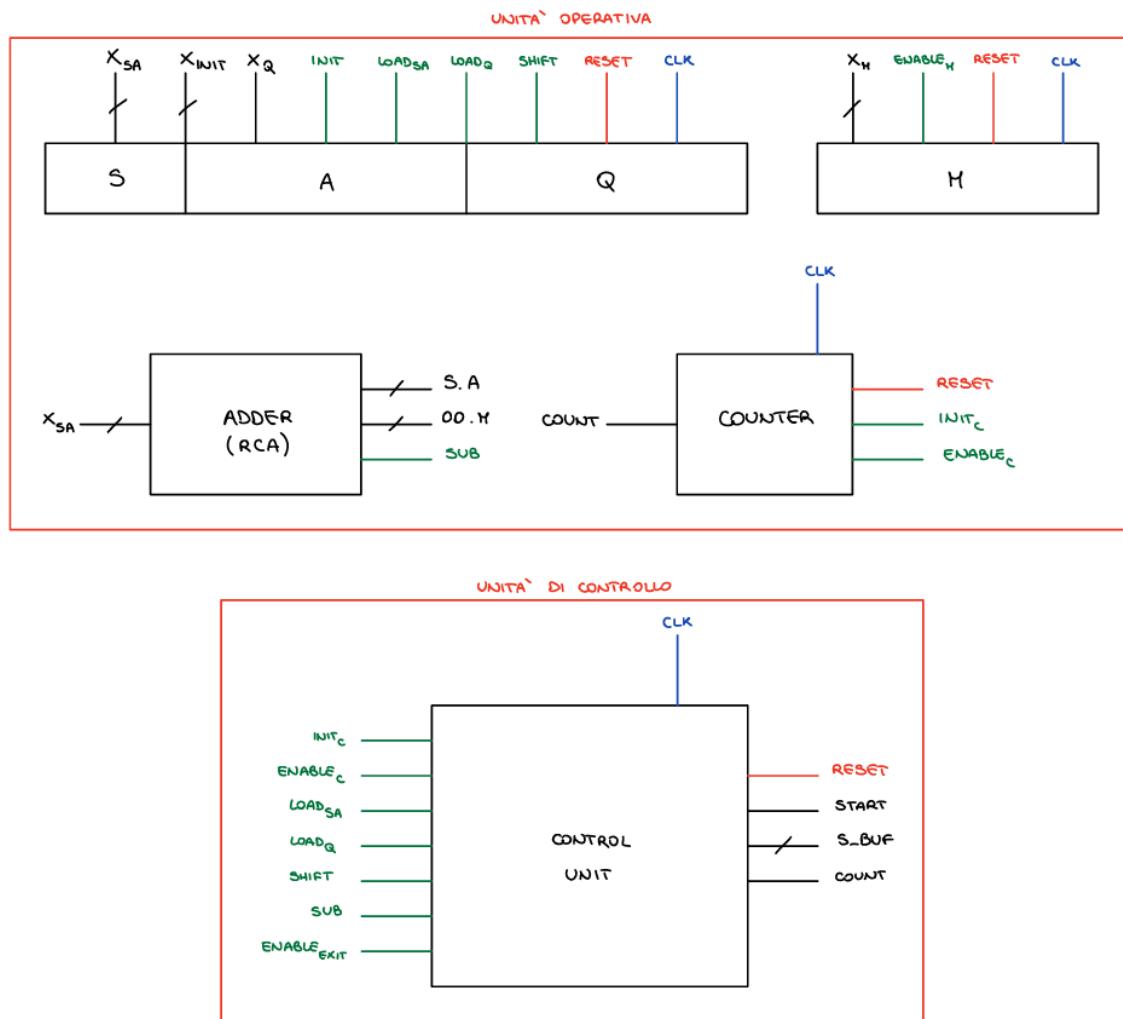


Figura 10.2: Architettura Divisore Non Restoring

Per risolvere problemi di temporizzazione, si è scelto di abilitare l'unità di controllo sul fronte di salita del clock, mentre i componenti dell'unità operativa sul fronte di discesa. In questo modo, non solo l'unità operativa viene abilitata quando i segnali generati dall'unità di controllo sono ormai stabili, ma anche l'unità di controllo valuta i segnali di uscita dell'unità operativa solo dopo che essi si sono stabilizzati.

Di seguito sono descritti e riportati rispettivamente i file relativi ad unità operativa e unità di controllo.

10.3.1 Unità Operativa

L'unità operativa, spesso indicata anche con il termine *datapath*, è costituita dall'insieme di componenti necessari per eseguire le operazioni desiderate.

Per la realizzazione del divisore non-restoring, l'unità operativa prevede i seguenti componenti: registri *SAQ* e *M*, un'unità aritmetica *adder-subtractor*, in grado di effettuare somma e sottrazione ed un contatore modulo 4.

File: RegisterSAQ.vhd

Il componente RegisterSAQ è un registro di 10 bit attivo sul fronte di discesa del segnale di clock. Dal punto di vista logico può essere scomposto in tre sotto-registri:

- *Registro S*: registro di 2 bit contenente informazioni relative al segno del risultato delle operazioni aritmetiche effettuate dall'*adder-subtracter*. Tali informazioni sono utilizzate sia per determinare l'i-esimo bit del quoziente che per valutare la prossima operazione aritmetica da eseguire.
È stato necessario aggiungere un ulteriore bit per la memorizzazione del segno del risultato, poiché esso viene valutato solo in seguito all'operazione di shift.
- *Registro A*: registro di 4 bit contenente inizialmente i 4 bit pari a zero. Al termine dell'algoritmo conterrà il resto della divisione.
- *Registro Q*: registro di 4 bit contenente inizialmente i 4 bit del dividendo. Al termine dell'algoritmo conterrà il quoziente della divisione.

Si è scelto di descrivere il registro attraverso il livello di astrazione **behavioural**, in modo da gestirne la complessità. Difatti, per poter implementare correttamente l'algoritmo di divisone non-restoring, il registro SAQ può operare in diverse modalità. La modalità di funzionamento è stabilita dai seguenti segnali di ingresso:

- INIT: consente l'inizializzazione del registro al valore X_INIT.
- LOAD_SA: consente il caricamento dei bit dei sotto-registri S ed A, mediante il valore di input X_SA.
- LOAD_Q: consente il caricamento del bit meno significato del sotto-registro Q, mediante il valore di input X_Q.
- SHIFT: consente di utilizzare il registro SAQ come un registro a scorrimento effettuando il left-shift di un bit del contenuto.

Codice Registro SAQ

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RegisterSAQ is
  generic( N : integer := 4);
  port(
    CLOCK      : in      STD_LOGIC;
    RESET       : in      STD_LOGIC;
    X_INIT      : in      STD_LOGIC_VECTOR(2*N-2 downto 0);
    X_SA        : in      STD_LOGIC_VECTOR(N downto 0);
    X_Q         : in      STD_LOGIC;
    INIT         : in      STD_LOGIC;
    LOAD_SA     : in      STD_LOGIC;
    LOAD_Q      : in      STD_LOGIC;
    SHIFT        : in      STD_LOGIC;
    Y           : out     STD_LOGIC_VECTOR(2*N+1 downto 0));
end RegisterSAQ;

architecture behavioral of RegisterSAQ is

  signal Y_TEMP : STD_LOGIC_VECTOR(2*N+1 downto 0);

begin
  REG_SAQ : process(CLOCK, RESET)
  begin
    if(RESET = '1') then

```

```

Y_TEMP      <= (others => '0');
elsif(falling_edge(CLOCK)) then
  if(INIT = '1') then
    Y_TEMP <= "000" & X_INIT;
  end if;
  if(LOAD_SA = '1') then
    Y_TEMP(2*N downto N) <= X_SA;
  end if;
  if(LOAD_Q = '1') then
    Y_TEMP(0) <= X_Q;
  end if;
  if(SHIFT = '1') then
    Y_TEMP(2*N+1 downto 1) <= Y_TEMP(2*N downto 0);
  end if;
end process REG_SAQ;

Y <= Y_TEMP;

end behavioral;

```

File: RegisterM.vhd

Il componente RegisterM (**behavioural**) è un registro parallelo-parallelo di 4 bit attivo sul fronte di discesa del segnale di clock, contenente il divisore dell'operazione.

Codice Registro M

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RegisterM is
  generic( N : integer := 4);
  port(
    CLOCK : in STD_LOGIC;
    RESET : in STD_LOGIC;
    ENABLE : in STD_LOGIC;
    X : in STD_LOGIC_VECTOR(N-1 downto 0);
    Y : out STD_LOGIC_VECTOR(N-1 downto 0));
end RegisterM;

architecture behavioral of RegisterM is

begin
  REG_M : process(CLOCK, RESET)
  begin
    if (RESET='1') then
      Y<=(others =>'0');
    elsif (falling_edge(CLOCK)) then
      if(ENABLE='1') then
        Y <= X;
      end if;
    end if;
  end process REG_M;
end behavioral;

```

File: Adder_Subtractor_Nbit.vhd

Il componente Adder_Subtractor_Nbit (**structural**) rappresenta l'unità aritmetica della parte operativa. È in grado di effettuare somma o sottrazione tra due operandi rappresentati in complemento a due, sulla base del segnale SUB, generato dall'unità di controllo. Se tale segnale è basso, viene eseguita la somma, viceversa, viene eseguita la sottrazione.

Per la realizzazione del dispositivo è stata seguita un'architettura multi-livello, riportata in figura 10.3, costituita da un componente RCA_NB1T i cui ingressi sono opportunamente collegati ai segnali di ingresso del componente del livello superiore. Il Ripple

Carry Adder realizzato è del tutto analogo a quello descritto nel Paragrafo 4.3. L'unica differenza sta nell'utilizzo del costrutto generic.

L'operazione di sottrazione viene ottenuta grazie ad una proprietà dell'operatore di XOR bit a bit, in particolare:

$$(X \oplus 1) + 1 = -X \quad (10.1)$$

Il sommatore RCA riceve quindi come operandi il primo operando e l'uscita di una porta XOR, ai cui ingressi sono posti il secondo operando ed il segnale di SUB. Riceve invece come riporto entrante il segnale di SUB stesso.

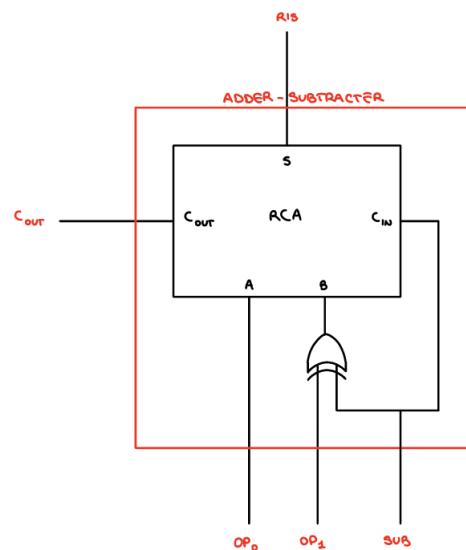


Figura 10.3: Architettura Adder-Subtracter

Codice Adder Substracter ad N bit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ADDER_SUBTRACTOR_Nbit is
    generic( N : POSITIVE := 5);
    port( OP_0      : in STD_LOGIC_VECTOR (N-1 downto 0);
          OP_1      : in STD_LOGIC_VECTOR (N-1 downto 0);
          SUB       : in STD_LOGIC;
          RIS       : out STD_LOGIC_VECTOR (N-1 downto 0);
          C_OUT     : out STD_LOGIC
        );
end ADDER_SUBTRACTOR_Nbit;

architecture structural of ADDER_SUBTRACTOR_Nbit is

component RCA_Nbit
    generic( N : positive);
    port( A      : in STD_LOGIC_VECTOR(N-1 downto 0);
          B      : in STD_LOGIC_VECTOR(N-1 downto 0);
          CIN   : in STD_LOGIC;
          S     : out STD_LOGIC_VECTOR(N-1 downto 0);
          COUT  : out STD_LOGIC);
end component;

signal OP_1_xor : std_logic_vector(N-1 downto 0);

-- Process Combinatorio
begin

```

```

loop : process (OP_1, SUB)
begin
    for i in 0 to N-1 loop
        OP_1_xor(i) <= OP_1(i) xor SUB;
    end loop;
end process;

Inst_RCA_Nbit: RCA_Nbit
generic map( N => N)
port map( A      => OP_0,
          B      => OP_1_xor,
          S      => RIS,
          CIN   => SUB,
          COUT  => C_OUT);

end structural;

```

File: RCA_NBIT.vhd

Codice RCA su N bit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RCA_Nbit is
    generic( n : POSITIVE := 4);
    port( A      : in STD_LOGIC_VECTOR (n-1 downto 0);
          B      : in STD_LOGIC_VECTOR (n-1 downto 0);
          CIN   : in STD_LOGIC;
          S      : out STD_LOGIC_VECTOR (n-1 downto 0);
          COUT  : out STD_LOGIC);
end RCA_Nbit;

architecture structural of RCA_Nbit is

component FULL_ADDER
    port( X      : in STD_LOGIC;
          Y      : in STD_LOGIC;
          C_IN  : in STD_LOGIC;
          Z      : out STD_LOGIC;
          C_OUT : out STD_LOGIC);
end component;

signal C : STD_LOGIC_VECTOR (n-2 downto 0);

begin
    FULL_ADDER_ALL : for i in 0 to n-1 generate
        LEAST: if i = 0 generate
            l: FULL_ADDER
                port map( X      => A(i),
                          Y      => B(i),
                          C_IN  => CIN,
                          Z      => C(i),
                          C_OUT => S(i));
        end generate;
        REST: if (i > 0 and i < n-1) generate
            r: FULL_ADDER
                port map( X      => A(i),
                          Y      => B(i),
                          C_IN  => C(i-1),
                          Z      => C(i),
                          C_OUT => S(i));
        end generate;
        MOST: if i = n-1 generate
            M: FULL_ADDER
                port map( X      => A(i),
                          Y      => B(i),
                          C_IN  => C(i-1),
                          Z      => COUT,
                          C_OUT => S(i));
        end generate;
    end generate;
end structural;

```

File: Full_adder.vhd

Il codice relativo al componente full_adder è uguale a quello riportato nel Paragrafo 4.3.

File: Counter.vhd

Il componente Counter (**behavioural**) rappresenta un contatore modulo 4, il cui scopo è quello di tenere traccia del numero di operazioni aritmetiche effettuate dall'adder-subtractor. Quando il segnale di conteggio raggiunge il valore massimo, viene segnalata la terminazione dell'algoritmo di divisione, alzando un opportuno segnale di uscita COUNT.

Codice contatore modulo 4

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Counter is
    generic( M : integer := 4;
             N : integer := 3);
    port( CLOCK : in STD_LOGIC;
          RESET : in STD_LOGIC;
          ENABLE : in STD_LOGIC;
          INIT   : in STD_LOGIC;
          COUNT  : out STD_LOGIC
        );
end Counter;

architecture Behavioral of Counter is

signal COUNT_TEMP : STD_LOGIC_VECTOR(N-1 downto 0);

begin
    COUNT_PROC : process(CLOCK, RESET)
    begin
        if(RESET = '1') then
            COUNT_TEMP <= (others => '0');
            COUNT <= '0';
        elsif (falling_edge(CLOCK)) then
            if(INIT = '1') then
                COUNT_TEMP <= (others => '0');
            elsif (ENABLE = '1') then
                if (COUNT_TEMP = std_logic_vector(to_unsigned(M-1,
                                                COUNT_TEMP' length))) then
                    COUNT_TEMP <= (others => '0');
                    COUNT <= '1';
                else
                    COUNT_TEMP <= COUNT_TEMP + "1";
                    COUNT <= '0';
                end if;
            end if;
        end process COUNT_PROC;
    end Behavioral;

```

File: Datapath.vhd

I componenti dell'unità operativa sono istanziati ed opportunamente connessi all'interno del modulo Datapath.

Codice Datapath

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Datapath is
  generic(N : integer := 4);
  port(
    CLOCK      : IN std_logic;
    RESET       : IN std_logic;
    SAQ         : IN std_logic_vector(N*2-2 downto 0);
    M           : IN std_logic_vector(N-1 downto 0);
    INIT        : IN std_logic;
    ENABLE_M   : IN std_logic;
    ENABLE_C   : IN std_logic;
    INIT_C     : IN std_logic;
    LOAD_SA    : IN std_logic;
    LOAD_Q     : IN std_logic;
    SHIFT       : IN std_logic;
    SUB         : IN std_logic;
    Q           : OUT std_logic_vector(N-1 downto 0);
    R           : OUT std_logic_vector(N-1 downto 0);
    S           : OUT std_logic_vector(1 downto 0);
    COUNT       : OUT std_logic
  );
end Datapath;

architecture Structural of Datapath is

COMPONENT RegisterM
generic( N:  integer );
PORT(
  CLOCK : IN std_logic;
  RESET : IN std_logic;
  ENABLE : IN std_logic;
  X : IN std_logic_vector(N-1 downto 0);
  Y : OUT std_logic_vector(N-1 downto 0)
);
END COMPONENT;

COMPONENT RegisterSAQ
generic( N : integer );
PORT(
  CLOCK : IN std_logic;
  RESET : IN std_logic;
  X_INIT : IN std_logic_vector(2*N-2 downto 0);
  X_SA : IN std_logic_vector(N downto 0);
  X_Q : IN std_logic;
  INIT : IN std_logic;
  LOAD_SA : IN std_logic;
  LOAD_Q : IN std_logic;
  SHIFT : IN std_logic;
  Y : OUT std_logic_vector(2*N+1 downto 0)
);
END COMPONENT;

COMPONENT ADDER_SUBTRACTOR_Nbit
generic( N : integer );
PORT(
  OP_0 : IN std_logic_vector(N-1 downto 0);
  OP_1 : IN std_logic_vector(N-1 downto 0);
  SUB : IN std_logic;
  RIS : OUT std_logic_vector(N-1 downto 0);
  C_OUT : OUT std_logic
);
END COMPONENT;

COMPONENT Counter
generic( M : integer;
        N : integer );
PORT(
  CLOCK : IN std_logic;
  RESET : IN std_logic;
  INIT   : in STD_LOGIC;
  ENABLE : IN std_logic;
  COUNT  : OUT std_logic
);
END COMPONENT;

signal Y_TEMP : std_logic_vector(N*2+1 downto 0);
signal M_TEMP : std_logic_vector(N-1 downto 0);
signal OP_1_TEMP : std_logic_vector(N downto 0);
signal RIS_TEMP : std_logic_vector(N downto 0);
signal X_Q_TEMP : std_logic;

```

```

begin
    Inst_RegisterM: RegisterM
        generic map( N => N)
        PORT MAP(
            CLOCK => CLOCK,
            RESET => RESET,
            ENABLE => ENABLE_M,
            X => M,
            Y => M_TEMP
        );
        X_Q_TEMP <= not (RIS_TEMP(N));
    end Inst_RegisterM;

    Inst_RegisterSAQ: RegisterSAQ
        generic map( N => N)
        PORT MAP(
            CLOCK => CLOCK,
            RESET => RESET,
            X_INIT => SAQ,
            X_SA => RIS_TEMP,
            X_Q => X_Q_TEMP,
            INIT => INIT,
            LOAD_SA => LOAD_SA,
            LOAD_Q => LOAD_Q,
            SHIFT => SHIFT,
            Y => Y_TEMP(N*2+1 downto 0)
        );
        OP_1_TEMP <= "0" & M_TEMP;
    end Inst_RegisterSAQ;

    Inst_ADDER_SUBTRACTOR_Nbit: ADDER_SUBTRACTOR_Nbit
        generic map( N => N+1)
        PORT MAP(
            OP_0 => Y_TEMP(N*2 downto N),
            OP_1 => OP_1_TEMP,
            SUB => SUB,
            RIS => RIS_TEMP
        );
        Q <= Y_TEMP(N-1 downto 0);
        R <= Y_TEMP(2*N-1 downto N);
        S <= Y_TEMP(2*N+1 downto 2*N);

    end Inst_ADDER_SUBTRACTOR_Nbit;

    Inst_Counter: Counter
        GENERIC MAP(M => 4,
                     N => 3)
        PORT MAP(
            CLOCK => CLOCK,
            RESET => RESET,
            ENABLE => ENABLE_C,
            COUNT => COUNT,
            INIT => INIT_C
        );
        Q <= Y_TEMP(N-1 downto 0);
        R <= Y_TEMP(2*N-1 downto N);
        S <= Y_TEMP(2*N+1 downto 2*N);

    end Inst_Counter;

end Structural;

```

10.3.2 Unità Di Controllo in logica cablata

File: Control_unit.vhd

L'unità di controllo è il componente responsabile della gestione dell'unità operativa.

Per la realizzazione del divisore non-restoring è stata implementata la rete sequenziale il cui automa è riportato in Figura 10.4. Da esso si evince che la macchina in questione è una *macchina di Mealy*.

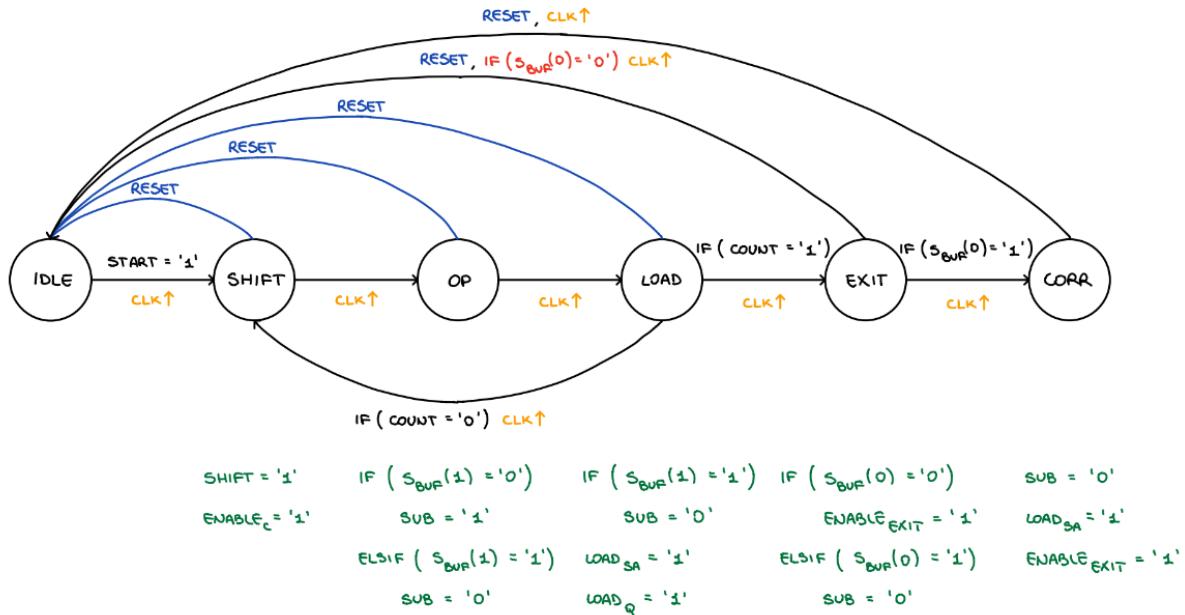


Figura 10.4: Macchina a Stati

L'automa evolve dunque attraverso i seguenti stati:

- IDLE_S: stato di inattività della macchina.
- SHIFT_S: stato in cui viene abilitato il conteggio e lo shift del registro SAQ.
- OP_S: stato in cui viene impostato il valore del segnale SUB, a seconda del bit più significativo del sotto-registro S. Se tale bit è basso viene effettuata la sottrazione, viceversa viene effettuata la somma.
- LOAD_S: stato in cui viene abilitato il caricamento dei sotto-registri S, A e Q. In particolare, in S e A viene caricato il risultato dell'operazione aritmetica eseguita, mentre in Q viene caricato l'opposto del bit di segno del risultato.
In tale stato viene inoltre valutata la condizione di terminazione dell'algoritmo ($COUNT = '1'$). Se la condizione è verificata, si passa allo stato di uscita, altrimenti si torna allo stato di shift.
- EXIT_S: stato di terminazione dell'algoritmo. Viene valutato il bit meno significativo del registro S, contenente il segnale del risultato dell'ultima operazione aritmetica eseguita. Se tale bit è basso viene abilitata la visualizzazione del risultato e si torna allo stato di idle. Viceversa, se tale bit è alto, si effettua un'operazione di correzione, abilitando un'ulteriore addizione e passando ad uno stato aggiuntivo.
- CORR_S: stato di correzione, in cui viene abilitato il caricamento del risultato dell'ultima addizione fra il valore del sotto-registro SA ed il valore del registro M, contenente il divisore. Viene quindi abilitata la visualizzazione del risultato e si torna allo stato di idle.

È opportuno sottolineare che se il segnale di RESET è asserito, l'automa si riporta nello stato IDLE_S qualunque sia lo stato in cui si trova.

È bene osservare inoltre che, non avendo specificato alcuna codifica per gli stati dell'automa, la codifica sarà effettuata automaticamente dallo strumento di sintesi.

Codice Control Unit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ControlUnit is
    port(
        CLOCK          : in STD_LOGIC;
        RESET          : in STD_LOGIC;
        START          : in STD_LOGIC;
        S_BUF          : in STD_LOGIC_VECTOR (1 downto 0);
        COUNT          : in STD_LOGIC;
        LOAD_SA        : out STD_LOGIC;
        LOAD_Q         : out STD_LOGIC;
        SHIFT          : out STD_LOGIC;
        ENABLE_C       : out STD_LOGIC;
        ENABLE_EXIT    : out STD_LOGIC;
        SUB             : out STD_LOGIC;
        INIT_COUNT     : out STD_LOGIC
    );
end ControlUnit;

architecture Behavioral of ControlUnit is

type STATE_TYPE is (
    IDLE_S,
    SHIFT_S,
    OP_S,
    LOAD_S,
    EXIT_S,
    CORR_S);

signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;
begin
    CURR_STATE_PROC : process(CLOCK, RESET)
    begin
        if(RESET = '1') then
            CURRENT_STATE <= IDLE_S;
        elsif(rising_edge(CLOCK)) then
            if(START = '1' and CURRENT_STATE = IDLE_S) then
                CURRENT_STATE <= SHIFT_S;
            else
                CURRENT_STATE <= NEXT_STATE;
            end if;
        end if;
    end process CURR_STATE_PROC;

    NEXT_STATE_PROC : process(CURRENT_STATE, S_BUF, COUNT)
    begin
        LOAD_SA          <= '0';
        LOAD_Q           <= '0';
        SHIFT            <= '0';
        ENABLE_C         <= '0';
        ENABLE_EXIT      <= '0';
        SUB              <= '1';
        INIT_COUNT       <= '0';

        case CURRENT_STATE is
            when IDLE_S =>
                NEXT_STATE <= IDLE_S;
                INIT_COUNT <= '1';

            when SHIFT_S =>
                SHIFT <= '1';
                ENABLE_C <= '1';
                NEXT_STATE <= OP_S;

            when OP_S =>
                if(S_BUF(1) = '0') then
                    SUB <= '1'; -- 1
                elsif(S_BUF(1) = '1') then
                    SUB <= '0'; -- 0
                end if;
                NEXT_STATE <= LOAD_S;
        end case;
    end process;
end Behavioral;

```

```

when LOAD_S =>
  if(S_BUF(1) = '1') then
    SUB <= '0'; -- 0
  end if;

  LOAD_SA <='1';
  LOAD_Q <='1';
  if(COUNT = '0') then
    NEXT_STATE <= SHIFT_S;
  elsif(COUNT = '1') then
    NEXT_STATE <= EXIT_S;
  end if;

when EXIT_S =>
  if(S_BUF(0) = '0') then
    ENABLE_EXIT <= '1';
    NEXT_STATE <= IDLE_S;
  elsif(S_BUF(0) = '1') then
    SUB <= '0'; -- 0
    NEXT_STATE <= CORR_S;
  end if;

when CORR_S =>
  SUB <= '0'; -- 0
  LOAD_SA <='1';
  ENABLE_EXIT <= '1';
  NEXT_STATE <= IDLE_S;

end case;
end process NEXT_STATE_PROC;
end Behavioral;

```

10.3.3 Unità Di Controllo in logica microprogrammata

In alternativa alla logica cablata è possibile adottare un'unità di controllo realizzata in logica microprogrammata. Si fa osservare che l'utilizzo di una logica piuttosto che un'altra comporta la sola riprogettazione dell'unità di controllo, lasciando invece inalterati tutti i componenti dell'unità operativa.

File: Control_Unit.vhd

Nel caso di logica microprogrammata, il componente Control_Unit è essenzialmente costituito da una micro-ROM, all'interno della quale sono contenute le microistruzioni da eseguire, e da una logica combinatoria in grado di gestire opportunamente i salti condizionati. In questo modo viene utilizzato un minor numero di microistruzioni e, di conseguenza, un minor numero di registri di memoria.

Codice Control Unit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Control_Unit is
  PORT(
    CLOCK      : in STD_LOGIC;
    RESET      : in STD_LOGIC;
    START      : in STD_LOGIC;
    S_BUF      : in STD_LOGIC_VECTOR (1 downto 0);
    COUNT      : in STD_LOGIC;
    en_sh      : out std_logic;
    en_ex      : out std_logic;
    en_cnt     : out std_logic;
    load_q     : out std_logic;
    load_sa    : out std_logic;
    sub        : out std_logic
  );
end Control_Unit;

architecture Behavioral of Control_Unit is
  COMPONENT Control_Store

```

```

PORT(
    address : IN std_logic_vector(2 downto 0);
    word : OUT std_logic_vector(10 downto 0)
);
END COMPONENT;

signal word : std_logic_vector(10 downto 0);
signal mpc : std_logic_vector(2 downto 0);
signal jcnt : std_logic;
signal jop : std_logic;
signal mir_reg : std_logic_vector(10 downto 0);

begin
    Inst_Control_Store: Control_Store PORT MAP (
        address => mpc,
        word => word
    );
    -- Jump Signals
    jcnd      <= mir_reg(6);
    jop       <= mir_reg(7);
    -- Output to datapath
    en_sh     <= mir_reg(0);
    en_ex     <= mir_reg(1);
    en_cnt    <= mir_reg(2);
    load_q    <= mir_reg(3);
    load_sa   <= mir_reg(4);
    sub       <= mir_reg(5);
    -- MIR Register
    mir_proc : process(clock) is
    begin
        if rising_edge(clock) then
            if reset = '1' then
                mir_reg <= "000000000000";
            else
                mir_reg <= word;
            end if;
        end if;
    end process mir_proc;
    -- MPC Register
    mpc <=
        "000" when reset = '1' else
        "001" when start = '1' else
        "010" when jop = '1' and S_BUF(1) = '0' else
        "011" when jop = '1' and S_BUF(1) = '1' else
        "101" when jcnd = '1' and COUNT = '1' and S_BUF(0) = '0' else
        "100" when jcnd = '1' and COUNT = '1' and S_BUF(0) = '1' else
        "001" when jcnd = '1' and COUNT = '0' else
        mir_reg(10 downto 8);
end Behavioral;

```

File: Control_Store.vhd

Il componente Control_Store rappresenta una micro-ROM di sei locazioni di memoria, ciascuna composta da undici bit. Tramite i bit di ciascuna microistruzione è possibile pilotare i segnali in ingresso al datapath al fine di realizzare l'algoritmo di divisione.

Ciascuna microistruzione è composta dai seguenti campi:

- **NEXT_ADDR (bit da 10 a 7)**: rappresentano l'indirizzo della prossima istruzione da eseguire. Essendo la memoria composta da sei registri, sono infatti sufficienti tre bit per indirizzarla correttamente.
- **JOP e JCNT (bit da 6 a 5)**: rappresentano i bit necessari per segnalare le condizioni di salto condizionato.
- **SUB (bit 4)**: rappresenta il segnale di sottrazione verso il componente *adder-subtractor*.

- **LOAD_SA** e **LOAD_Q** (bit da 4 a 3): rappresentano i segnali di abilitazione al caricamento dei registri SA e Q.
- **EN_CNT**, **EN_EX** e **EN_SH** (bit da 2 a 0): rappresentano rispettivamente il segnale di abilitazione al conteggio verso il contatore, il segnale di abilitazione alla visualizzazione verso il display e il segnale di abilitazione allo shift del registro SAQ.

Codice Control Store

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Control_Store is
  port (
    address : in std_logic_vector(2 downto 0);
    word    : out std_logic_vector(10 downto 0)
  );
end entity Control_Store;

architecture dataflow of Control_Store is
  -- Types
  --! Control store content
  type ctrl_str_type is array (5 downto 0) of std_logic_vector(10 downto 0);

  -- Constants
  constant words : ctrl_str_type := (
    --BEGIN_WORDS_ENTRY
    0 => "0000000000",
    1 => "01010000101",
    2 => "00101111000",
    3 => "00101011000",
    4 => "10100010000",
    5 => "00000000010",
    others => (others => '0')
    --END_WORDS_ENTRY
  );
begin
  -- architecture dataflow
  word <= words(to_integer(unsigned(address)));
end architecture dataflow;
```

10.4 Schematici

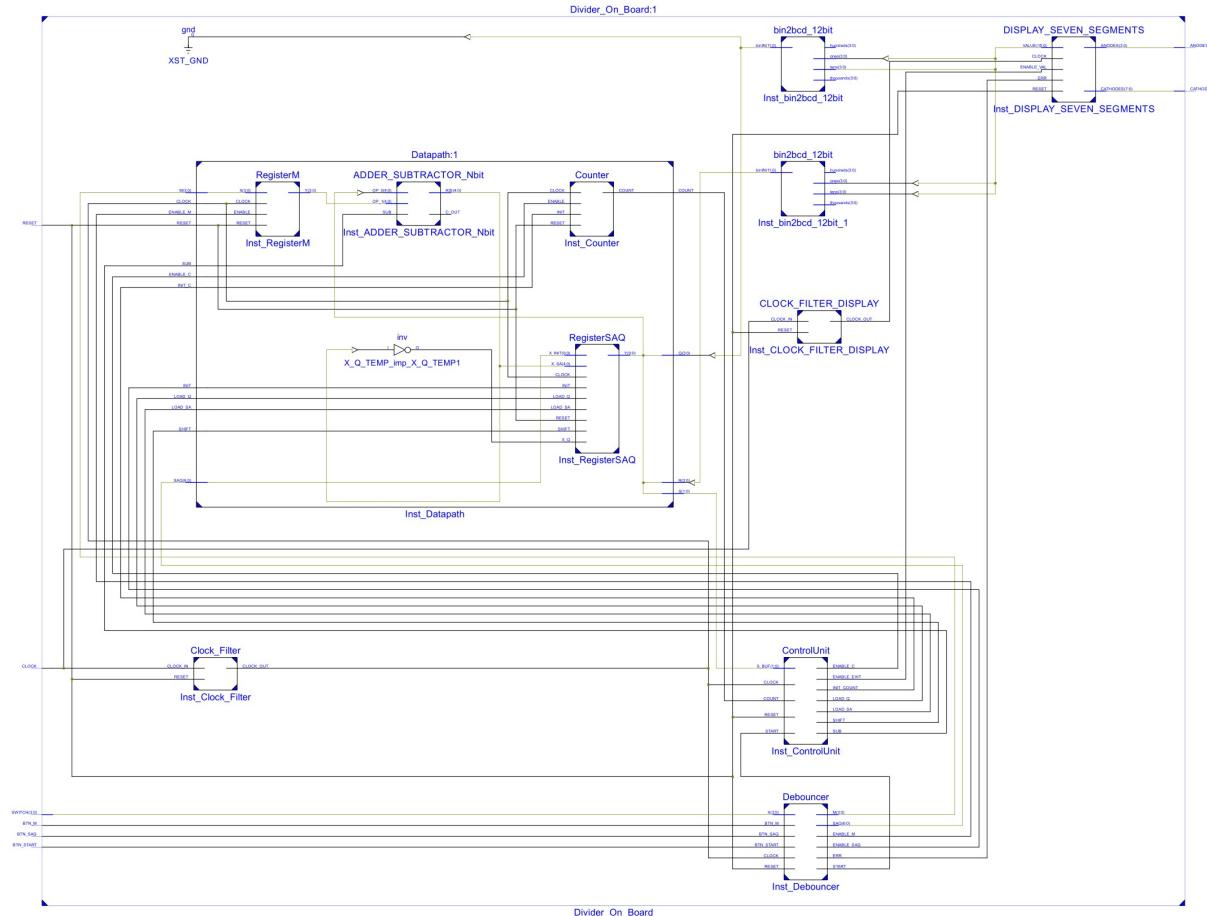


Figura 10.5: Schematico Divisore On Board

10.5 Sintesi

Come richiesto dalla traccia, è stata effettuata la sintesi del dispositivo sulla board. A tal fine sono stati prodotti ulteriori moduli VHDL.

Il dispositivo sintetizzato sulla board presenta ancora una volta una struttura multi-livello, come riportato in Figura 10.5. Al livello più alto si trova il *top-level-module*, ovvero Divider_On_Board (**structural**). Tale componente definisce i segnali di ingresso e uscita del dispositivo e descrive le connessioni tra i componenti appartenenti al livello inferiore:

- **Datapath (structural)**: sulla base di quanto esposto in precedenza, rappresenta l'unità operativa del divisore non-restoring.
 - **ControlUnit (behavioural)**: sulla base di quanto esposto in precedenza, rappresenta l'unità di controllo del divisore non-restoring.

- **clock_filter (behavioural)**: componente responsabile della divisione in frequenza del segnale di clock, del tutto analogo a quello descritto nel Paragrafo 5.5. Tuttavia, a differenza del precedente, tale componente è stato realizzato in modo da generare in uscita un segnale avente un *duty cycle* del 50%.
- **Debouncer (behavioural)**: ricevuti in ingresso i segnali di clock, reset e due segnali di abilitazione alla lettura dagli switch, gestisce l'acquisizione degli operandi. Tale componente è analogo a quello descritto nel Paragrafo 5.5. Tuttavia, a differenza del precedente riceve un ulteriore segnale di ingresso che sta ad indicare l'inizio dell'algoritmo di divisione e fornisce in uscita un segnale di errore in caso si tenti di effettuare una divisione per zero.
- **clock_filter_Display (behavioural)**: componente responsabile della divisione in frequenza del segnale di clock, del tutto identico a quello descritto nel Paragrafo 5.5.
- **display_seven_segments (structural)**: riceve in ingresso un valore numerico rappresentato su 16 bit e restituisce in uscita le configurazioni di anodi e catodi tali da mostrare su quattro cifre del display la codifica in BCD di quoziente e resto della divisione. Tale componente è a sua volta costituito da:
 - **cathodes_manager (behavioural)**: componente responsabile della gestione dei catodi.
 - **anodes_manager (behavioural)**: componente responsabile della gestione degli anodi.
 - **counter_mod4 (behavioural)**: contatore modulo 4.

All'interno del modulo è presente inoltre un process responsabile dell'aggiornamento dei valori da visualizzare e della gestione di un'eventuale condizione di errore, verificata in seguito al tentativo di divisione per zero.

- **bin2bcd_12bit (behavioural)**: convertitore da binario a BCD, del tutto identico a quello descritto nel Paragrafo 6.3.3.

All'interno del top-level-module sono istanziati due convertitori, per convertire rispettivamente quoziente e resto della divisione.

File: Divider_On_Board.vhd

Codice divisiore On Board

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

entity Divider_On_Board is
  GENERIC( N : integer := 4);
  port(
    CLOCK : in STD_LOGIC;
    RESET : in STD_LOGIC;
    SWITCH : in std_logic_vector(3 downto 0);
    BTN_SAQ : in STD_LOGIC;
    BTN_M : in STD_LOGIC;
    BTN_START : in STD_LOGIC;
    ANODES : out std_logic_vector(3 downto 0);
    CATHODES : out std_logic_vector(7 downto 0));
end Divider_On_Board;
```

```

architecture Structural of Divider_On_Board is
COMPONENT Datapath
  GENERIC( N : integer);
  PORT(
    CLOCK      : IN std_logic;
    RESET      : IN std_logic;
    SAQ        : IN std_logic_vector(N*2-2 downto 0);
    M          : IN std_logic_vector(N-1 downto 0);
    INIT       : IN std_logic;
    ENABLE_M   : IN std_logic;
    ENABLE_C   : IN std_logic;
    INIT_C     : IN std_logic;
    LOAD_SA    : IN std_logic;
    LOAD_Q     : IN std_logic;
    SHIFT      : IN std_logic;
    SUB        : IN std_logic;
    Q          : OUT std_logic_vector(N-1 downto 0);
    R          : OUT std_logic_vector(N-1 downto 0);
    S          : OUT std_logic_vector(1 downto 0);
    COUNT      : OUT std_logic
  );
END COMPONENT;

COMPONENT ControlUnit
  PORT(
    CLOCK : IN std_logic;
    RESET : IN std_logic;
    START : IN std_logic;
    S_BUF : IN std_logic_vector(1 downto 0);
    COUNT : IN std_logic;
    LOAD_SA : OUT std_logic;
    LOAD_Q : OUT std_logic;
    SHIFT : OUT std_logic;
    ENABLE_C : OUT std_logic;
    ENABLE_EXIT : OUT std_logic;
    SUB : OUT std_logic;
    INIT_COUNT : out STD_LOGIC
  );
END COMPONENT;

COMPONENT Clock_Filter
  generic( CLOCK_FREQUENCY_IN : integer;
           CLOCK_FREQUENCY_OUT : integer);
  PORT(
    CLOCK_IN : IN std_logic;
    RESET : IN std_logic;
    CLOCK_OUT : OUT std_logic
  );
END COMPONENT;

COMPONENT CLOCK_FILTER_DISPLAY
  generic( CLOCK_FREQUENCY_IN : integer;
           CLOCK_FREQUENCY_OUT : integer);
  PORT(
    CLOCK_IN : IN std_logic;
    RESET : IN std_logic;
    CLOCK_OUT : OUT std_logic
  );
END COMPONENT;

COMPONENT DISPLAY_SEVEN_SEGMENTS
  PORT(
    CLOCK : IN std_logic;
    RESET : IN std_logic;
    ERR   : in STD_LOGIC;
    ENABLE_VAL : IN std_logic;
    VALUE : IN std_logic_vector(15 downto 0);
    ANODES : OUT std_logic_vector(3 downto 0);
    CATHODES : OUT std_logic_vector(7 downto 0)
  );
END COMPONENT;

COMPONENT bin2bcd_12bit
  PORT(
    binIN : IN std_logic_vector(11 downto 0);
    ones : OUT std_logic_vector(3 downto 0);
    tens : OUT std_logic_vector(3 downto 0);
    hundreds : OUT std_logic_vector(3 downto 0);
    thousands : OUT std_logic_vector(3 downto 0)
  );
END COMPONENT;

COMPONENT Debouncer
  generic (N : integer);
  PORT(

```

```

CLOCK : IN std_logic;
RESET : IN std_logic;
X : IN std_logic_vector(N-1 downto 0);
BTN_SAQ : IN std_logic;
BTN_M : IN std_logic;
BTN_START : IN std_logic;
SAQ : OUT std_logic_vector(2*N-2 downto 0);
M : OUT std_logic_vector(N-1 downto 0);
START : OUT std_logic;
ENABLE_SAQ : OUT std_logic;
ENABLE_M : OUT std_logic;
ERR : out STD_LOGIC
);
END COMPONENT;

signal CLOCK_FX : std_logic;
signal CLOCK_FX_1 : std_logic;
signal SAQ_TEMP : std_logic_vector(N*2-2 downto 0);
signal M_TEMP : std_logic_vector(N-1 downto 0);
signal INIT_TEMP : std_logic;
signal ENABLE_M_TEMP : std_logic;
signal LOAD_SA_TEMP : std_logic;
signal LOAD_Q_TEMP : std_logic;
signal SHIFT_TEMP : std_logic;
signal SUB_TEMP : std_logic;
signal START_TEMP : std_logic;
signal ENABLE_C_TEMP : std_logic;
signal COUNT_TEMP : std_logic;
signal S_BUF_TEMP : std_logic_vector(1 downto 0);
signal Q_TEMP : std_logic_vector(N-1 downto 0);
signal R_TEMP : std_logic_vector(N-1 downto 0);
signal Q_CONV : std_logic_vector(11 downto 0);
signal R_CONV : std_logic_vector(11 downto 0);
signal VALUE : std_logic_vector(15 downto 0);
signal ENABLE_EXIT : std_logic;
signal INIT_COUNT_TEMP : STD_LOGIC;
signal ERR_TEMP : STD_LOGIC;

begin
Inst_Datapath: Datapath
GENERIC MAP (N => 4)
PORT MAP (
    CLOCK => CLOCK_FX,
    RESET => RESET,
    SAQ => SAQ_TEMP,
    M => M_TEMP,
    INIT => INIT_TEMP,
    ENABLE_M => ENABLE_M_TEMP,
    LOAD_SA => LOAD_SA_TEMP,
    LOAD_Q => LOAD_Q_TEMP,
    SHIFT => SHIFT_TEMP,
    SUB => SUB_TEMP,
    ENABLE_C => ENABLE_C_TEMP,
    INIT_C => INIT_COUNT_TEMP,
    Q => Q_TEMP,
    R => R_TEMP,
    S => S_BUF_TEMP,
    COUNT => COUNT_TEMP
);
Inst_ControlUnit: ControlUnit
PORT MAP (
    CLOCK => CLOCK_FX,
    RESET => RESET,
    START => START_TEMP,
    S_BUF => S_BUF_TEMP,
    COUNT => COUNT_TEMP,
    LOAD_SA => LOAD_SA_TEMP,
    LOAD_Q => LOAD_Q_TEMP,
    SHIFT => SHIFT_TEMP,
    ENABLE_C => ENABLE_C_TEMP,
    ENABLE_EXIT => ENABLE_EXIT,
    SUB => SUB_TEMP,
    INIT_COUNT => INIT_COUNT_TEMP
);
Inst_Clock_Filter: Clock_Filter
GENERIC MAP (
    CLOCK_FREQUENCY_IN => 50000000,
    CLOCK_FREQUENCY_OUT => 500
)
PORT MAP (
    CLOCK_IN => CLOCK,
    RESET => RESET,
    CLOCK_OUT => CLOCK_FX
)

```

```

);
Inst_Debouncer: Debouncer
GENERIC MAP(N => 4)
PORT MAP(
    CLOCK => CLOCK_FX,
    RESET => RESET,
    X => SWITCH,
    BTN_SAQ => BTN_SAQ,
    BTN_M => BTN_M,
    BTN_START => BTN_START,
    SAQ => SAQ_TEMP,
    M => M_TEMP,
    START => START_TEMP,
    ENABLE_SAQ => INIT_TEMP,
    ENABLE_M => ENABLE_M_TEMP,
    ERR => ERR_TEMP
);
Inst_CLOCK_FILTER_DISPLAY: CLOCK_FILTER_DISPLAY
GENERIC MAP(
    CLOCK_FREQUENCY_IN => 50000000,
    CLOCK_FREQUENCY_OUT => 500
)
PORT MAP(
    CLOCK_IN => CLOCK,
    RESET => RESET,
    CLOCK_OUT => CLOCK_FX_1
);
Inst_DISPLAY_SEVEN_SEGMENTS: DISPLAY_SEVEN_SEGMENTS PORT MAP(
    CLOCK => CLOCK_FX_1,
    RESET => RESET,
    ERR => ERR_TEMP,
    ENABLE_VAL => ENABLE_EXIT,
    VALUE => VALUE,
    ANODES => ANODES,
    CATHODES => CATHODES
);
Q_CONV <= "0000000" & Q_TEMP;
Inst_bin2bcd_12bit: bin2bcd_12bit PORT MAP(
    binIN => Q_CONV,
    ones => value(11 downto 8),
    tens => value(15 downto 12)
);
R_CONV <= "0000000" & R_TEMP;
Inst_bin2bcd_12bit_1: bin2bcd_12bit PORT MAP(
    binIN => R_CONV,
    ones => value(3 downto 0),
    tens => value(7 downto 4)
);
end Structural;

```

File: Clock_filter.vhd

Il componente `Clock_filter`, a differenza di quelli precedentemente realizzati e dello stesso `Clock_filter_Display`, è stato realizzato in modo da generare in uscita un segnale avente un *duty cycle* del 50%. In questo modo è possibile far lavorare unità operativa e unità di controllo su fronti opposti, garantendo che i segnali di controllo in ingresso al datapath e le uscite dei registri in ingresso alla control unit abbiano tempo sufficiente per stabilizzarsi.

Codice Clock Filter

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Clock_Filter is
    generic(
        CLOCK_FREQUENCY_IN      : integer := 50000000;
        CLOCK_FREQUENCY_OUT     : integer := 500);
    port(
        CLOCK_IN : in STD_LOGIC;
        RESET   : in STD_LOGIC;

```

```

        CLOCK_OUT : out  STD_LOGIC);
end Clock_Filter;

architecture Behavioral of Clock_Filter is
begin
    signal CLOCK_FX : std_logic := '0';
    constant COUNT_MAX_VALUE: integer := CLOCK_FREQUENCY_IN/(CLOCK_FREQUENCY_OUT)-1;
    begin
        CLOCK_OUT <= CLOCK_FX;

        count_for_division: process(CLOCK_IN, RESET)
        variable COUNTER : integer range 0 to COUNT_MAX_VALUE := 0;
        begin
            if RESET = '1' then
                COUNTER := 0;
                CLOCK_FX <= '0';
            elsif CLOCK_IN' event and CLOCK_IN = '1' then
                if COUNTER = COUNT_MAX_VALUE then
                    if(CLOCK_FX = '0') then
                        CLOCK_FX <= '1';
                    elsif(CLOCK_FX = '1')then
                        CLOCK_FX <= '0';
                    end if;
                    COUNTER := 0;
                else
                    COUNTER := COUNTER + 1;
                end if;
            end if;
        end process;
    end Behavioral;

```

File: Debouncer.vhd

Codice Debouncer

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Debouncer is
    generic (N : integer := 4);
    port (CLOCK : in STD_LOGIC;
          RESET : in STD_LOGIC;
          X : in STD_LOGIC_VECTOR(N-1 downto 0); --switch
          BTN_SAQ : in STD_LOGIC;
          BTN_M : in STD_LOGIC;
          BTN_START : in STD_LOGIC;
          SAQ : out STD_LOGIC_VECTOR(2*N-2 downto 0); -- dividendo
          M : out STD_LOGIC_VECTOR(N-1 downto 0); -- divisore
          START : out STD_LOGIC;
          ENABLE_SAQ : out STD_LOGIC;
          ENABLE_M : out STD_LOGIC;
          ERR : out STD_LOGIC
        );
end Debouncer;

architecture Behavioral of Debouncer is

begin
    DEB_PROC : process (CLOCK, RESET )
    variable i : integer := 0;
    begin
        if(RESET = '1') then
            SAQ <= (others =>'0');
            M <=(others =>'0');
            ENABLE_SAQ <= '1';
            ENABLE_M <= '1';
            ERR_TEMP <= '0';
            i := 0;
        end if;
        if(CLOCK'event and CLOCK = '1') then
            if(i = N) then
                if(SAQ <= (others =>'0')) then
                    M <= (others =>'0');
                    ENABLE_SAQ <= '0';
                    ENABLE_M <= '0';
                    ERR_TEMP <= '1';
                end if;
            else
                i := i + 1;
            end if;
        end if;
    end process;
end Behavioral;

```

```

        elsif(rising_edge(CLOCK)) then
            if(BTN_SAQ = '1' and i = 0) then
                SAQ <= "000" & X;
                ENABLE_SAQ <= '1';
                i := 1;
            elsif(BTN_M = '1' and i = 1) then
                M <= X;
                M_TEMP <= X;
                ENABLE_M <= '1';
                i := 2;
            elsif(BTN_START = '1' and i = 2) then
                if (M_TEMP = std_logic_vector(to_unsigned(CONST, M_TEMP' length))) then
                    ERR_TEMP <= '1';
                else
                    START<= '1';
                    ERR_TEMP <= '0';
                end if;
                ENABLE_SAQ <= '0';
                ENABLE_M <= '0';
                i := 0;
            else
                ENABLE_SAQ <= '0';
                ENABLE_M <= '0';
                START <= '0';
            end if;
        end if;
    end process DEB_PROC;

    ERR <= ERR_TEMP;
end Behavioral;

```

File: Clock_filter_display.vhd

Il codice relativo al componente Clock_filter_display è uguale a quello riportato nel Paragrafo 5.5.

File: seven_segment_array.vhd

Codice Display 7 Segmenti

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DISPLAY_SEVEN_SEGMENTS is
    port(    CLOCK      : in STD_LOGIC;
              RESET      : in STD_LOGIC;
              ENABLE_VAL : in STD_LOGIC;
              ERR        : in STD_LOGIC;
              VALUE      : in STD_LOGIC_VECTOR (15 downto 0);
              ANODES    : out STD_LOGIC_VECTOR (3 downto 0);
              CATHODES  : out STD_LOGIC_VECTOR (7 downto 0));
end DISPLAY_SEVEN_SEGMENTS;

architecture structural of DISPLAY_SEVEN_SEGMENTS is

    signal COUNTER : STD_LOGIC_VECTOR (1 downto 0);
    signal TEMP_VAL : std_logic_vector(15 downto 0);
    signal ERR_OUT : std_logic;

    component COUNTER_MOD4
        port(    CLOCK      : in STD_LOGIC;
                  RESET      : in STD_LOGIC;
                  COUNTER    : out STD_LOGIC_VECTOR (1 downto 0));
    end component;

    component CATHODES_MANAGER
        port(    COUNTER : IN STD_LOGIC_VECTOR (1 downto 0);
                  ERR      : in STD_LOGIC;
                  VALUE   : IN std_logic_vector(15 downto 0);
                  CATHODES : OUT std_logic_vector(7 downto 0));
    end component;

    component ANODES_MANAGER
        port(    COUNTER : IN STD_LOGIC_VECTOR (1 downto 0);
                  ANODES : OUT std_logic_vector(3 downto 0));
    end component;

```

```

end component;

begin
  COUNTER_INSTANCE: COUNTER_MOD4 port map(
    CLOCK      => CLOCK,
    RESET      => RESET,
    COUNTER    => COUNTER
  );
  CATHODES_INSTANCE: CATHODES_MANAGER port map(
    COUNTER    => COUNTER,
    ERR        => ERR_OUT,
    VALUE      => TEMP_VAL,
    CATHODES   => CATHODES
  );
  ANODES_INSTANCE: ANODES_MANAGER port map(
    COUNTER    => COUNTER,
    ANODES     => ANODES
  );
  VAL_PROC : process(CLOCK, RESET)
  begin
    if(RESET = '1') then
      TEMP_VAL <= (others => '0');
      ERR_OUT <= '0';
    elsif(falling_edge(CLOCK)) then
      if(ERR = '1') then
        ERR_OUT <= '1';
      elsif(ENABLE_VAL = '1') then
        ERR_OUT <= '0';
        TEMP_VAL <= VALUE;
      end if;
    end if;
  end process VAL_PROC;
end structural;

```

File: Counter_mod4.vhd

Il codice relativo al componente Counter_mod4 è uguale a quello riportato nel Paragrafo 8.5.

File: cathodes_manager.vhd

Il componente cathodes_manager è analogo a quello riportato nel Paragrafo 8.5. Tuttavia, presenta alcune differenze dovute alla visualizzazione di un messaggio di errore in caso di tentativo di divisione per zero.

Codice Manager Catodi

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CATHODES_MANAGER is
  port ( COUNTER : in STD_LOGIC_VECTOR (1 downto 0);
         ERR      : in STD_LOGIC;
         VALUE    : in STD_LOGIC_VECTOR (15 downto 0);
         CATHODES : out STD_LOGIC_VECTOR (7 downto 0));
end CATHODES_MANAGER;

architecture behavioral of CATHODES_MANAGER is

  constant zero  : STD_LOGIC_VECTOR(6 downto 0) := "1000000";
  constant one   : STD_LOGIC_VECTOR(6 downto 0) := "1111001";
  constant two   : STD_LOGIC_VECTOR(6 downto 0) := "0100100";
  constant three  : STD_LOGIC_VECTOR(6 downto 0) := "0110000";
  constant four   : STD_LOGIC_VECTOR(6 downto 0) := "0011001";
  constant five   : STD_LOGIC_VECTOR(6 downto 0) := "0010010";
  constant six   : STD_LOGIC_VECTOR(6 downto 0) := "0000010";
  constant seven  : STD_LOGIC_VECTOR(6 downto 0) := "1111000";
  constant eight  : STD_LOGIC_VECTOR(6 downto 0) := "0000000";
  constant nine   : STD_LOGIC_VECTOR(6 downto 0) := "0010000";

```

```

constant a      : STD_LOGIC_VECTOR(6 downto 0) := "0001000";
constant b      : STD_LOGIC_VECTOR(6 downto 0) := "0000011";
constant c      : STD_LOGIC_VECTOR(6 downto 0) := "1000110";
constant d      : STD_LOGIC_VECTOR(6 downto 0) := "0100001";
constant e      : STD_LOGIC_VECTOR(6 downto 0) := "0000110";
constant f      : STD_LOGIC_VECTOR(6 downto 0) := "0001110";
constant r      : STD_LOGIC_VECTOR(6 downto 0) := "0101111";
constant empty   : STD_LOGIC_VECTOR(6 downto 0) := "1111111";

alias DIGIT_0 is VALUE (3 downto 0);
alias DIGIT_1 is VALUE (7 downto 4);
alias DIGIT_2 is VALUE (11 downto 8);
alias DIGIT_3 is VALUE (15 downto 12);

signal CATHODES_FOR_DIGIT      : STD_LOGIC_VECTOR(6 downto 0) := (others => '0');
signal NIBBLE                  : STD_LOGIC_VECTOR(4 downto 0) := (others => '0');
signal DOT                     : STD_LOGIC;

begin

    DIGIT_SWITCHING: process(COUNTER, VALUE, ERR)
    begin
        case COUNTER is
            when "00" =>
                if(ERR = '0') then
                    NIBBLE <= '0' & DIGIT_0;
                    DOT <= '0';
                elsif(ERR = '1') then
                    NIBBLE <= "11101";
                    DOT <= '0';
                end if;
            when "01" =>
                if(ERR = '0') then
                    NIBBLE <= '0' & DIGIT_1;
                    DOT <= '0';
                elsif(ERR = '1') then
                    NIBBLE <= "11110";
                    DOT <= '1';
                end if;
            when "10" =>
                if(ERR = '0') then
                    NIBBLE <= '0' & DIGIT_2;
                    DOT <= '1';
                elsif(ERR = '1') then
                    NIBBLE <= "11110";
                    DOT <= '0';
                end if;
            when "11" =>
                if(ERR = '0') then
                    NIBBLE <= '0' & DIGIT_3;
                    DOT <= '0';
                elsif(ERR = '1') then
                    NIBBLE <= "11111";
                    DOT <= '0';
                end if;
            when others =>
                NIBBLE <= (others => '0');
        end case;
    end process;

    SEVENT_SEGMENT_DECODER_PROCESS: process(nibble)
    begin
        case NIBBLE is
            when "00000" => CATHODES_FOR_DIGIT <= zero;
            when "00001" => CATHODES_FOR_DIGIT <= one;
            when "00010" => CATHODES_FOR_DIGIT <= two;
            when "00011" => CATHODES_FOR_DIGIT <= three;
            when "00100" => CATHODES_FOR_DIGIT <= four;
            when "00101" => CATHODES_FOR_DIGIT <= five;
            when "00110" => CATHODES_FOR_DIGIT <= six;
            when "00111" => CATHODES_FOR_DIGIT <= seven;
            when "01000" => CATHODES_FOR_DIGIT <= eight;
            when "01001" => CATHODES_FOR_DIGIT <= nine;
            when "01010" => CATHODES_FOR_DIGIT <= a;
            when "01011" => CATHODES_FOR_DIGIT <= b;
            when "01100" => CATHODES_FOR_DIGIT <= c;
            when "01101" => CATHODES_FOR_DIGIT <= d;
            when "01110" => CATHODES_FOR_DIGIT <= e;
            when "01111" => CATHODES_FOR_DIGIT <= f;
            when "11111" => CATHODES_FOR_DIGIT <= e;
            when "11110" => CATHODES_FOR_DIGIT <= r;
            when "11101" => CATHODES_FOR_DIGIT <= empty;
            when others => CATHODES_FOR_DIGIT <= (others => '0');
        end case;
    end process SEVENT_SEGMENT_DECODER_PROCESS;

```

```
CATHODES <= not (DOT) & CATHODES_FOR_DIGIT;  
end behavioral;
```

File: anodes_manager.vhd

Il codice relativo al componente anodes_manager è uguale a quello riportato nel Paragrafo 8.5.

File: Converter.vhd

Il codice relativo al componente Converter è uguale a quello riportato nel Paragrafo 6.3.3.

File: Basys_250K.ucf

Codice vincoli board Basys

```
# clock pin for Basys rev E Board  
NET "CLOCK" LOC = "P54";  
  
# onboard 7seg display  
NET "CATHODES<0>" LOC = "P25";  
NET "CATHODES<1>" LOC = "P16";  
NET "CATHODES<2>" LOC = "P23";  
NET "CATHODES<3>" LOC = "P21";  
NET "CATHODES<4>" LOC = "P20";  
NET "CATHODES<5>" LOC = "P17";  
NET "CATHODES<6>" LOC = "P83";  
NET "CATHODES<7>" LOC = "P22";  
  
NET "ANODES<0>" LOC = "P34";  
NET "ANODES<1>" LOC = "P33";  
NET "ANODES<2>" LOC = "P32";  
NET "ANODES<3>" LOC = "P26";  
  
# Switches  
NET "SWITCH<0>" LOC = "P38";  
NET "SWITCH<1>" LOC = "P36";  
NET "SWITCH<2>" LOC = "P29";  
NET "SWITCH<3>" LOC = "P24";  
  
# Buttons  
NET "BTN_SAQ" LOC = "P69";  
NET "BTN_M" LOC = "P48";  
NET "BTN_START" LOC = "P47";  
NET "RESET" LOC = "P41";
```

10.5.1 Simulazione

File: Divider_On_Board_tb.vhd

Codice Testbench Divider On Board

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
  
ENTITY Divider_On_Board_tb IS  
END Divider_On_Board_tb;  
  
ARCHITECTURE behavior OF Divider_On_Board_tb IS  
  
-- Component Declaration for the Unit Under Test (UUT)
```

```

COMPONENT Divider_On_Board
PORT(
    CLOCK : IN std_logic;
    RESET : IN std_logic;
    SWITCH : IN std_logic_vector(3 downto 0);
    BTN_SAQ : IN std_logic;
    BTN_M : IN std_logic;
    BTN_START : IN std_logic;
    ANODES : OUT std_logic_vector(3 downto 0);
    CATHODES : OUT std_logic_vector(7 downto 0)
);
END COMPONENT;

--Inputs
signal CLOCK : std_logic := '0';
signal RESET : std_logic := '0';
signal SWITCH : std_logic_vector(3 downto 0) := (others => '0');
signal BTN_SAQ : std_logic := '0';
signal BTN_M : std_logic := '0';
signal BTN_START : std_logic := '0';

--Outputs
signal ANODES : std_logic_vector(3 downto 0);
signal CATHODES : std_logic_vector(7 downto 0);

-- Clock period definitions
constant CLOCK_period : time := 50 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: Divider_On_Board PORT MAP (
        CLOCK => CLOCK,
        RESET => RESET,
        SWITCH => SWITCH,
        BTN_SAQ => BTN_SAQ,
        BTN_M => BTN_M,
        BTN_START => BTN_START,
        ANODES => ANODES,
        CATHODES => CATHODES
    );

    -- Clock process definitions
    CLOCK_process :process
    begin
        CLOCK <= '0';
        wait for CLOCK_period/2;
        CLOCK <= '1';
        wait for CLOCK_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;

        wait for CLOCK_period*10;

        -- insert stimulus here
        SWITCH <= "1000";      -- 8
        BTN_SAQ     <= '1';

        wait for 12 ms;

        SWITCH <= "0100";      -- 4
        BTN_M       <= '1';
        BTN_SAQ     <= '0';

        wait for 12 ms;

        BTN_START    <= '1';
        BTN_M       <= '0';

        wait for 12 ms;
        BTN_START    <= '0';

        wait for 200 ms;

        SWITCH <= "1001";      -- 9
        BTN_SAQ     <= '1';
    end process;

```

```
wait for 12 ms;

SWITCH <= "0100";      -- 4
BTN_M    <= '1';
BTN_SAQ  <= '0';

wait for 12 ms;
BTN_START <= '1';
BTN_M     <= '0';

wait for 12 ms;
BTN_START <= '0';

wait for 200 ms;

SWITCH <= "1111";      -- 15
BTN_SAQ  <= '1';

wait for 12 ms;
SWITCH <= "1010";      -- 10
BTN_M    <= '1';
BTN_SAQ  <= '0';

wait for 12 ms;
BTN_START <= '1';
BTN_M     <= '0';

wait for 12 ms;
BTN_START <= '0';

wait;
end process;

END;
```



Figura 10.6: Simulazione Divisore On Board

Capitolo 11

Esercizio 11

11.1 Traccia

Sfruttando l'implementazione fornita dalla Digilent di un dispositivo UART (componente *RS232RefComp.vhd*), progettare ed implementare in VHDL i seguenti componenti:

- a. **UART_TAPPO**: il componente acquisisce una stringa di 8 bit (fornita attraverso gli switch della board di sviluppo) e la serializza tramite la sezione di trasmissione del dispositivo UART; l'output seriale della UART viene re-inviato in ingresso alla sezione di ricezione dello stesso dispositivo (configurazione a tappo), e il dato deserializzato viene visualizzato sui led della board di sviluppo.
- b. **2_UART**: il componente acquisisce una stringa di 8 bit (fornita dall'utente tramite gli switch della board di sviluppo), la serializza tramite la sezione di trasmissione di un primo dispositivo UART, la deserializza tramite la sezione di ricezione di un secondo dispositivo UART collegato a valle del primo, e mostra la stringa sui led della board di sviluppo.
- c. **UART_PC** (facoltativo): il componente realizza la comunicazione fra la board di sviluppo e un terminale seriale in esecuzione su PC (es. Termite), previa connessione di PC e board tramite dispositivo fisico RS232 (uno degli endpoint di comunicazione è rappresentato dal PC). Il componente deve poter acquisire una stringa di 8 bit che rappresenta un carattere in codifica ASCII (fornita attraverso gli switch della board di sviluppo), ed inviarla tramite il dispositivo UART al terminale in esecuzione sul PC, in cui il carattere viene visualizzato. Allo stesso modo, il componente deve essere in grado di ricevere attraverso lo stesso dispositivo UART (oppure una seconda UART) un carattere trasmesso dal terminale e mostrarlo sui led.

11.2 Introduzione

Per la risoluzione degli esercizi proposti è stato necessario un profondo studio della periferica UART e della particolare implementazione fornita dalla Digilent. Sono stati quindi realizzati i componenti richiesti, in cui è previsto l'utilizzo della periferica in diverse configurazioni.

11.3 UART_TAPPO

11.3.1 Soluzione

Una periferica UART, *Universal Asynchronous Receiver-Transmitter*, è un dispositivo hardware, di uso generale o dedicato, in grado di convertire dati da un formato parallelo ad un formato seriale asincrono, e viceversa. Ogni UART contiene un registro a scorrimento, elemento fondamentale per effettuare la conversione tra i due formati.

Di seguito si riporta in Figura 11.1 l'architettura della periferica UART fornita dalla Digilent suddivisa nelle due parti di ricezione e trasmissione.

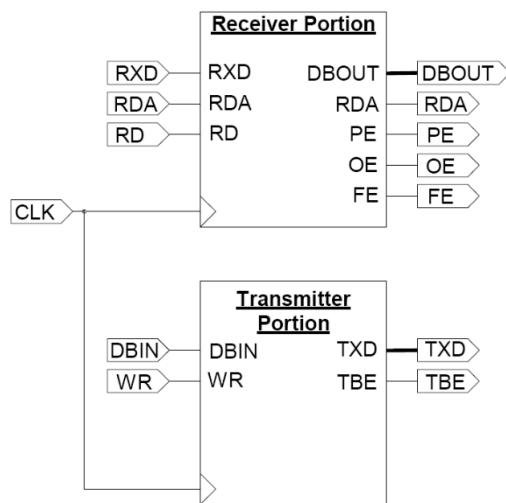


Figura 11.1: Architettura Periferica UART

Per ulteriori dettagli si rimanda a [3].

È stato realizzato il componente UART_Tappo come richiesto dalla traccia, utilizzando il componente fornito dalla Digilent UARTcomponent. Al fine di ottenere il comportamento desiderato, è stata connessa l'uscita TXD della porzione trasmettitore della periferica con l'ingresso RXD della porzione ricevitore della stessa.

11.3.2 Schematici

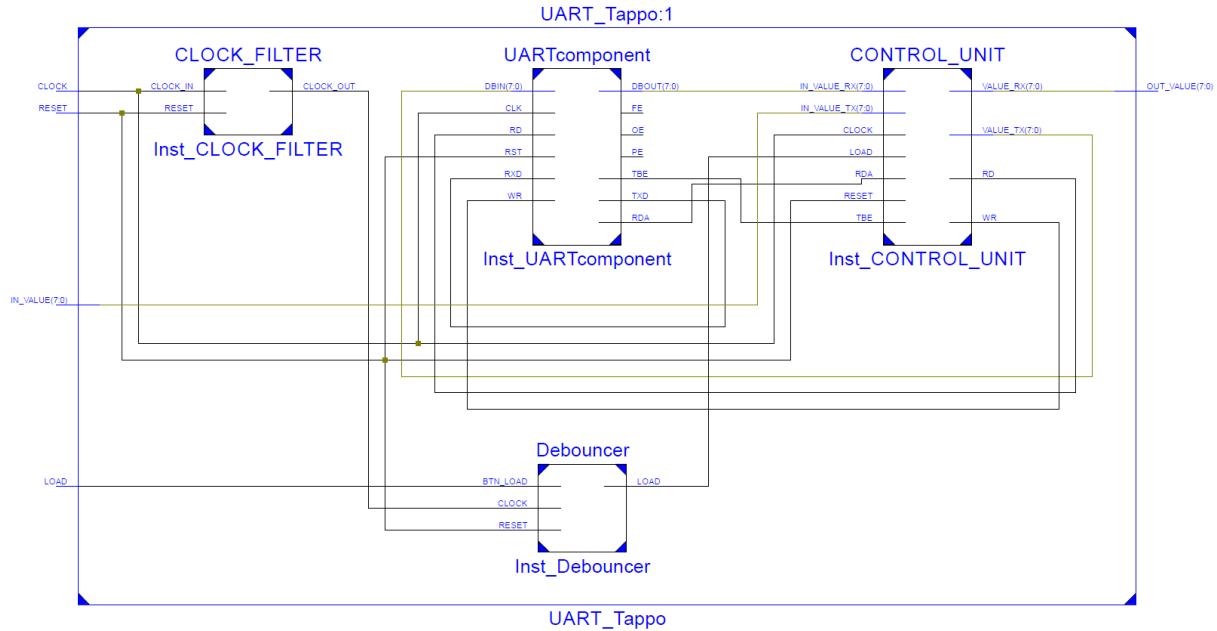


Figura 11.2: Schematico UART configurazione Tappo

11.3.3 Sintesi

È stata effettuata la sintesi del dispositivo sulla board ed il relativo collegamento con alcune delle sue periferiche. A tal fine sono stati prodotti ulteriori file VHDL.

Il dispositivo sintetizzato sulla board presenta una struttura multi-livello, come riportato in Figura 11.2. Al livello più alto si trova il *top-level-module*, ovvero **UART_Tappo** (**structural**). Tale componente definisce i segnali di ingresso e uscita del dispositivo e descrive le connessioni tra i vari componenti appartenenti al livello inferiore:

- **UARTcomponent** (**behavioral**): rappresenta l'implementazione fornita dalla Digilent di una periferica UART.
- **clock_filter** (**behavioural**): componente responsabile della divisione in frequenza del segnale di clock, del tutto identico a quello descritto nel Paragrafo 5.5.
- **Debouncer** (**behavioural**): ricevuti in ingresso i segnali di clock, reset e un segnale di abilitazione alla lettura dagli switch, si occupa di effettuare il *debouncing* del pulsante di lettura. Tale componente è analogo a quello descritto nel Paragrafo 5.5.
- **control_unit** (**behavioural**): ricevuti in ingresso i segnali di clock, reset e un segnale di abilitazione, gestisce trasmissione e ricezione dei dati attraverso la periferica UART.

File: UART_Tappo.vhd

Si evidenziano di seguito alcune delle connessioni tra i componenti istanziati nel *top-level module*:

- **UARTcomponent**: riceve in ingresso come segnale di clock il clock della scheda, come dati di ingresso e segnali di lettura/scrittura i segnali in uscita dalla **control_unit**. I dati di uscita sono invece connessi ai led presenti sulla board.
- **Debouncer**: riceve come segnale di clock l'uscita del componente **clock_filter**. Il segnale di caricamento dei dati da trasmettere è associato ad uno dei pulsanti presenti sulla scheda.
- **control_unit**: riceve come segnale di clock il clock della scheda, come dati da trasmettere il valore associato agli switch e come segnale di abilitazione il segnale in uscita dal Debouncer. Riceve inoltre in ingresso i segnali di stato in uscita dalla periferica, in modo da poterne gestire trasmissione e ricezione.

Codice UART Tappo

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity UART_Tappo is
    port(
        CLOCK : in      STD_LOGIC;
        RESET : in      STD_LOGIC;
        LOAD  : in      STD_LOGIC;
        IN_VALUE : in    STD_LOGIC_VECTOR(7 downto 0);
        OUT_VALUE : out   STD_LOGIC_VECTOR(7 downto 0));
end UART_Tappo;

architecture structural of UART_Tappo is

component UARTcomponent
    port(
        RXD : IN STD_LOGIC;
        CLK : IN STD_LOGIC;
        DBIN : IN STD_LOGIC_VECTOR(7 downto 0);
        RD : IN STD_LOGIC;
        WR : IN STD_LOGIC;
        RST : IN STD_LOGIC;
        RDA : INOUT STD_LOGIC;
        TXD : OUT STD_LOGIC;
        DBOUT : OUT STD_LOGIC_VECTOR(7 downto 0);
        TBE : OUT STD_LOGIC;
        PE : OUT STD_LOGIC;
        FE : OUT STD_LOGIC;
        OE : OUT STD_LOGIC);
end component;

component CONTROL_UNIT
    port(
        CLOCK : in      STD_LOGIC;
        RESET : in      STD_LOGIC;
        LOAD  : in      STD_LOGIC;
        IN_VALUE_TX : in    STD_LOGIC_VECTOR(7 downto 0);
        IN_VALUE_RX : in    STD_LOGIC_VECTOR(7 downto 0);
        TBE : in      STD_LOGIC;
        RDA : in      STD_LOGIC;
        VALUE_TX : out   STD_LOGIC_VECTOR(7 downto 0);
        VALUE_RX : out   STD_LOGIC_VECTOR(7 downto 0);
        WR : out     STD_LOGIC;
        RD : out     STD_LOGIC);
end component;

COMPONENT Debouncer
PORT(
    CLOCK : IN STD_LOGIC;
    RESET : IN STD_LOGIC;
    BTN_LOAD : IN STD_LOGIC;
    LOAD : OUT STD_LOGIC
);
END COMPONENT;

```

```

component CLOCK_FILTER
  port (
    CLOCK_IN : IN std_logic;
    RESET : IN std_logic;
    CLOCK_OUT : OUT std_logic);
end component;

signal CLOCK_FX      : std_logic;
signal VALUE_TX_TEMP : std_logic_vector(7 downto 0);
signal VALUE_RX_TEMP : std_logic_vector(7 downto 0);
signal TBE_TEMP      : std_logic;
signal WR_TEMP       : std_logic;
signal RD_TEMP       : std_logic;
signal TEMP          : std_logic;
signal RDA_TEMP      : std_logic;
signal LOAD_TEMP     : std_logic;

begin

  Inst_UARTcomponent: UARTcomponent PORT MAP(
    TXD => TEMP,
    RXD => TEMP,
    CLK => CLOCK,      -- Clock board
    DBIN => VALUE_TX_TEMP,   -- Uscita control unit
    DBOUT => VALUE_RX_TEMP,
    RDA => RDA_TEMP,
    TBE => TBE_TEMP,
    RD => RD_TEMP,      -- Uscita control unit
    WR => WR_TEMP,      -- Uscita control unit
    RST => RESET
  );

  Inst_CONTROL_UNIT: CONTROL_UNIT PORT MAP(
    CLOCK => CLOCK,
    RESET => RESET,
    LOAD => LOAD_TEMP,
    IN_VALUE_TX => IN_VALUE,
    IN_VALUE_RX => VALUE_RX_TEMP,
    TBE => TBE_TEMP,
    RDA => RDA_TEMP,
    VALUE_TX => VALUE_TX_TEMP,
    VALUE_RX => OUT_VALUE,   -- Led
    WR => WR_TEMP,
    RD => RD_TEMP
  );

  Inst_Debouncer: Debouncer PORT MAP(
    CLOCK => CLOCK_FX,
    RESET => RESET,
    BTN_LOAD => LOAD,
    LOAD => LOAD_TEMP
  );

  Inst_CLOCK_FILTER: CLOCK_FILTER PORT MAP(
    CLOCK_IN => CLOCK,
    RESET => RESET,
    CLOCK_OUT => CLOCK_FX
  );

end structural;

```

File: Clock_filter.vhd

Il codice relativo al componente `clock_filter` è uguale a quello riportato nel Paragrafo 5.5.

File: Control_unit.vhd

L'unità di controllo è il componente responsabile della gestione di trasmissione e ricezioni dei dati mediante la periferica UART.

In particolare, quando il segnale di abilitazione è alto, sul fronte di salita successivo del segnale di clock, i dati in ingresso vengono forniti alla periferica e viene alzato il segnale WR, in modo da abilitarne la trasmissione seriale. Abilitata la trasmissione, l'unità di controllo si pone in attesa del segnale TBE di avvenuta trasmissione. Rilevata tale

condizione, viene abilitata la lettura del buffer della porzione ricevitore, abbassando il segnale RD. Il completamento dell'operazione di lettura viene rilevata attraverso il segnale RDA. Rilevata tale condizione, vengono quindi visualizzati i dati ricevuti sui led della scheda e viene riportato il segnale RD al valore alto.

Codice Control Unit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CONTROL_UNIT is
    generic( N : integer := 4);
    port( CLOCK      : in STD_LOGIC;
          RESET       : in STD_LOGIC;
          LOAD        : in STD_LOGIC;
          IN_VALUE_TX : in std_logic_vector (7 downto 0);
          IN_VALUE_RX : in std_logic_vector (7 downto 0);
          TBE         : in STD_LOGIC;
          RDA         : in STD_LOGIC;
          VALUE_TX    : out std_logic_vector (7 downto 0);
          VALUE_RX    : out std_logic_vector (7 downto 0);
          WR          : out STD_LOGIC;
          RD          : out STD_LOGIC
    );
end CONTROL_UNIT;

architecture behavioral of CONTROL_UNIT is

signal VALUE_TX_TEMP : std_logic_vector (7 downto 0);
signal VALUE_RX_TEMP : std_logic_vector (7 downto 0);

begin
    INPUT_PROC : process (CLOCK, RESET)
    begin
        if(RESET = '1') then
            VALUE_TX_TEMP <= (others => '0');
            VALUE_RX_TEMP <= (others => '0');
        elsif(rising_edge(CLOCK)) then
            WR <= '0';
            if(LOAD = '1') then
                VALUE_TX_TEMP <= IN_VALUE_TX;
                WR <= '1';
            elsif(TBE = '1') then
                RD <= '0';
            elsif(RDA = '1') then
                RD <= '1';
                VALUE_RX_TEMP <= IN_VALUE_RX;
            end if;
        end if;
    end process INPUT_PROC;

    VALUE_TX <= VALUE_TX_TEMP;
    VALUE_RX <= VALUE_RX_TEMP;
end behavioral;

```

File: Debouncer.vhd

Codice Debouncer

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Debouncer is
    port( CLOCK      : in STD_LOGIC;
          RESET       : in STD_LOGIC;
          BTN_LOAD   : in STD_LOGIC;
          LOAD        : out STD_LOGIC);
end Debouncer;

architecture behavioral of Debouncer is

begin
    main: process(CLOCK, RESET)
    begin

```

```

if RESET = '1' then
    LOAD <= '0';
elsif rising_edge(clock) then
    if BTN_LOAD = '1' then
        LOAD <= '1';
    else
        LOAD <= '0';
    end if;
end if;
end process main;

end behavioral;

```

File: RS232RefComp2.vhd

Codice UART component

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-----
--Title:      UARTcomponent entity
--
--Inputs:      7      :      RXD
--              CLK
--              DBIN
--              RDA
--              RD
--              WR
--              RST
--
--Outputs:     7      :      TXD
--              DBOUT
--              RDA
--              TBE
--              PE
--              FE
--              OE
--
--Description: This describes the UART component entity. The inputs are
--              the Pegasus 50 MHz clock, a reset button, The RXD from
--              the serial cable, an 8-bit data bus from the parallel
--              port, and Read Data Available (RDA)and Transfer Buffer
--              Empty(TBE) handshaking signals. The outputs are the TXD
--              signal for the serial port, an 8-bit data bus for the
--              parallel port, RDA and TBE handshaking signals, and three
--              error signals for parity, frame, and overwrite errors.
--

entity UARTcomponent is
  Generic (
    --@50MHz
    BAUD_DIVIDE_G : integer := 326;      --9600 baud
    BAUD_RATE_G   : integer := 5210
  );
  Port (
    TXD      : out      std_logic      := '1';
    -- Transmitted serial data output
    RXD      : in       std_logic;
    -- Received serial data input
    CLK      : in       std_logic;
    DBIN     : in       std_logic_vector(7 downto 0);           -- Clock signal
    -- Input parallel data to be transmitted
    DBOUT    : out      std_logic_vector(7 downto 0);
    -- Received parallel data output
    RDA      : inout    std_logic;
    TBE      : out      std_logic      := '1';
    -- Transfer Buffer Emty
    RD       : in       std_logic;
    WR       : in       std_logic;
    PE       : out      std_logic;
    FE       : out      std_logic;
    OE       : out      std_logic;
    RST     : in       std_logic      := '0');                  -- Read Data Available
                                                               -- Read Strobe
                                                               -- Write Strobe
                                                               -- Parity error
                                                               -- Frame error
                                                               -- Overwrite error
                                                               -- Reset signal

```

```

end UARTcomponent;

architecture Behavioral of UARTcomponent is

-----  

-- Local Type and Signal Declarations  

-----  

-----  

--Title: Local Type Declarations  

--  

--Description: There are two state machines used in this entity. The  

-- rstate is used to synchronize the receiving portion of  

-- the UART, and the tstate is used to synchronize the  

-- sending portion of the UART.  

--  

-----  

type rstate is (
    strIdle,
    strEightDelay,
    strGetData,
    strWaitFor0,
    strWaitFor1,
    strCheckStop
);
  

type tstate is (
    sttIdle,
    sttTransfer,
    sttShift,
    sttDelay,
    sttWaitWrite
);
  

-----  

--Title: Local Signal Declarations  

--  

--Description: The constants and signals used by this entity are  

-- described below:  

--  

--      -baudRate : This is the Baud Rate constant used to  

-- synchronize the Pegasus 50 MHz clock with a  

-- baud rate of 9600. To get this number, divide  

-- 50MHz by 9600.  

--      -baudDivide : This is the Baud Rate divider used to safely  

-- read data transmitted at a baud rate of 9600.  

-- It is simply the above described baudRate  

-- constant divided by 16.  

--  

--      -rdReg : this is the receive holding register  

--      -rdSReg : this is the receive shift register  

--      -tfReg : this is the transfer holding register  

--      -tfSReg : this is the transfer shifting register  

--      -clkDiv : counter used to get rClk  

--      -ctr : used for delay times  

--      -tfCtr : used to delay in the transfer process  

--      -dataCtr : counts the number of read data bits  

--      -parError : parity error bit  

--      -frameError : frame error bit  

--      -CE : clock enable bit for the writing latch  

--      -ctRst : reset for the ctr  

--      -load : load signal used to load the transfer shift  

--              register  

--      -shift : shift signal used to unload the transfer  

--              shift register  

--      -par : represents the parity in the transfer  

--              holding register  

--      -tClkRST : reset for the tfCtr  

--      -rShift : shift signal used to load the receive shift  

--              register  

--      -dataRST : reset for the dataCtr  

--      -dataIncr : signal to increment the dataCtr  

--      -tfIncr : signal to increment the tfCtr  

--      -tDelayCtr : counter used to delay the transfer state  

--                  machine.  

--      -tDelayRst : reset signal for the tDelayCtr counter.  

--  

-- The following signals are used by the two state machines  

-- for state control:  

--      -Receive State Machine : strCur, strNext  

--      -Transfer State Machine : sttCur, sttNext
-----
```

```

constant baudRate      : std_logic_vector(12 downto 0)
:= conv_std_logic_vector(BAUD_RATE_G,13);
constant baudDivide   : std_logic_vector(8 downto 0)
:= conv_std_logic_vector(BAUD_DIVIDE_G-1,9);

signal rdReg          : std_logic_vector(7 downto 0)      := "00000000";
signal rdSReg         : std_logic_vector(9 downto 0)      := "1111111111";
signal tfReg          : std_logic_vector(7 downto 0);
signal tfSReg         : std_logic_vector(10 downto 0)     := "111111111111";
signal clkDiv          : std_logic_vector(9 downto 0)      := "0000000000";
signal ctr             : std_logic_vector(3 downto 0)      := "0000";
signal tfCtr          : std_logic_vector(3 downto 0)      := "0000";
signal dataCtr        : std_logic_vector(3 downto 0)      := "0000";
signal parError        : std_logic;
signal frameError     : std_logic;
signal CE              : std_logic;
signal ctRst          : std_logic      := '0';
signal load            : std_logic      := '0';
signal shift           : std_logic      := '0';
signal par             : std_logic;
signal tClkRST         : std_logic      := '0';
signal rShift          : std_logic      := '0';
signal dataRST         : std_logic      := '0';
signal dataIncr        : std_logic      := '0';
signal tfIncr          : std_logic      := '0';
signal tDelayCtr       : std_logic_vector (12 downto 0);
signal tDelayRst       : std_logic      := '0';

signal strCur          : rstate      := strIdle;
signal strNext         : rstate;
signal sttCur          : tstate      := sttIdle;
signal sttNext         : tstate;

-----  

-- Module Implementation
-----
begin

--  

--Title: Initial signal definitions
--  

--Description: The following lines of code define 4 internal and 1
--external signal. The most significant bit of the rdSReg
--signifies the frame error bit, so frameError is tied to
--that signal. The parError is high if there is a parity
--error, so it is set equal to the inverse of rdSReg(8)
--XOR-ed with the data bits. In this manner, it can
--determine if the parity bit found in rdSReg(8) matches
--the data bits. The parallel information output is equal
--to rdReg, so DBOUT is set equal to rdReg. Likewise, the
--input parallel information is equal to DBIN, so tfReg is
--set equal to DBIN. Because the tfSReg is used to shift
--out transmitted data, the TXD port is set equal to the
--first bit of tfSReg. Finally, the par signal represents
--the parity of the data, so par is set to the inverse of
--the data bits XOR-ed together. This UART can be changed
--to use EVEN parity if the "not" is omitted from the par
--definition.
--  

-----  

frameError <= not rdSReg(9);
parError <= not ( rdSReg(8) xor (((rdSReg(0) xor rdSReg(1)) xor
(rdSReg(2) xor rdSReg(3))) xor ((rdSReg(4) xor rdSReg(5)) xor
(rdSReg(6) xor rdSReg(7)))) );
DBOUT <= rdReg;
tfReg <= DBIN;
TXD <= tfSReg(0);
par <= not (((tfReg(0) xor tfReg(1)) xor (tfReg(2) xor tfReg(3))) xor
((tfReg(4) xor tfReg(5)) xor (tfReg(6) xor tfReg(7))) );
-----  

--  

--Title: Clock Divide counter
--  

--Description: This process defines clkDiv as a signal that increments
--with the clock up until it is either reset by ctRst, or
>equals baudDivide. This signal is used to define a
--counter called ctr that increments at the rate of the
--divided baud rate.
--  

-----  

process (CLK, clkDiv)
begin
    if (CLK = '1' and CLK'event) then
        if (clkDiv = baudDivide or ctRst = '1') then
            clkDiv <= "0000000000";

```

```

        else
            clkDiv <= clkDiv +1;
        end if;
    end if;
end process;

--Title: Transfer delay counter
--Description: This process defines tDelayCtr as a counter that runs
--until it equals baudRate, or until it is reset by
--tDelayRst. This counter is used to measure delay times
--when sending data out on the TXD signal. When the
--counter is equal to baudRate, or is reset, it is set
--equal to 0.
--

process (CLK, tDelayCtr)
begin
    if (CLK = '1' and CLK'event) then
        if (tDelayCtr = baudRate or tDelayRst = '1') then
            tDelayCtr <= "0000000000000000";
        else
            tDelayCtr <= tDelayCtr+1;
        end if;
    end if;
end process;

--Title: ctr set up
--Description: This process sets up ctr, which uses clkDiv to count
--increase at a rate needed to properly receive data in
--from RXD. If ctRst is strobed, the counter is reset. If
--clkDiv is equal to baudDivide, then ctr is incremented
--once. This signal is used by the receiving state machine
--to measure delay times between RXD reads.
--

process (CLK)
begin
    if CLK = '1' and CLK'Event then
        if ctRst = '1' then
            ctr <= "0000";
        elsif clkDiv = baudDivide then
            ctr <= ctr + 1;
        else
            ctr <= ctr;
        end if;
    end if;
end process;

--Title: transfer counter
--Description: This process makes tfCtr increment whenever the tfIncr
--signal is strobed high. If the tClkRst signal is strobed
--high, the tfCtr is reset to "0000." This counter is used
--to keep track of how many data bits have been
--transmitted.
--

process (CLK, tClkRST)
begin
    if (CLK = '1' and CLK'event) then
        if tClkRST = '1' then
            tfCtr <= "0000";
        elsif tfIncr = '1' then
            tfCtr <= tfCtr +1;
        end if;
    end if;
end process;

--Title: Error and RDA flag controller
--Description: This process controls the error flags FE, OE, and PE, as
--well as the Read Data Available (RDA) flag. When CE goes
--high, it means that data has been read into the rdSReg.
--This process then analyzes the read data for errors, sets
--rdReg equal to the eight data bits in rdSReg, and flags
--RDA to indicate that new data is present in rdReg. FE
--and PE are simply equal to the frameError and parError
--signals. OE is flagged high if RDA is already high when
--CE is strobed. This means that unread data was still in

```

```

--          the rdReg when it was written over with the new data.
--

process (CLK, RST, RD, CE)
begin
    if RD = '1' or RST = '1' then
        FE <= '0';
        OE <= '0';
        RDA <= '0';
        PE <= '0';
    elsif CLK = '1' and CLK'event then
        if CE = '1' then
            FE <= frameError;
            PE <= parError;
            rdReg(7 downto 0) <= rdSReg (7 downto 0);
            if RDA = '1' then
                OE <= '1';
            else
                OE <= '0';
                RDA <= '1';
            end if;
        end if;
    end process;

--Title: Receiving shift register
--
--Description: This process controls the receiving shift register
--(rdSReg). Whenever rShift is high, implying that data
--needs to be shifted in, rdSReg is shifts in RXD to the
--most significant bit, while shifting its existing data
--right.
--

process (CLK, rShift)
begin
    if CLK = '1' and CLK'Event then
        if rShift = '1' then
            rdSReg <= (RXD & rdSReg(9 downto 1));
        end if;
    end if;
end process;

--Title: Incoming Data counter
--
--Description: This process controls the dataCtr to keep track of
--shifted values into the rdSReg. The dataCtr signal is
--incremented once every time dataIncr is strobed high.
--

process (CLK, dataRST)
begin
    if (CLK = '1' and CLK'event) then
        if dataRST = '1' then
            dataCtr <= "0000";
        elsif dataIncr = '1' then
            dataCtr <= dataCtr +1;
        end if;
    end if;
end process;

--Title: Receiving State Machine controller
--
--Description: This process takes care of the Receiving state machine
--movement. It causes the next state to be evaluated on
--each rising edge of CLK. If the RST signal is strobed,
--the state is changed to the default starting state,
--which is strIdle
--

process (CLK, RST)
begin
    if CLK = '1' and CLK'Event then
        if RST = '1' then -- nadj
            strCur <= strIdle;
        else
            strCur <= strNext;
        end if;
    end if;
end process;

```

```

--Title: Receiving State Machine
--
--Description: This process contains all of the next state logic for the
--             Receiving state machine.
--

process (strCur, ctr, RXD, dataCtr)
begin
    case strCur is
        --
        --Title: strIdle state
        --
        --Description: This state is the idle and startup default stage for the
        --             Receiving state machine. The machine stays in this state
        --             until the RXD signal goes low. When this occurs, the
        --             ctrRst signal is strobed to reset ctr for the next state,
        --             which is strEightDelay.
        --

        when strIdle =>
            dataIncr <= '0';
            rShift <= '0';
            dataRst <= '1';
            CE <= '0';
            ctRst <= '1';

            if RXD = '0' then
                strNext <= strEightDelay;
            else
                strNext <= strIdle;
            end if;
        --
        --Title: strEightDelay state
        --
        --Description: This state simply delays the state machine for eight clock
        --             cycles. This is needed so that the incoming RXD data
        --             signal is read in the middle of each data emission. This
        --             ensures an accurate RXD signal reading. ctr counts from
        --             0 to 8 to keep track of rClk cycles. When it equals 8
        --             (1000) the next state, strWaitFor0, is loaded. During
        --             this state, the dataRst signal is strobed high to reset
        --             the shift-in data counter (dataCtr).
        --

        when strEightDelay =>
            dataIncr <= '0';
            rShift <= '0';
            dataRst <= '1';
            CE <= '0';
            ctRst <= '0';

            if ctr(3 downto 0) = "1000" then
                strNext <= strWaitFor0;
            else
                strNext <= strEightDelay;
            end if;
        --
        --Title: strGetData state
        --
        --Description: In this state, the dataIncr and rShift signals are
        --             strobed high for one clock cycle. By doing this, the
        --             rdSReg shift register shifts in RXD once, while the
        --             dataCtr is incremented by one. This state simply
        --             captures the incoming data on RXD into the rdSReg shift
        --             register. The next state loaded is strWaitFor0, which
        --             starts the two delay states needed between data shifts.
        --

        when strGetData =>
            CE <= '0';
            dataRst <= '0';
            ctRst <= '0';
            dataIncr <= '1';
            rShift <= '1';

            strNext <= strWaitFor0;
        --
        --Title: strWaitFor0 state
        --
        --Description: This state is a delay state, which delays the receive

```

```

-- state machine if not all of the incoming serial data has
-- not been shifted in yet. If dataCtr does not equal 10
-- (1010), the state is stayed in until the fourth bit of
-- ctr is equal to 1. When this happens, half of the delay
-- has been achieved, and the second delay state is loaded,
-- which is strWaitFor1. If dataCtr does equal 10 (1010),
-- all of the needed data has been acquired, so the
-- strCheckStop state is loaded to check for errors and
-- reset the receive state machine.
--

when strWaitFor0 =>
    CE <= '0';
    dataRst <= '0';
    ctRst <= '0';
    dataIncr <= '0';
    rShift <= '0';

    if dataCtr = "1010" then
        strNext <= strCheckStop;
    elsif ctr(3) = '0' then
        strNext <= strWaitFor1;
    else
        strNext <= strWaitFor0;
    end if;

--Title: strEightDelay state
--
--Description: This state is much like strWaitFor0, except it waits for
the fourth bit of ctr to equal 1. Once this occurs, the
strGetData state is loaded in order to shift in the next
data bit from RXD. Because strWaitFor0 is the only state
that calls this state, no other signals need to be
checked.
--

when strWaitFor1 =>
    CE <= '0';
    dataRst <= '0';
    ctRst <= '0';
    dataIncr <= '0';
    rShift <= '0';

    if ctr(3) = '0' then
        strNext <= strWaitFor1;
    else
        strNext <= strGetData;
    end if;

--Title: strCheckStop state
--
--Description: This state allows the newly acquired data to be checked
for errors. The CE flag is strobed to start the
previously defined error checking process. This state is
passed straight through to the strIdle state.
--

when strCheckStop =>
    dataIncr <= '0';
    rShift <= '0';
    dataRst <= '0';
    ctRst <= '0';
    CE <= '1';
    strNext <= strIdle;
end case;
end process;

--Title: Transfer shift register controller
--
--Description: This process uses the load, shift, and clk signals to
control the transfer shift register (tfSReg). Once load
is equal to '1', the tfSReg gets a '1', the parity bit,
the data bits found in tfReg, and a '0'. Under this
format, the shift register can be used to shift out the
appropriate signal to serially transfer the data. The
data is shifted out of the tfSReg whenever shift = '1'.
--

process (load, shift, CLK, tfSReg)
begin
    if CLK = '1' and CLK'Event then
        if load = '1' then

```

```

        tfSReg (10 downto 0) <= ('1' & par & tfReg(7 downto 0) &'0');
    elsif shift = '1' then
        tfSReg (10 downto 0) <= ('1' & tfSReg(10 downto 1));
    end if;
end if;
end process;

<--Title: Transfer State Machine controller
--Description: This process takes care of the Transfer state machine
--movement. It causes the next state to be evaluated on
--each rising edge of CLK. If the RST signal is strobed,
--the state is changed to the default starting state, which
--is sttIdle.

process (CLK, RST)
begin
    if (CLK = '1' and CLK'Event) then
        if RST = '1' then
            sttCur <= sttIdle;
        else
            sttCur <= sttNext;
        end if;
    end if;
end process;

<--Title: Transfer State Machine
--Description: This process controls the next state logic in the
transfer state machine. The transfer state machine
controls the shift and load signals that are used to load
and transmit the parallel data in a serial form. It also
controls the Transmit Buffer Empty (TBE) signal that
indicates if the transmit buffer (tfSReg) is in use or
not.

process (sttCur, tfCtr, WR, tDelayCtr)
begin
    case sttCur is
<--Title: sttIdle state
--Description: This state is the idle and startup default stage for the
transfer state machine. The state is stayed in until
the WR signal goes high. Once it goes high, the
sttTransfer state is loaded. The load and shift signals
are held low in the sttIdle state, while the TBE signal
is held high to indicate that the transmit buffer is not
currently in use. Once the idle state is left, the TBE
signal is held low to indicate that the transfer state
machine is using the transmit buffer.

when sttIdle =>
    TBE <= '1';
    tClkRST <= '0';
    tfIncr <= '0';
    shift <= '0';
    load <= '0';
    tDelayRst <= '1';

    if WR = '0' then
        sttNext <= sttIdle;
    else
        sttNext <= sttTransfer;
    end if;

<--Title: sttTransfer state
--Description: This state sets the load, tClkRST, and tDelayRst signals
high, while setting the TBE signal low. The load signal
is set high to load the transfer shift register with the
appropriate data, while the tClkRST and tDelayRst signals
are strobed to reset the tfCtr and tDelayCtr. The next
state loaded is the sttDelay state.

when sttTransfer =>
    TBE <= '0';

```

```

    shift <= '0';
    load <= '1';
    tClkRST <= '1';
    tfIncr <= '0';
    tDelayRst <= '1';

    sttNext <= sttDelay;
-----
--Title: sttShift state
--
--Description: This state strobes the shift and tfIncr signals high, and
--checks the tfCtr to see if enough data has been
--transmitted. By strobing the shift and tfIncr signals
--high, the tfSReg is shifted, and the tfCtr is incremented
--once. If tfCtr does not equal 9 (1001), then not all of
--the bits have been transmitted, so the next state loaded
--is the sttDelay state. If tfCtr does equal 9, the final
--state, sttWaitWrite, is loaded.
--

when sttShift =>
    TBE <= '0';
    shift <= '1';
    load <= '0';
    tfIncr <= '1';
    tClkRST <= '0';
    tDelayRst <= '0';

    if tfCtr = "1010" then
        sttNext <= sttWaitWrite;
    else
        sttNext <= sttDelay;
    end if;
-----
--Title: sttDelay state
--
--Description: This state is responsible for delaying the transfer state
--machine between transmissions. All signals are held low
--while the tDelayCtr is tested. Once tDelayCtr is equal
--to baudRate, the sttShift state is loaded.
--

when sttDelay =>
    TBE <= '0';
    shift <= '0';
    load <= '0';
    tClkRst <= '0';
    tfIncr <= '0';
    tDelayRst <= '0';

    if tDelayCtr = baudRate then
        sttNext <= sttShift;
    else
        sttNext <= sttDelay;
    end if;
-----
--Title: sttWaitWrite state
--
--Description: This state checks to make sure that the initial WR signal
--that triggered the transfer state machine has been
--brought back low. Without this state, a write signal
--that is held high for a long time will result in multiple
--transmissions. Once the WR signal is low, the sttIdle
--state is loaded to reset the transfer state machine.
--

when sttWaitWrite =>
    TBE <= '0';
    shift <= '0';
    load <= '0';
    tClkRst <= '0';
    tfIncr <= '0';
    tDelayRst <= '0';

    if WR = '1' then
        sttNext <= sttWaitWrite;
    else
        sttNext <= sttIdle;
    end if;
end case;
end process;
end Behavioral;

```

File: Basys_250K.ucf

Codice vincoli board Basys

```
# clock pin for Basys rev E Board
NET "CLOCK"      LOC = "P54";

# Leds
NET "OUT_VALUE<0>"    LOC = "P15";
NET "OUT_VALUE<1>"    LOC = "P14";
NET "OUT_VALUE<2>"    LOC = "P8";
NET "OUT_VALUE<3>"    LOC = "P7";
NET "OUT_VALUE<4>"    LOC = "P5";
NET "OUT_VALUE<5>"    LOC = "P4";
NET "OUT_VALUE<6>"    LOC = "P3";
NET "OUT_VALUE<7>"    LOC = "P2";

# Switches
NET "IN_VALUE<0>"    LOC = "P38";
NET "IN_VALUE<1>"    LOC = "P36";
NET "IN_VALUE<2>"    LOC = "P29";
NET "IN_VALUE<3>"    LOC = "P24";
NET "IN_VALUE<4>"    LOC = "P18";
NET "IN_VALUE<5>"    LOC = "P12";
NET "IN_VALUE<6>"    LOC = "P10";
NET "IN_VALUE<7>"    LOC = "P6";

# Buttons
NET "RESET"        LOC = "P69";
NET "LOAD"          LOC = "P48";
```

11.3.4 Simulazione

File: UART_Tappo_tb.vhd

Codice UART Tappo

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY UART_Tappo_tb IS
END UART_Tappo_tb;

ARCHITECTURE behavior OF UART_Tappo_tb IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT UART_Tappo
    PORT(
        CLOCK : IN std_logic;
        RESET : IN std_logic;
        LOAD : IN std_logic;
        IN_VALUE : IN std_logic_vector(7 downto 0);
        OUT_VALUE : OUT std_logic_vector(7 downto 0)
    );
    END COMPONENT;
    --Inputs
    signal CLOCK : std_logic := '0';
    signal RESET : std_logic := '0';
    signal LOAD : std_logic := '0';
    signal IN_VALUE : std_logic_vector(7 downto 0) := (others => '0');
    --Outputs
    signal OUT_VALUE : std_logic_vector(7 downto 0);
    -- Clock period definitions
    constant CLOCK_period : time := 20 ns;
BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: UART_Tappo PORT MAP (
        CLOCK => CLOCK,
        RESET => RESET,
```

```

LOAD => LOAD,
IN_VALUE => IN_VALUE,
OUT_VALUE => OUT_VALUE
);

-- Clock process definitions
CLOCK_process :process
begin
    CLOCK <= '0';
    wait for CLOCK_period/2;
    CLOCK <= '1';
    wait for CLOCK_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100 ns.
    wait for 100 ms;

    wait for CLOCK_period*10;

    -- insert stimulus here
    IN_VALUE <= "00000011";
    LOAD <= '1';

    wait for 6 ms;

    LOAD <= '0';

    wait for 100 ms;

    IN_VALUE <= "00000101";
    LOAD <= '1';

    wait for 6 ms;

    LOAD <= '0';

    wait;
end process;

END;

```

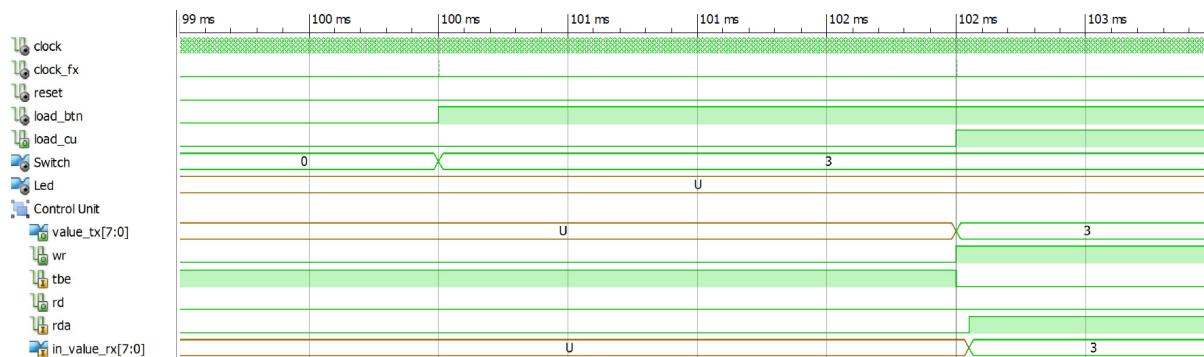


Figura 11.3: Simulazione UART Tappo fase di trasmissione

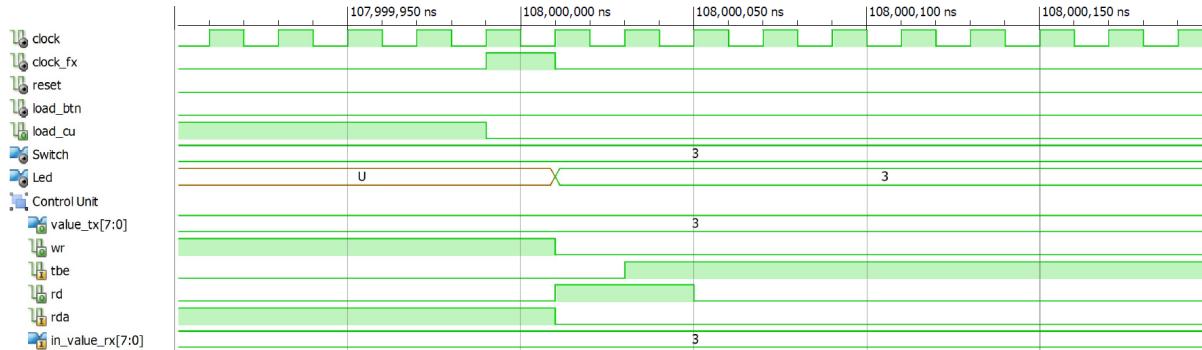


Figura 11.4: Simulazione UART Tappo fase di ricezione

11.4 2_UART

11.4.1 Soluzione

È stato realizzato il componente 2_UART come richiesto dalla traccia, utilizzando il componente fornito dalla Digilent UARTcomponent. Al fine di ottenere il comportamento desiderato, sono state istanziate due periferiche UART ed è stata connessa l'uscita TXD della porzione trasmettitore della prima periferica con l'ingresso RXD della porzione ricevitore della seconda.

11.4.2 Schematici

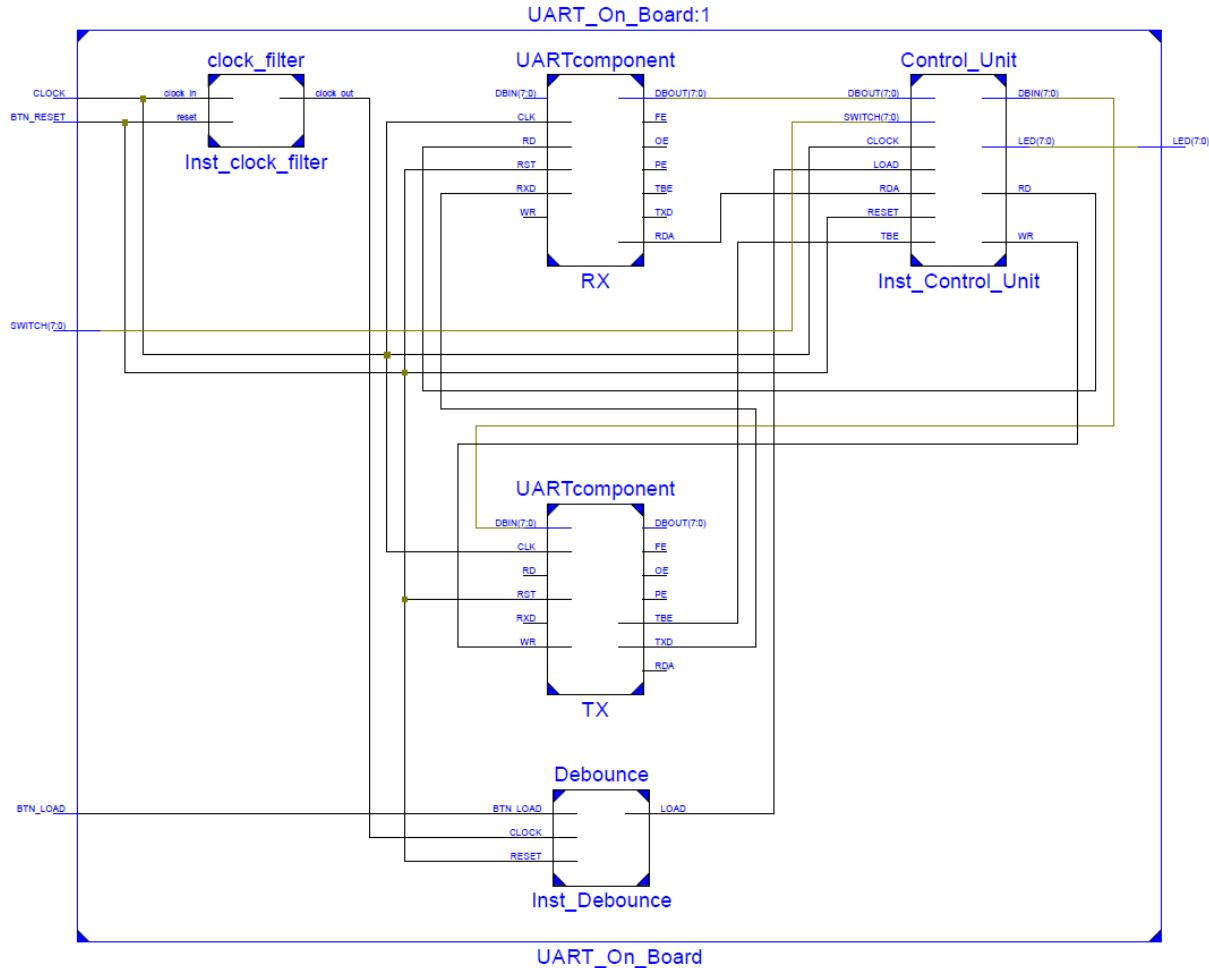


Figura 11.5: Schematico 2_UART

11.4.3 Sintesi

È stata effettuata la sintesi del dispositivo sulla board ed il relativo collegamento con alcune delle sue periferiche. A tal fine sono stati prodotti ulteriori file VHDL.

Il dispositivo sintetizzato sulla board presenta una struttura multi-livello, come riportato in Figura 11.5. Al livello più alto si trova il *top-level-module*, ovvero 2_UART (**structural**). Tale componente definisce i segnali di ingresso e uscita del dispositivo e descrive le connessioni tra i vari componenti appartenenti al livello inferiore:

1. **UARTcomponent (behavioral)**: rappresenta l'implementazione fornita dalla Digilent di un dispositivo UART.
Sono stati istanziate all'interno del *top-level-module* due periferiche UART, una abilitata in trasmissione, l'altra in ricezione.

2. **clock_filter (behavioural)**: componente responsabile della divisione in frequenza del segnale di clock, del tutto identico a quello descritto nel Paragrafo 5.5.
3. **Debouncer (behavioural)**: ricevuti in ingresso i segnali di clock, reset e un segnale di abilitazione alla lettura dagli switch, si occupa di effettuare il *debouncing* del pulsante di lettura.
Tale componente è analogo a quello descritto nel Paragrafo 5.5.
4. **control_unit (behavioural)**: ricevuti in ingresso i segnali di clock, reset e un segnale di abilitazione, gestisce trasmissione e ricezione attraverso le due periferiche UART.

File: **UART_On_Board.vhd**

Si evidenziano di seguito alcune delle connessioni tra i componenti istanziati nel *top-level module*:

- **UART_TX**: riceve in ingresso come segnale di clock il clock della scheda, come dati di ingresso e segnale di scrittura alcuni dei segnali in uscita dalla **control_unit**. L'uscita seriale è connessa alla seconda periferica UART.
- **UART_RX**: riceve in ingresso come segnale di clock il clock della scheda, come ingresso seriale l'uscita seriale della prima periferica e come segnale di scrittura uno dei segnali in uscita dalla **control_unit**. I dati di uscita sono invece connessi ai led presenti sulla board.
- **Debouncer**: riceve come segnale di clock l'uscita del componente **clock_filter**. Il segnale di caricamento dei dati da trasmettere è associato ad uno dei pulsanti presenti sulla scheda.
- **control_unit**: riceve come segnale di clock il clock della scheda, come dati da trasmettere il valore associato agli switch e come segnale di abilitazione il segnale in uscita dal Debouncer. Riceve inoltre in ingresso i segnali di stato in uscita dalle due periferiche, in modo da poterne gestire trasmissione e ricezione.

Codice UART On Board

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

entity UART_On_Board is
    port(
        CLOCK          : in      STD_LOGIC;
        BTN_RESET      : in      STD_LOGIC;
        BTN_LOAD       : in      STD_LOGIC;
        SWITCH         : in      STD_LOGIC_VECTOR(7 downto 0);
        LED            : out     STD_LOGIC_VECTOR(7 downto 0));
end UART_On_Board;

architecture structural of UART_On_Board is

COMPONENT UARTcomponent
    PORT(
        RXD : IN std_logic;
        CLK : IN std_logic;
        DBIN : IN std_logic_vector(7 downto 0);
        RD : IN std_logic;
```

```

WR : IN std_logic;
RST : IN std_logic;
RDA : INOUT std_logic;
TXD : OUT std_logic;
DBOUT : OUT std_logic_vector(7 downto 0);
TBE : OUT std_logic;
PE : OUT std_logic;
FE : OUT std_logic;
OE : OUT std_logic
);
END COMPONENT;

COMPONENT Control_Unit
PORT(
    CLOCK : IN std_logic;
    RESET : IN std_logic;
    LOAD : IN std_logic;
    TBE : IN std_logic;
    RD : OUT std_logic;
    SWITCH : IN std_logic_vector(7 downto 0);
    DBOUT : IN std_logic_vector(7 downto 0);
    RDA : IN std_logic;
    WR : OUT std_logic;
    LED : OUT std_logic_vector(7 downto 0);
    DBIN : OUT std_logic_vector(7 downto 0)
);
END COMPONENT;

COMPONENT Debounce
PORT(
    CLOCK : IN std_logic;
    RESET : IN std_logic;
    BTN_LOAD : IN std_logic;
    LOAD : OUT std_logic
);
END COMPONENT;

COMPONENT clock_filter
PORT(
    clock_in : IN std_logic;
    reset : IN std_logic;
    clock_out : OUT std_logic
);
END COMPONENT;

signal SERIAL_CONNECTION : std_logic;
signal RDA_TEMP : std_logic;
signal TBE_TEMP : std_logic;
signal DBIN_TEMP : std_logic_vector(7 downto 0);
signal DBOUT_TEMP : std_logic_vector(7 downto 0);
signal WR_TEMP : std_logic;
signal RD_TEMP : std_logic;
signal gnd_val :std_logic := '0';
signal high_vol :std_logic := '1';
signal LOAD_TEMP : std_logic;
signal CLOCK_FX : std_logic;

begin
TX: UARTcomponent PORT MAP(
    TXD => SERIAL_CONNECTION,
    CLK => CLOCK,
    DBIN => DBIN_TEMP,
    TBE => TBE_TEMP,
    WR => WR_TEMP,
    RST => BTN_RESET,
    RXD => gnd_val,
    RD => high_vol
);
RX: UARTcomponent PORT MAP(
    RXD => SERIAL_CONNECTION,
    CLK => CLOCK,
    RD => RD_TEMP,
    DBOUT => DBOUT_TEMP,
    RDA => RDA_TEMP,
    RST => BTN_RESET,
    WR => gnd_val,
    DBIN => "00000000"
);
Inst_Control_Unit: Control_Unit PORT MAP(
    CLOCK => CLOCK,
    RESET => BTN_RESET,
    LOAD => LOAD_TEMP,

```

```

TBE => TBE_TEMP,
RD => RD_TEMP,
RDA => RDA_TEMP,
SWITCH => SWITCH,
DBOUT => DBOUT_TEMP,
WR => WR_TEMP,
LED => LED,
DBIN => DBIN_TEMP
);

Inst_Debounce: Debounce PORT MAP (
    CLOCK => CLOCK_FX,
    RESET => BTN_RESET,
    BTN_LOAD => BTN_LOAD,
    LOAD => LOAD_TEMP
);

Inst_clock_filter: clock_filter PORT MAP (
    clock_in => CLOCK,
    reset => BTN_RESET,
    clock_out => CLOCK_FX
);

end structural;

```

File: clock_filter.vhd

Il codice relativo al componente `clock_filter` è uguale a quello riportato nel Paragrafo 5.5.

File: Control_unit.vhd

L'unità di controllo è il componente responsabile della gestione di trasmissione e ricezione dei dati mediante le due periferiche UART.

In particolare, quando il segnale di abilitazione è alto, sul fronte di salita successivo del segnale di clock, i dati in ingresso vengono forniti alla prima periferica e viene alzato il segnale WR, in modo da abilitarne la trasmissione seriale. Abilitata la trasmissione, l'unità di controllo si pone in attesa del segnale TBE di avvenuta trasmissione. Rilevata tale condizione, viene abilitata la lettura del buffer dalla seconda periferica UART, abbassando il segnale RD. Il completamento dell'operazione di lettura viene rilevata attraverso il segnale RDA. Rilevata tale condizione, vengono quindi visualizzati i dati ricevuti sui led della scheda e viene riportato il segnale RD al valore alto.

Codice Control Unit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Control_Unit is
    port(
        CLOCK : in STD_LOGIC;
        RESET : in STD_LOGIC;
        LOAD : in STD_LOGIC;
        TBE : in STD_LOGIC;
        RD : out STD_LOGIC;
        RDA : in STD_LOGIC;
        SWITCH : in STD_LOGIC_VECTOR (7 downto 0);
        DBOUT : in STD_LOGIC_VECTOR (7 downto 0);
        WR : out STD_LOGIC;
        LED : out STD_LOGIC_VECTOR (7 downto 0);
        DBIN : out STD_LOGIC_VECTOR (7 downto 0));
end Control_Unit;

architecture Behavioral of Control_Unit is
begin
    PROC : process(CLOCK, RESET)

```

```

begin
    if(RESET = '1') then
        LED <= (others =>'0');
        WR <= '0';
        DBIN <= (others =>'0');
        RD <= '1';
    elsif(rising_edge(CLOCK)) then
        WR <= '0';
        if(LOAD = '1') then
            DBIN <= SWITCH;
            WR <= '1';
        elsif(TBE = '1') then
            RD <= '0';
        elsif(RDA = '1') then
            LED <= DBOUT;
            RD <= '1';
        end if;
    end if;
end process PROC;

end Behavioral;

```

File: Debounce.vhd

Codice Debounce

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Debounce is
    port(
        CLOCK          : in      STD_LOGIC;
        RESET          : in      STD_LOGIC;
        BTN_LOAD      : in      STD_LOGIC;
        LOAD           : OUT     STD_LOGIC);
end Debounce;

architecture Behavioral of Debounce is
begin
    DEB : process(CLOCK, RESET)
    begin
        if(RESET = '1')then
            LOAD <= '0';
        elsif(rising_edge(CLOCK)) then
            if(BTN_LOAD = '1') then
                LOAD <= '1';
            else
                LOAD <= '0';
            end if;
        end if;
    end process DEB;
end Behavioral;

```

File: RS232RefComp2.vhd

Il codice relativo al componente UARTcomponent è uguale a quello riportato nel Paragrafo 11.3.3.

File: Basys _ 250K.ucf

Codice vincoli board Basys

```

# clock pin for Basys rev E Board
NET "CLOCK"      LOC = "P54";

# Leds
NET "LED<0>"    LOC = "P15";

```

```

NET "LED<1>"      LOC = "P14";
NET "LED<2>"      LOC = "P8";
NET "LED<3>"      LOC = "P7";
NET "LED<4>"      LOC = "P5";
NET "LED<5>"      LOC = "P4";
NET "LED<6>"      LOC = "P3";
NET "LED<7>"      LOC = "P2";

# Switches
NET "SWITCH<0>"    LOC = "P38";
NET "SWITCH<1>"    LOC = "P36";
NET "SWITCH<2>"    LOC = "P29";
NET "SWITCH<3>"    LOC = "P24";
NET "SWITCH<4>"    LOC = "P18";
NET "SWITCH<5>"    LOC = "P12";
NET "SWITCH<6>"    LOC = "P10";
NET "SWITCH<7>"    LOC = "P6";

# Buttons
NET "BTN_LOAD"       LOC = "P69";
NET "BTN_RESET"       LOC = "P48";

```

11.4.4 Simulazione

File: `UART_On_Board_tb.vhd`

Codice Testbench

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Uart_On_Board_tb IS
END Uart_On_Board_tb;

ARCHITECTURE behavior OF Uart_On_Board_tb IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT UART_On_Board
PORT(
    CLOCK : IN std_logic;
    BTN_RESET : IN std_logic;
    BTN_LOAD : IN std_logic;
    SWITCH : IN std_logic_vector(7 downto 0);
    LED : OUT std_logic_vector(7 downto 0)
);
END COMPONENT;

--Inputs
signal CLOCK : std_logic := '0';
signal BTN_RESET : std_logic := '0';
signal BTN_LOAD : std_logic := '0';
signal SWITCH : std_logic_vector(7 downto 0) := (others => '0');

--Outputs
signal LED : std_logic_vector(7 downto 0);

-- Clock period definitions
constant CLOCK_period : time := 20 ns;

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: UART_On_Board PORT MAP (
    CLOCK => CLOCK,
    BTN_RESET => BTN_RESET,
    BTN_LOAD => BTN_LOAD,
    SWITCH => SWITCH,
    LED => LED
);

-- Clock process definitions
CLOCK_process :process
begin
    CLOCK <= '0';
    wait for CLOCK_period/2;
    CLOCK <= '1';

```

```

        wait for CLOCK_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100 ns.
    wait for 10 ms;

    wait for CLOCK_period*10;

    -- insert stimulus here
    SWITCH <= "00000011";
    BTN_LOAD <= '1';

    wait for 6 ms;

    BTN_LOAD <= '0';

    wait for 50 ms;

    SWITCH <= "00010001";
    BTN_LOAD <= '1';

    wait for 6 ms;

    BTN_LOAD <= '0';

    wait;
end process;

END;

```

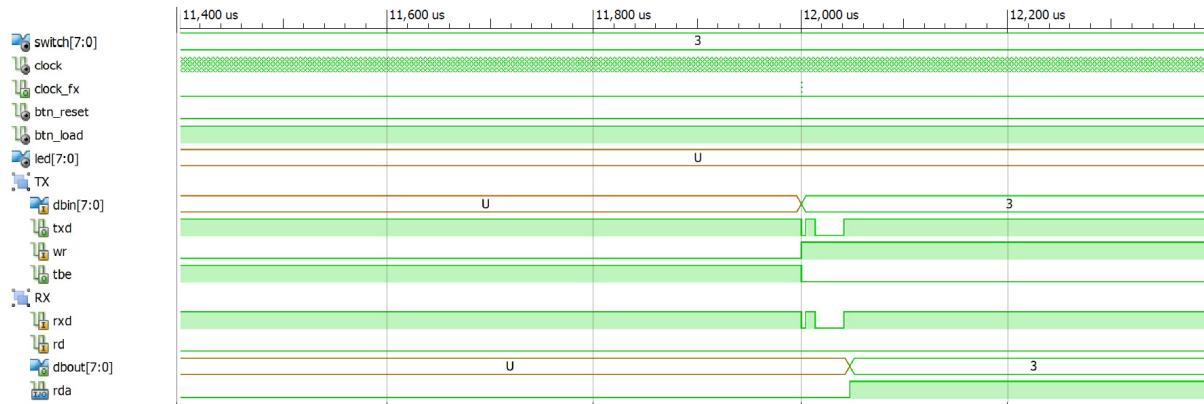


Figura 11.6: Simulazione 2_UART fase di trasmissione



Figura 11.7: Simulazione 2 _UART fase di ricezione

11.5 UART_PC

11.5.1 Soluzione

È stato realizzato il componente **UART_PC** come richiesto dalla traccia, utilizzando il componente fornito dalla Digilent **UARTcomponent**. Al fine di ottenere il comportamento desiderato, è stata connessa l'uscita **TXD** della porzione trasmettitore della periferica con l'uscita omonima della porta RS232 presente sulla board. Allo stesso modo l'ingresso **RXD** della porzione ricevitore è stato associato all'ingresso omonimo della porta RS232.

È bene osservare che per lo svolgimento dell'esercizio corrente, a differenza dei precedenti, è stata utilizzata la board di sviluppo **Nexys 2**, in quanto la Basys non dispone di una porta RS232.

Il collegamento tra la board di sviluppo e il PC è stato realizzato attraverso un apposito dispositivo fisico avente ad un capo una porta RS232, connessa alla board, e all'altro una porta USB, connessa al PC. Per gestire le operazioni di trasmissione e ricezione attraverso il PC è stato utilizzato il tool *Termite*. Per garantire la corretta comunicazione tra i dispositivi sono stati configurati opportunamente alcuni parametri, come illustrato in Figura 11.8.

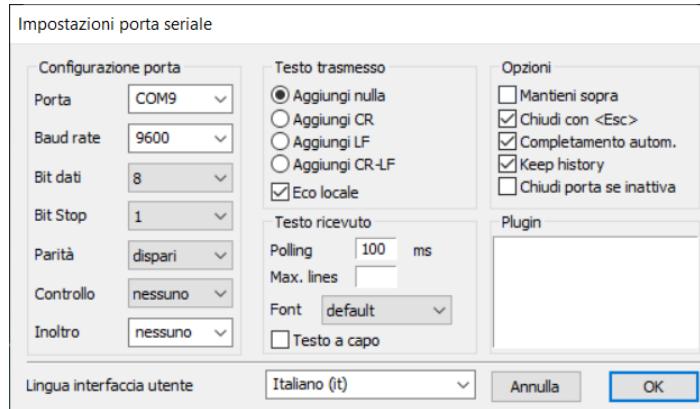


Figura 11.8: Impostazioni Termite

Per ottenere una comunicazione seriale con baud rate pari a 9600 è stato necessario inoltre impostare alcuni parametri definiti all'interno della periferica UART implementata dalla Digilent. In particolare, le costanti BAUD_DIVIDE_G e BAUD_RATE_G sono state poste rispettivamente a 326 e 5210. Tali valori sono stati ricavati attraverso le seguenti equazioni:

$$\text{BAUD_RATE_G} = \frac{\text{CLOCK_FREQUENCY}}{\text{BAUD_RATE}} = \frac{50 * 10^6}{9600}$$

$$\text{BAUD_DIVIDE_G} = \frac{\text{BAUD_RATE_G}}{16}$$

Per trasmettere i dati dalla scheda al PC è sufficiente impostare opportunamente gli switch della scheda e premere il pulsante di abilitazione alla trasmissione. In questo modo, verrà visualizzato nella console di Termite il carattere la cui codifica ASCII corrisponde al valore impostato mediante gli switch, come riportato in Figura 11.9.

Viceversa, per ricevere i dati inviati dal PC alla scheda è necessario tenere premuto il pulsante di abilitazione alla ricezione, digitare un carattere nella console di Termite e premere il tasto invio. In questo modo, verrà visualizzata sui led della scheda la codifica ASCII corrispondente al carattere trasmesso.

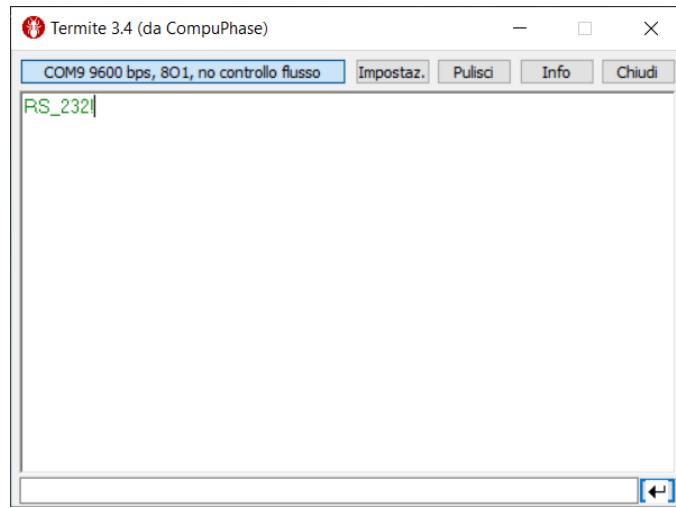


Figura 11.9: Trasmissione e ricezione attraverso Termite

11.5.2 Schematici

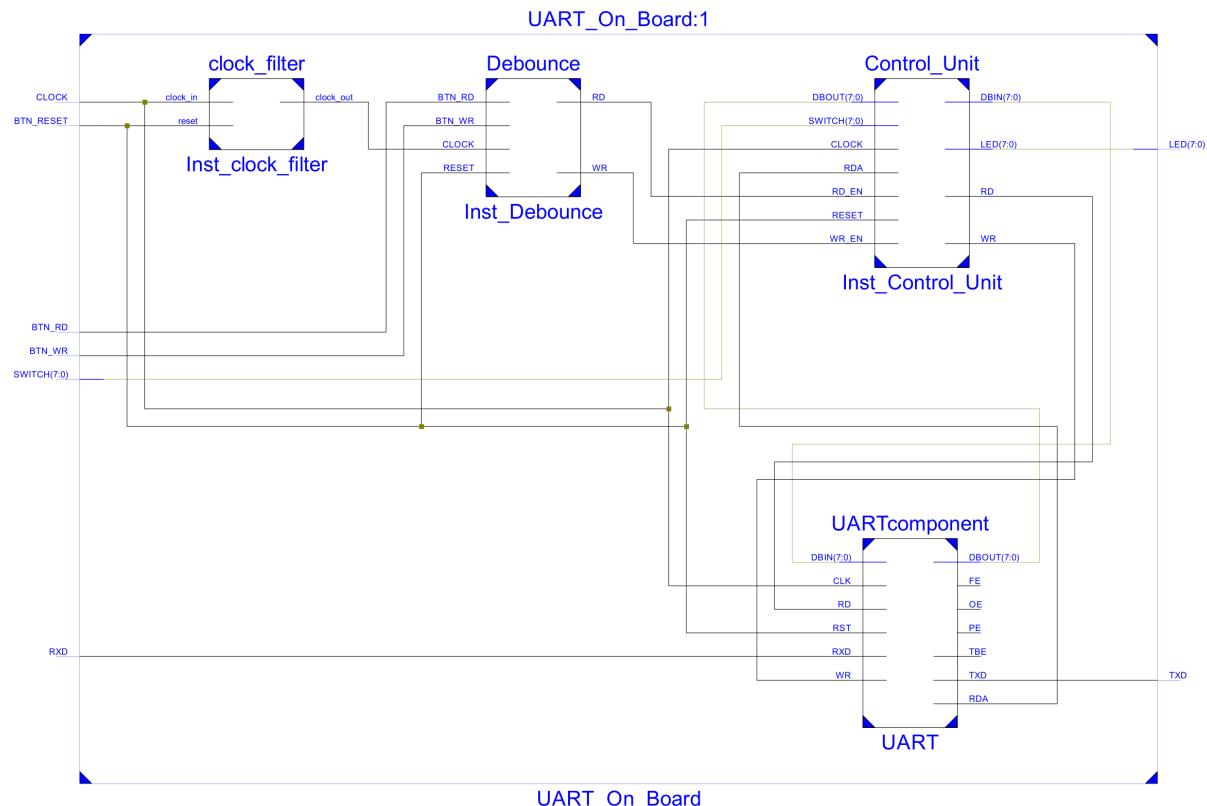


Figura 11.10: Schematico UART_PC

11.5.3 Sintesi

È stata effettuata la sintesi del dispositivo sulla board ed il relativo collegamento con alcune delle sue periferiche. A tal fine sono stati prodotti ulteriori file VHDL.

Il dispositivo sintetizzato sulla board presenta una struttura multi-livello, come riportato in Figura 11.10. Al livello più alto si trova il *top-level-module*, ovvero **UART_PC (structural)**. Tale componente definisce i segnali di ingresso e uscita del dispositivo e descrive le connessioni tra i vari componenti appartenenti al livello inferiore:

1. **UARTcomponent (behavioral)**: rappresenta l'implementazione fornita dalla Digilent di un dispositivo UART.
2. **clock_filter (behavioural)**: componente responsabile della divisione in frequenza del segnale di clock, del tutto identico a quello descritto nel Paragrafo 5.5.
3. **Debouncer (behavioural)**: ricevuti in ingresso i segnali di clock, reset e due segnali di abilitazione alla trasmissione e alla ricezione, si occupa di effettuare il *debouncing* dei pulsanti.
Tale componente è analogo a quello descritto nel Paragrafo 5.5.
4. **control_unit (behavioural)**: ricevuti in ingresso i segnali di clock, reset e un segnale di abilitazione, gestisce trasmissione e ricezione attraverso la periferica UART.

File: **UART_On_Board.vhd**

Si evidenziano di seguito alcune delle connessioni tra i componenti istanziati nel *top-level-module*:

- **UARTcomponent**: riceve in ingresso come segnale di clock il clock della scheda, come dati di ingresso e segnali di lettura/scrittura alcuni dei segnali in uscita dalla **control_unit**. I dati di uscita sono invece connessi ai led presenti sulla board.
- **Debouncer**: riceve come segnale di clock l'uscita del componente **clock_filter**. I segnali di trasmissione e ricezione sono associati a due dei pulsanti presenti sulla scheda.
- **control_unit**: riceve come segnale di clock il clock della scheda, come dati da trasmettere il valore associato agli switch e come segnali di abilitazione i segnali in uscita dal **Debouncer**. Riceve inoltre in ingresso i segnali di stato in uscita dalla periferica, in modo da poterne gestire trasmissione e ricezione.

Codice UART On Board

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

entity UART_On_Board is
  port(
    CLOCK : in STD_LOGIC;
    BTN_RESET : in STD_LOGIC;
    BTN_WR : in STD_LOGIC;
```

```

        BTN_RD      : in      STD_LOGIC;
        SWITCH     : in      STD_LOGIC_VECTOR (7 downto 0);
        RXD        : in      STD_LOGIC;
        LED         : out      STD_LOGIC_VECTOR (7 downto 0);
        TXD        : out      STD_LOGIC;
end UART_On_Board;

architecture structural of UART_On_Board is

COMPONENT UARTcomponent
PORT(
    RXD : IN std_logic;
    CLK : IN std_logic;
    DBIN : IN std_logic_vector(7 downto 0);
    RD : IN std_logic;
    WR : IN std_logic;
    RST : IN std_logic;
    RDA : INOUT std_logic;
    TXD : OUT std_logic;
    DBOUT : OUT std_logic_vector(7 downto 0);
    TBE : OUT std_logic;
    PE : OUT std_logic;
    FE : OUT std_logic;
    QE : OUT std_logic
);
END COMPONENT;

COMPONENT Control_Unit
PORT(
    CLOCK : IN std_logic;
    RESET : IN std_logic;
    SWITCH : IN std_logic_vector(7 downto 0);
    WR_EN : IN std_logic;
    RD_EN : IN std_logic;
    DBOUT : IN std_logic_vector(7 downto 0);
    RDA : IN std_logic;
    WR : OUT std_logic;
    RD : OUT std_logic;
    LED : OUT std_logic_vector(7 downto 0);
    DBIN : OUT std_logic_vector(7 downto 0)
);
END COMPONENT;

COMPONENT Debounce
PORT(
    CLOCK : IN std_logic;
    RESET : IN std_logic;
    BTN_WR : in      STD_LOGIC;
    BTN_RD : in      STD_LOGIC;
    WR : OUT std_logic;
    RD : OUT std_logic
);
END COMPONENT;

COMPONENT clock_filter
PORT(
    clock_in : IN std_logic;
    reset : IN std_logic;
    clock_out : OUT std_logic
);
END COMPONENT;

signal RDA_TEMP : std_logic;
signal TBE_TEMP : std_logic;
signal DBIN_TEMP : std_logic_vector(7 downto 0);
signal DBOUT_TEMP : std_logic_vector(7 downto 0);
signal WR_TEMP : std_logic;
signal RD_TEMP : std_logic;
signal LOAD_TEMP : std_logic;
signal CLOCK_FX : std_logic;
signal WR_EN_TEMP : std_logic;
signal RD_EN_TEMP : std_logic;
signal GROUND : std_logic := '0';

begin

UART: UARTcomponent PORT MAP(
    TXD => TXD,
    CLK => CLOCK,
    DBIN => DBIN_TEMP,
    TBE => TBE_TEMP,
    WR => WR_TEMP,
    RST => BTN_RESET,
    RXD => RXD,
    RD => RD_TEMP,

```

```

        DBOUT => DBOUT_TEMP,
        RDA => RDA_TEMP
    );
Inst_Control_Unit: Control_Unit PORT MAP (
    CLOCK => CLOCK,
    RESET => BTN_RESET,
    SWITCH => SWITCH,
    WR_EN => WR_EN_TEMP,
    RD_EN => RD_EN_TEMP,
    DBOUT => DBOUT_TEMP,
    RDA => RDA_TEMP,
    WR => WR_TEMP,
    RD => RD_TEMP,
    LED => LED,
    DBIN => DBIN_TEMP
);
Inst_Debounce: Debounce PORT MAP (
    CLOCK => CLOCK_FX,
    RESET => BTN_RESET,
    BTN_WR => BTN_WR,
    BTN_RD => BTN_RD,
    WR => WR_EN_TEMP,
    RD => RD_EN_TEMP
);
Inst_clock_filter: clock_filter PORT MAP (
    clock_in => CLOCK,
    reset => BTN_RESET,
    clock_out => CLOCK_FX
);
end structural;

```

File: Clock_filter.vhd

Il codice relativo al componente `clock_filter` è uguale a quello riportato nel Paragrafo 5.5.

File: Control_unit.vhd

L'unità di controllo è il componente responsabile della gestione di trasmissione e ricezione dei dati mediante la periferica UART.

In particolare, quando il segnale di abilitazione alla trasmissione è alto, sul fronte di salita successivo del segnale di clock, i dati in ingresso vengono forniti alla periferica e viene alzato il segnale WR, in modo da abilitarne la trasmissione seriale.

Viceversa, quando il segnale di abilitazione alla ricezione è alto, sul fronte di salita successivo del segnale di clock, viene abbassato il segnale RD, in modo da abilitare la periferica in ricezione. Il completamento dell'operazione di lettura viene rilevato attraverso il segnale RDA. Rilevata tale condizione, vengono quindi visualizzati i dati ricevuti sui led della scheda e viene riportato il segnale RD al valore alto.

Codice Control Unit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Control_Unit is
    port(
        CLOCK          : in   STD_LOGIC;
        RESET          : in   STD_LOGIC;
        SWITCH         : in   STD_LOGIC_VECTOR (7 downto 0);
        WR_EN          : in   STD_LOGIC;
        RD_EN          : in   STD_LOGIC;
        DBOUT          : in   STD_LOGIC_VECTOR (7 downto 0);
        RDA            : in   STD_LOGIC;
        WR             : out  STD_LOGIC;

```

```

RD           : out      STD_LOGIC;
LED          : out      STD_LOGIC_VECTOR (7 downto 0);
DBIN         : out      STD_LOGIC_VECTOR (7 downto 0));
end Control_Unit;

architecture Behavioral of Control_Unit is
begin
  PROC : process(CLOCK, RESET)
  begin
    if(RESET = '1') then
      LED <= (others =>'0');
      WR <= '0';
      DBIN <= (others =>'0');
    elsif(rising_edge(CLOCK)) then
      WR <= '0';
      RD <= '1';
      if(WR_EN = '1') then
        DBIN <= SWITCH;
        WR <= '1';
      elsif(RD_EN = '1') then
        RD <= '0';
      elsif(RDA = '1') then
        RD <= '1';
        LED <= DBOUT;
      end if;
    end if;
  end process PROC;
end Behavioral;

```

File: Debouncer.vhd

Codice Debouncer

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Debounce is
  port(
    CLOCK           : in      STD_LOGIC;
    RESET           : in      STD_LOGIC;
    BTN_WR : IN std_logic;
    BTN_RD : IN std_logic;
    WR : OUT std_logic;
    RD : OUT std_logic
  );
end Debounce;

architecture Behavioral of Debounce is
begin
  DEB : process(CLOCK, RESET)
  begin
    if(RESET = '1')then
      WR <= '0';
      RD <= '0';
    elsif(rising_edge(CLOCK)) then
      if(BTN_WR = '1') then
        WR <= '1';
      elsif(BTN_RD = '1') then
        RD <= '1';
      else
        WR <= '0';
        RD <= '0';
      end if;
    end if;
  end process DEB;
end Behavioral;

```

File: RS232RefComp2.vhd

Il codice relativo al componente UARTcomponent è uguale a quello riportato nel Paragrafo 11.3.3.

File: constraints_Nexys2_1200.ucf

Codice vincoli board Nexys 2

```
# clock pin for Nexys 2 Board
NET "CLOCK" LOC = "B8";

# Leds
NET "LED<0>" LOC = "J14";
NET "LED<1>" LOC = "J15";
NET "LED<2>" LOC = "K15";
NET "LED<3>" LOC = "K14";
NET "LED<4>" LOC = "E16";
NET "LED<5>" LOC = "P16";
NET "LED<6>" LOC = "E4";
NET "LED<7>" LOC = "P4";

# Switches
NET "SWITCH<0>" LOC = "G18";
NET "SWITCH<1>" LOC = "H18";
NET "SWITCH<2>" LOC = "K18";
NET "SWITCH<3>" LOC = "K17";
NET "SWITCH<4>" LOC = "L14";
NET "SWITCH<5>" LOC = "L13";
NET "SWITCH<6>" LOC = "N17";
NET "SWITCH<7>" LOC = "R17";

# Buttons
NET "BTN_RESET" LOC = "B18";
NET "BTN_WR" LOC = "D18";
NET "BTN_RD" LOC = "E18";

# RS232 connector
NET "RXD" LOC = "U6";
NET "TXD" LOC = "P9";
```

11.5.4 Simulazione

File: UART_On_Board_tb.vhd

Codice Testbench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY UART_On_Board_tb IS
END UART_On_Board_tb;

ARCHITECTURE behavior OF UART_On_Board_tb IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT UART_On_Board
        PORT(
            CLOCK : IN std_logic;
            BTN_RESET : IN std_logic;
            BTN_WR : IN std_logic;
            BTN_RD : IN std_logic;
            SWITCH : IN std_logic_vector(7 downto 0);
            RXD : IN std_logic;
            LED : OUT std_logic_vector(7 downto 0);
            TXD : OUT std_logic
        );
    END COMPONENT;
    -- Inputs
    signal CLOCK : std_logic := '0';
    signal BTN_RESET : std_logic;
    signal BTN_WR : std_logic;
    signal BTN_RD : std_logic;
    signal SWITCH : std_logic_vector(7 downto 0);
    signal RXD : std_logic;
    signal LED : std_logic_vector(7 downto 0);
    signal TXD : std_logic;
```

```

signal BTN_RESET : std_logic := '0';
signal BTN_WR : std_logic := '0';
signal BTN_RD : std_logic := '0';
signal SWITCH : std_logic_vector(7 downto 0) := (others => '0');
signal RXD : std_logic := '0';

--Outputs
signal LED : std_logic_vector(7 downto 0);
signal TXD : std_logic;

-- Clock period definitions
constant CLOCK_period : time := 20 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: UART_On_Board PORT MAP (
        CLOCK => CLOCK,
        BTN_RESET => BTN_RESET,
        BTN_WR => BTN_WR,
        BTN_RD => BTN_RD,
        SWITCH => SWITCH,
        RXD => RXD,
        LED => LED,
        TXD => TXD
    );

    -- Clock process definitions
    CLOCK_process :process
    begin
        CLOCK <= '0';
        wait for CLOCK_period/2;
        CLOCK <= '1';
        wait for CLOCK_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- insert stimulus here

        --Linea di ricezione alta in stato di riposo
        RXD <= '1';

        -- Simulazione fase di trasmissione
        wait for 30 ms;

        SWITCH <= "01011101";

        wait for 30 ms;

        BTN_WR <= '1';

        wait for 30 ms;

        BTN_WR <= '0';

        wait for 30 ms;

        BTN_RD <= '1';

        wait for 30 ms;

        -- Simulazione fase di ricezione
        RXD <= '0';
        wait for 4.64 us;
        RXD <= '1';
        wait for 4.64 us;
        RXD <= '0';
        wait for 4.64 us;
        RXD <= '1';
        wait for 4.64 us;
        RXD <= '0';
        wait for 4.64 us;
        RXD <= '0';
    end process;

```

```

wait for 4.64 us;
RXD <= '1';
wait for 4.64 us;
RXD <= '1';
wait for 4.64 us;

wait for 30 ms;

BTN_RD <= '0';

wait;
end process;
END;

```

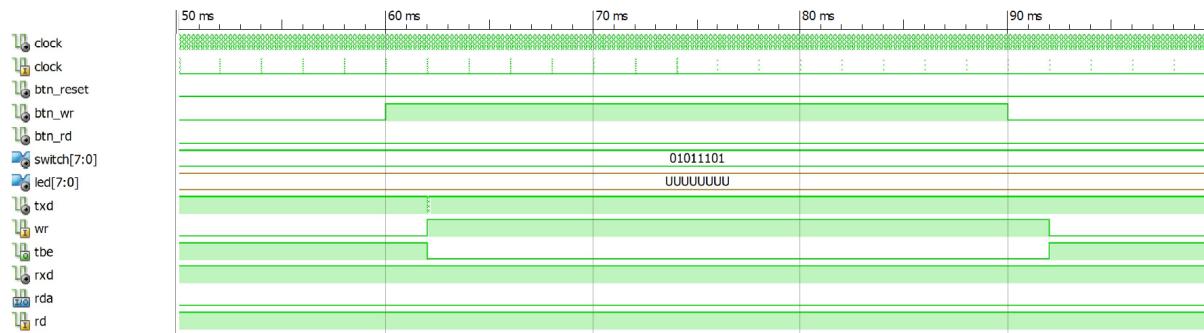


Figura 11.11: Simulazione UART _ PC fase di trasmissione

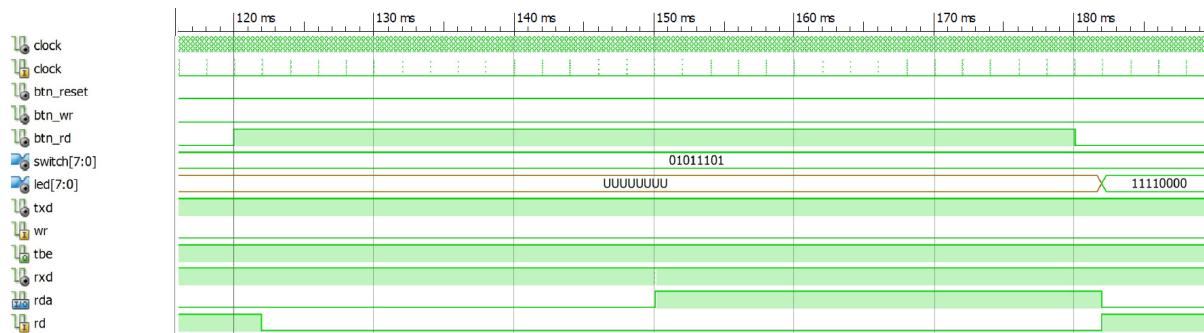


Figura 11.12: Simulazione UART _ PC fase di ricezione

Capitolo 12

Esercizio 12

12.1 Traccia

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

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

12.2 Introduzione

Per la risoluzione dell'esercizio proposto è stato necessario un profondo studio del processore *Mic-1*, il quale implementa un sottoinsieme delle istruzioni della Java Virtual Machine. Tale processore è detto anche IJVM, in quanto opera unicamente su valori interi. È stato poi portato avanti lo studio relativo alle istruzioni `BIPUSH` ed `IADD`, analizzando attentamente l'evoluzione delle rispettive microprocedure. Si è quindi scelto di modificare il comportamento dell'istruzione `IAND`, in modo da farle eseguire l'operazione di OR. Infine, è stato descritto il comportamento del processore in merito ad un'istruzione di I/O, nello specifico, è stata scelta l'istruzione `GSTORE`.

È bene osservare che la struttura di tale capitolo differisce da quella adottata sinora al fine di facilitare la trattazione del problema proposto.

12.3 Soluzione

Il processore Mic-1 possiede una caratteristica particolare: non dispone di registri generali per dati e, eventualmente, indirizzi. Tale processore presenta infatti un'architettura **a**

stack, secondo cui le istruzioni aritmetiche e logiche non hanno operandi espliciti, bensì impliciti. Difatti, gli operandi sono prelevati da una struttura *Last-In-First-Out* (LIFO) allocata nella memoria principale, in cui devono essere precedentemente caricati. Lo stato dello stack, naturalmente, evolve durante l'esecuzione delle istruzioni. Per eseguire una istruzione IJVM è quindi necessario controllare gli accessi in memoria, l'ALU, ed eseguire delle operazioni di *book-keeping*.

L'implementazione processore Mic-1 utilizzata nell'esercizio è prevede un'unità di controllo realizzata in **logica microprogrammata**. Ciascuna istruzione IJVM è implementata come una sequenza di microistruzioni, detta microprocedura. Un insieme di microistruzioni compone il microprogramma, il quale è tipicamente memorizzato in una micro-ROM interna al processore.

Per comprendere il funzionamento del processore ed analizzare le microistruzioni richieste per la realizzazione di una istruzione IJVM è necessario presentare l'architettura del processore stesso, mostrata in Figura 12.1.

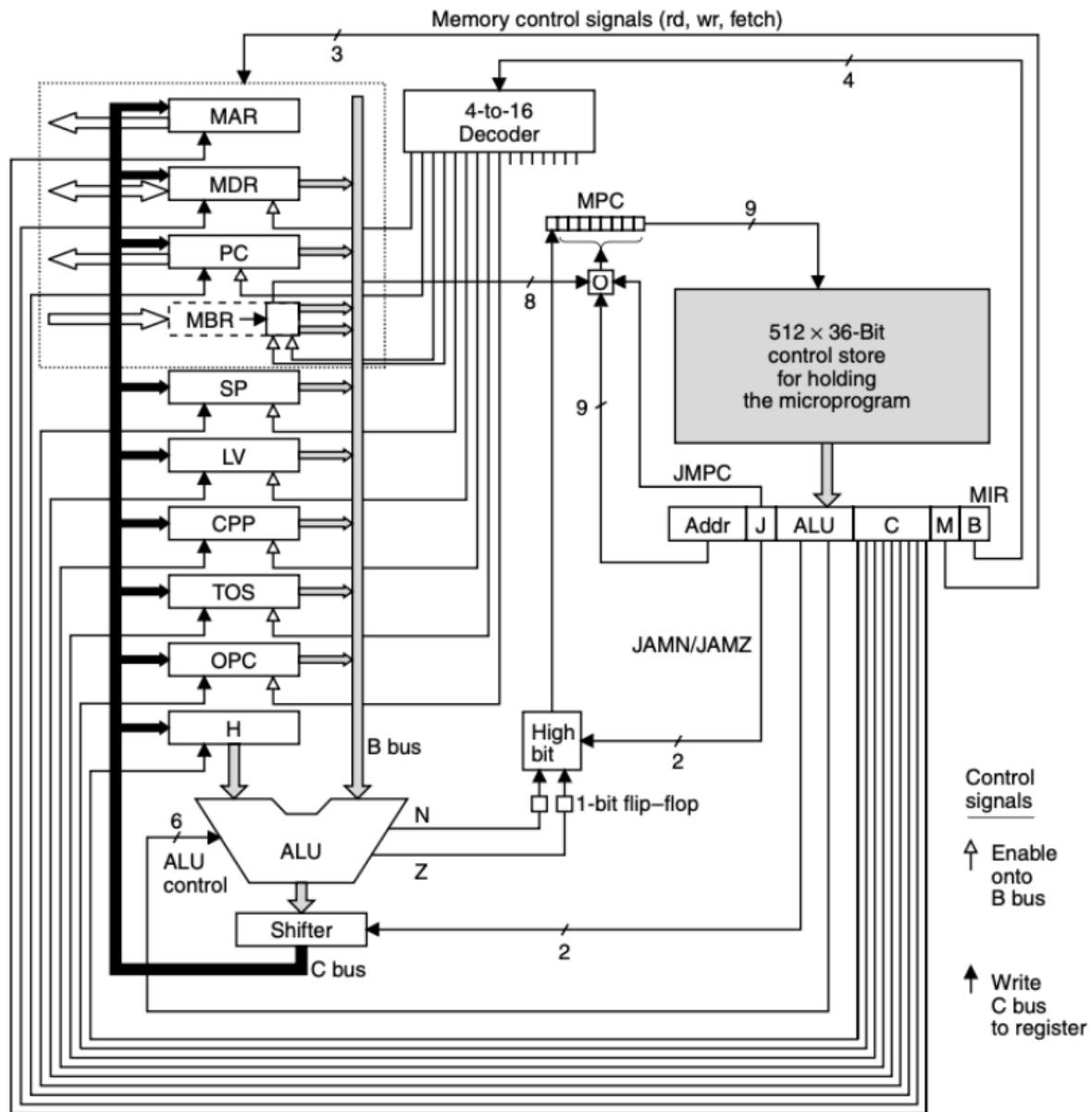


Figura 12.1: Architettura Processore Mic-1

Per ulteriori dettagli riguardanti il funzionamento del processore si rimanda a [4].

12.3.1 Esercizio A

Come richiesto dalla traccia è stata effettuata l'analisi di due istruzioni. In particolare si è scelto di analizzare le istruzioni BIPUSH ed IADD.

BIPUSH

Per analizzare il funzionamento dell'istruzione BIPUSH si è sviluppato il programma in linguaggio IJVM riportato di seguito:

Istruzione BIPUSH in linguaggio IJVM

```
.main
BIPUSH 0x56
.endmethod
```

Scrivere a mano le microistruzioni è possibile, ma è molto semplice commettere errori. Inoltre, il microprogramma così ottenuto sarebbe complicato da comprendere e modificare. Per semplificare la scrittura di un microprogramma è possibile dunque utilizzare ad un particolare linguaggio denominato **MAL** (Micro Assembly Language). A partire dalla specifica in linguaggio MAL di una istruzione IJVM, un particolare tool, detto **microassemblatore**, è in grado di ricavare le microistruzioni corrispondenti, espresse nel formato del processore *Mic-1*. Di seguito è riportata la specifica in linguaggio MAL dell'istruzione IJVM BIPUSH.

Specifiche dell'istruzione BIPUSH in linguaggio MAL

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

A questo punto è possibile proseguire con l'analisi delle microistruzioni eseguite dal processore al fine di realizzare l'istruzione BIPUSH. La simulazione dell'esecuzione è riportata in Figura 12.2.

Si noti che l'analisi dell'istruzione BIPUSH è stata eseguita a partire da 205 ns in quanto, nel lasso temporale precedente, sono eseguite alcune microistruzioni relative alle istruzioni IJVM INVOKEVIRTUAL e MAIN.

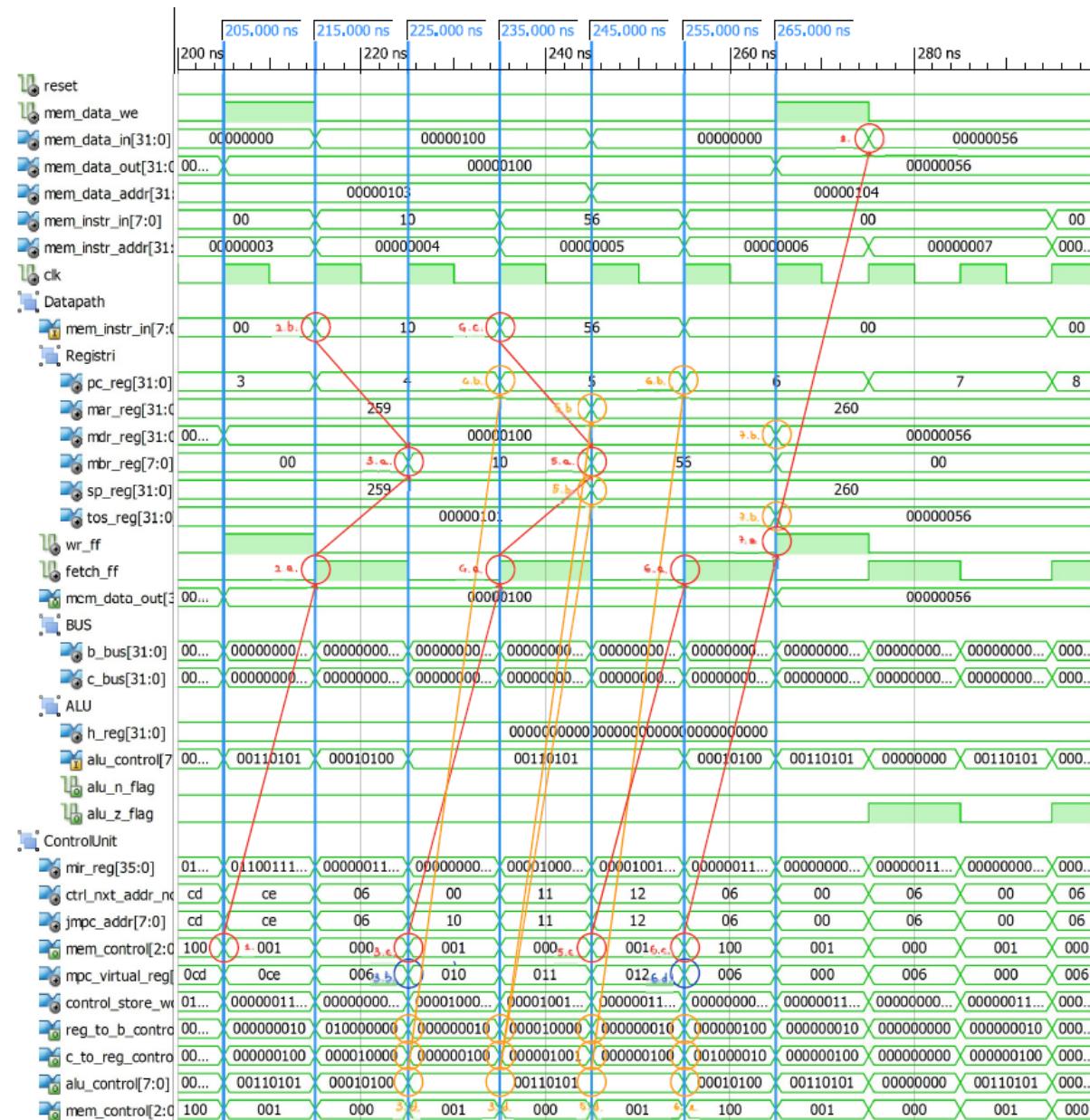


Figura 12.2: Simulazione esecuzione istruzione BIPUSH

Dalla simulazione si evincono le seguenti operazioni principali, necessarie alla corretta esecuzione dell'istruzione:

1. (205 ns) INVOKEVIRTUAL: Viene richiesto il fetch di un byte della prossima istruzione situata in RAM, impostando opportunamente i bit del registro mem_control, contenente i segnali di controllo della memoria RAM.
2. (215 ns)
 - (a) Viene alzato il segnale fetch_ff di richiesta di fetch verso la memoria.

- (b) Il valore del byte a cui si riferisce il Program Counter (PC) della RAM è 0x10, codice operativo dell'istruzione BIPUSH.
- (c) Viene inoltre eseguita l'ultima microistruzione dell'istruzione INVOKEVIRTUAL.
3. (225 ns) MAIN:
- (a) Viene prelevato il codice operativo dell'istruzione BIPUSH e caricato nel registro MBR
- (b) Viene aggiornato il valore del micro-Program Counter (MPC), in modo da eseguire la micropocedura associata all'istruzione BIPUSH: per ottenere il valore aggiornato del MPC viene eseguita l'operazione di OR bit a bit tra i registri NEXT_ADDR ed MBR. Tale operazione è indicata dalla presenza del bit JMP_C pari ad 1 all'interno del registro MIR, registro contenente la microistruzione correntemente eseguita. È bene osservare che l'indirizzo della prima microistruzione da eseguire è pari al codice operativo dell'istruzione stessa.
- (c) Viene richiesto il fetch di un byte della prossima istruzione situata in RAM, impostando opportunamente i bit del registro mem_control. Tale fetch è necessario per caricare l'operando dell'istruzione BIPUSH.
- (d) Vengono opportunamente impostati i segnali di controllo dei bus B e C e dell'ALU, in modo da incrementare il PC.
4. (235 ns) Inizio BIPUSH:
- (a) Viene alzato il segnale fetch_ff di richiesta di fetch verso la memoria.
- (b) Il valore del byte a cui si riferisce il PC della RAM è 0x56, valore dell'operando della BIPUSH.
- (c) Viene aggiornato il valore del PC.
- (d) Viene incrementato di uno il valore del MPC, in modo da eseguire la microistruzione successiva.
- (e) Vengono opportunamente impostati i segnali di controllo dei bus B e C e dell'ALU, in modo da incrementare il valore dello Stack Pointer (SP) e posizionare tale valore sia nel registro SP che nel registro MAR.
5. (245 ns)
- (a) Viene prelevato l'operando 0x56 della BIPUSH e caricato nel registro MBR.
- (b) Viene aggiornato il valore dei registri MAR e SP. Di questo modo sarà possibile caricare in memoria il contenuto del registro MDR all'indirizzo contenuto nel registro MAR.
- (c) Viene incrementato di uno il valore del MPC, in modo da eseguire la microistruzione successiva.
- (d) Viene richiesto il fetch di un byte della prossima istruzione situata in RAM, impostando opportunamente i bit del registro mem_control. Tale richiesta di fetch è necessaria a caricare nel registro MBR il primo byte della prossima

istruzione da eseguire. Al termine dell'esecuzione della BIPUSH, il controllo passa al MAIN, il quale porta in esecuzione l'istruzione presente nel registro MBR ed effettua la richiesta di una nuova fetch impostando opportunamente i bit del registro mem_control.

- (e) Vengono opportunamente impostati i segnali di controllo dei bus B e C e dell'ALU, in modo da incrementare il PC.

6. (255 ns)

- (a) Viene alzato il segnale fetch_ff di richiesta di fetch verso la memoria.
- (b) Viene aggiornato il valore del PC.
- (c) Viene richiesta la scrittura in memoria del contenuto del registro MDR, impostando opportunamente i bit del registro mem_control.
- (d) Viene aggiornato il valore del MPC, in modo da eseguire la microprocedura associata all'istruzione MAIN. L'indirizzo di tale istruzione è codificato stesso nel campo next_addr del registro MIR e dunque non rappresenta un salto.
- (e) Vengono opportunamente impostati i segnali di controllo dei bus B e C e dell'ALU, in modo da lasciare inalterato il valore del registro MBR e posizionarlo sia nel registro TOS che MDR.

7. (265 ns)

- (a) Viene alzato il segnale wr_ff di richiesta di scrittura verso la memoria.
- (b) Viene aggiornato il valore dei registri TOS e MDR. Solo in questo momento l'operando 0x56 è pronto per essere scritto in memoria.

8. (275 ns) L'operazione di scrittura dell'operando 0x56 in memoria è completata. È bene osservare che i due invarianti su cui si basa il funzionamento del processore Mic-1 sono preservati al termine dell'esecuzione dell'istruzione BIPUSH. In particolare, si nota che lo stack pointer viene aggiornato all'istante 245 ns, mentre il registro TOS all'istante 265 ns.

Si riporta, infine, un'analisi temporale più approfondita del contenuto dei registri del processore Mic-1 durante l'esecuzione dell'istruzione.

Analisi temporale istruzione BIPUSH

```

205 ns:      MEM_CONTROL = 001 (FETCHA)
215 ns:      MEM_ISTR_IN = x10 (preleva BIPUSH)
              MPC_VIRTUAL = x006
              REG_TO_B_CONTROL = 010000000 (-> TOS IN)
              REG_TO_C_CONTROL = 000010000 (-> LV OUT)
              ALU_CONTROL = 00010100 (PASSA B INALTERATO)
              FETCH_FF = 1
              MIR_REG -> bit JMPC = 1 (MIR_REG presenta il bit JMPC alto)
              (MPC = NEXT_ADDR OR MBR = x010)
              MPC_VIRTUAL = x010 (prossima istruzione -> BIPUSH)
              MBR_REG = x10 (PRELEVATO PRIMO BYTE ISTRUZIONE)
              REG_TO_B_CONTROL = 000000010 (-> PC IN)
              REG_TO_C_CONTROL = 000000100 (-> PC OUT)
              ALU_CONTROL = 00110101 (B+1)
              MEM_CONTROL = 001 (FETCHA)
235 ns:      MEM_ISTR_IN = x50 (preleva OPERANDO)
              MPC_VIRTUAL = x011 (inizializza BIPUSH)
              REG_TO_B_CONTROL = 000010000 (-> SP IN)

```

```

REG_TO_C_CONTROL = 000001001 (-> SP e MAR OUT)
ALU_CONTROL = 00110101 (B+1)
FETCH_FF = 1
245 ns: MPC_VIRTUAL = x012
MBR_REG = x56 (PRELEVATO SECONDO BYTE ISTRUZIONE)
REG_TO_B_CONTROL = 000000010 (-> PC IN)
REG_TO_C_CONTROL = 000000100 (-> PC OUT)
ALU_CONTROL = 00110101 (B+1)
MEM_CONTROL = 001 (FETCHA)
MEM_ISTR_IN = x00 (termina BIPUSH)
MPC_VIRTUAL = x006 (GOTO MAIN)
REG_TO_B_CONTROL = 000000100 (-> SP IN)
REG_TO_C_CONTROL = 001000010 (-> TOS e MDR OUT)
ALU_CONTROL = 00010100 (PASSA B INALTERATO)
MEM_CONTROL = 100 (SCRIVI)
FETCH_FF = 1
255 ns: MIR_REG -> bit JMPC = 1 (MIR_REG presenta il bit JMPC alto)
(MPC = NEXT_ADDR OR bit a bit MBR = x00)
MPC_VIRTUAL = x000 (programma TERMINATO)
MBR_REG = x00
TOS = MDR = x56
W_FF = 1
265 ns: MEM_DATA_IN = 0x00000056
275 ns:

```

IADD

Per analizzare il funzionamento dell'istruzione IADD si è sviluppato il programma in linguaggio IJVM riportato di seguito:

Istruzione IADD in linguaggio IJVM

```

.main
BIPUSH 0x1
BIPUSH 0x2
IADD
.endmethod

```

Dal programma IJVM si evince che per effettuare la somma di due operandi è stato necessario inserire preventivamente nello stack i due operandi, tramite due istruzioni BIPUSH. Gli operandi caricati sono i valori 0x1 e 0x2.

Si osserva inoltre che l'istruzione IADD, a differenza della BIPUSH non necessita di alcun operando: è implicita l'esecuzione della somma fra gli ultimi due valori caricati nello stack. Questo perché l'architettura del processore Mic-1 prevede istruzioni di lunghezza variabile.

Di seguito è riportata la specifica in linguaggio MAL dell'istruzione IJVM IADD.

Specifiche dell'istruzione IADD in linguaggio MAL

```

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

```

A questo punto è possibile proseguire con l'analisi delle microistruzioni eseguite dal processore al fine di realizzare l'istruzione IADD. La simulazione dell'esecuzione è riportata in Figura 12.3.

Si noti che l'analisi dell'istruzione IADD è stata eseguita a partire da 285 ns in quanto, nel lasso temporale precedente, sono eseguite delle microistruzioni relative alle istruzioni IJVM BIPUSH e MAIN.

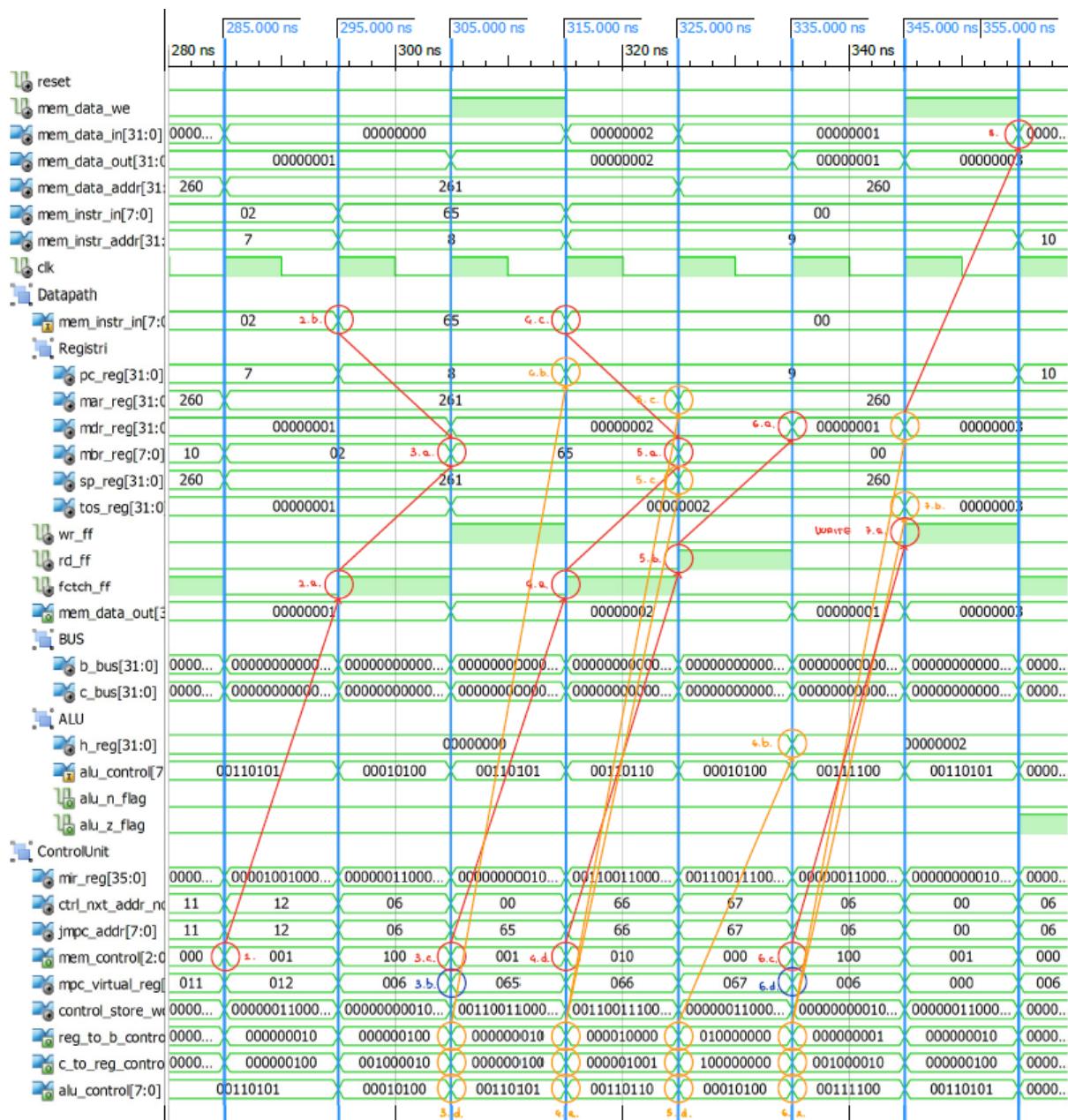


Figura 12.3: Simulazione esecuzione istruzione IADD

Dalla simulazione si evincono le seguenti operazioni principali, necessarie alla corretta esecuzione dell'istruzione:

1. (285 ns) BIPUSH

- Viene richiesto il fetch di un byte della prossima istruzione situata in RAM, impostando opportunamente i bit del registro **mem_control**, contenente i segnali di controllo della memoria RAM.
- Vengono inoltre eseguite ulteriori operazioni riguardanti l'istruzione BIPUSH.

2. (295 ns) BIPUSH:

- (a) Viene alzato il segnale `fetch_ff` di richiesta di fetch verso la memoria.
- (b) Il valore del byte a cui si riferisce il PC della RAM è 0x65, codice operativo dell'istruzione IADD.
- (c) Vengono inoltre eseguite le ultime operazioni riguardanti l'istruzione BIPUSH.

3. (305 ns) MAIN:

- (a) Viene prelevata il codice operativo dell'istruzione IADD e caricato nel registro MBR.
- (b) Viene aggiornato il valore del MPC, in modo da eseguire la microprocedura associata all'istruzione IADD: per ottenere il valore aggiornato del MPC viene eseguita l'operazione di OR bit a bit tra i registri NEXT_ADDR ed MBR. Tale operazione è indicata dal presenza del bit JMP_C pari ad 1 all'interno del registro MIR, registro contenente la microistruzione corrente. È bene osservare che l'indirizzo della prima microistruzione da eseguire è pari al codice operativo dell'istruzione stessa.
- (c) Viene richiesto il fetch del prossimo byte della word situata in RAM, impostando opportunamente i bit del registro mem_control.
- (d) Vengono opportunamente impostati i segnali di controllo dei bus B e C e dell'ALU, in modo da incrementare il PC.

4. (315 ns) Inizio IADD:

- (a) Viene alzato il segnale `fetch_ff` di richiesta di fetch verso la memoria.
- (b) Il valore del byte a cui si riferisce il PC della RAM è 0x00.
- (c) Viene aggiornato il valore del PC.
- (d) Viene incrementato di uno il valore del MPC, in modo da eseguire la microistruzione successiva.
- (e) Viene richiesta la lettura dalla memoria all'indirizzo contenuto nel registro MAR, impostando opportunamente i bit del registro mem_control.
- (f) Vengono opportunamente impostati i segnali di controllo dei bus B e C e dell'ALU, in modo da decrementare il valore dello SP e posizionare tale valore sia nel registro SP che nel registro MAR.

5. (325 ns)

- (a) Viene prelevato il byte 0x00 e caricato nel registro MBR
- (b) Viene alzato il segnale `rd_ff` di richiesta di lettura dalla memoria.
- (c) Viene aggiornato il valore dei registri MAR e SP. In questo modo viene specificato di leggere dalla memoria il byte contenuto alla locazione puntata dal nuovo SP, rappresentante l'operando B 0x1. Tuttavia, si osserva che il valore del registro TOS non viene variato, per cui conterrà ancora il valore dell'operando A 0x2.

- (d) Viene incrementato di uno il valore del MPC, in modo da eseguire la microistruzione successiva.
- (e) Vengono opportunamente impostati i segnali di controllo dei bus B e C e dell'ALU, in modo da caricare il contenuto del registro TOS in H, corrispondente al bus A dell'ALU.

6. (335 ns)

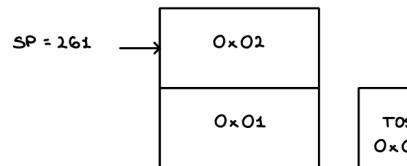
- (a) Viene aggiornato il registro MDR con il valore dell'operando B, richiesto dalla lettura in memoria. Si osserva che il dato richiesto è disponibile solo dopo due periodi di clock dal momento della richiesta.
- (b) Viene aggiornato il valore del registro H.
- (c) Viene richiesta la scrittura in memoria del contenuto del registro MDR, impostando opportunamente i bit del registro mem_control.
- (d) Viene aggiornato il valore del MPC, in modo da eseguire la microprocedura associata all'istruzione MAIN. L'indirizzo di tale istruzione è codificato stesso nel campo next_addr del registro MIR e dunque non rappresenta un salto.
- (e) Vengono opportunamente impostati i segnali di controllo dei bus B e C e dell'ALU, in modo da effettuare la somma tra i valori dei registri H e MDR. Il risultato della somma deve essere caricato sia nel registro TOS che MDR.

7. (345 ns)

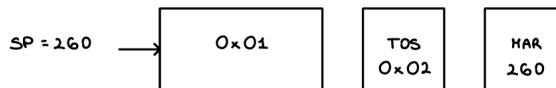
- (a) Viene alzato il segnale wr_ff di richiesta di scrittura verso la memoria.
- (b) Viene aggiornato il valore dei registri TOS e MDR. Solo in questo momento il risultato della somma 0x03 è pronto per essere scritto in memoria.

8. (355 ns) L'operazione di scrittura del risultato 0x03 in memoria è completata. È bene osservare che i due invarianti su cui si basa il funzionamento del processore *Mic-1* sono preservati al termine dell'esecuzione dell'istruzione IADD. In particolare, si nota che lo stack pointer viene aggiornato all'istante 325 ns, mentre il registro TOS all'istante 335 ns.

Di seguito, si riporta in Figura 12.4 l'evoluzione dello stato dello stack e di alcuni dei principali registri del processore coinvolti nell'istruzione di IADD.



(a) Stato dello stack prima dell'esecuzione dell'IADD.



(b) Viene decrementato lo SP e caricato il nuovo valore nel registro MAR.



(c) Viene caricato il valore di TOS in H e il secondo operando in MDR.



(d) Viene caricato il risultato della somma in TOS e MDR.



(e) Viene scritto il risultato nello stack.

Figura 12.4: Evoluzione dello stack e dei principali registri coinvolti nell'IADD.

Si riporta, infine, un'analisi temporale più approfondita del contenuto dei registri del processore *Mic-1* durante l'esecuzione dell'istruzione.

Analisi temporale istruzione IADD

```

285 ns:      MEM_CONTROL = 001 (FETCHA)
295 ns:      MEM_ISTR_IN = x65 (preleva IADD)
              MPC_VIRTUAL = x006
              REG_TO_B_CONTROL = 000000100 (-> MBR IN)
              REG_TO_C_CONTROL = 001000010 (-> TOS e MDR OUT)
              ALU_CONTROL = 00010100 (PASSA B INALTERATO)
              MEM_CONTROL = 100 (SCRIVI)
              FETCH_FF = 1
305 ns:      MIR_REG -> bit JMPC = 1 (MIR_REG presenta il bit JMPC alto)
              (MPC = NEXT_ADDR OR MBR = x065)
              MPC_VIRTUAL = x065 (prossima istruzione -> IADD)
              MBR_REG = x65 (PRELEVATO PRIMO BYTE ISTRUZIONE)
              REG_TO_B_CONTROL = 000000010 (-> PC IN)
              REG_TO_C_CONTROL = 000000100 (-> PC OUT)
              ALU_CONTROL = 00110101 (B+1)
              MEM_CONTROL = 001 (FETCHA)
              TOS = MDR = x02
              W_FF = 1
315 ns:      MEM_ISTR_IN = x00 (PRELEVVA SECONDO BYTE)
              MPC_VIRTUAL = x066 (inizializza IADD)
              REG_TO_B_CONTROL = 000010000 (-> SP IN)
              REG_TO_C_CONTROL = 000001001 (-> SP e MAR OUT)
              ALU_CONTROL = 00110110 (B-1)
              FETCH_FF = 1

```

```

325 ns:      MEM_CONTROL = 010 (LEGGI DATO ALL'IND. MAR) (-> VIENE DATO IL SEGNALE DI READ)
            MPC_VIRTUAL = x067
            MBR_REG = x00 (PRELEVATO SECONDO BYTE)
            REG_TO_B_CONTROL = 010000000 (-> TOS IN)
            REG_TO_C_CONTROL = 100000000 (-> H OUT)
            ALU_CONTROL = 00010100 (PASSA B INALTERATO)
            READ_FF = 1
335 ns:      MPC_VIRTUAL = x006 (GOTO MAIN)
            MDR_REG = x01 (OP B) (-> RICEVUTO IL DATO LETTO DOPO 2 Tck)
            H_REG = x02 (OP A)
            REG_TO_B_CONTROL = 000000001 (-> MDR IN)
            REG_TO_C_CONTROL = 001000010 (-> TOS e MDR OUT)
            ALU_CONTROL = 00111100 (A + B)
            MEM_CONTROL = 100 (SCRIVI)
            MIR_REG -> bit JMPC = 1 (MIR_REG presenta il bit JMPC alto)
            (MPC = NEXT_ADDR OR bit a bit MBR = x00)
            MPC_VIRTUAL = x000 (programma TERMINATO)
            MBR_REG = x00
            TOS = MDR = x03
            W_FF = 1
355 ns:      MEM_DATA_IN = 0x00000003

```

12.3.2 Esercizio B

Come richiesto dalla traccia si è proceduto con la modifica di un codice operativo. In particolare, si è scelto di modificare il codice operativo dell'istruzione IAND in modo da far eseguire l'istruzione IOR.

Per poter effettuare le modifiche necessarie si è sviluppato il programma in linguaggio IJVM riportato di seguito, il quale esegue l'istruzione IAND dopo aver caricato due operandi nello stack.

Istruzione IAND in linguaggio IJVM

```

.main
BIPUSH 0x1
BIPUSH 0x2
IAND
.endmethod

```

Di seguito è riportata la specifica in linguaggio MAL dell'istruzione IJVM IAND.

Specifiche dell'istruzione IAND in linguaggio MAL

```

iand = 0x7E:
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR AND H; wr; goto main

```

A questo punto è possibile proseguire con l'analisi delle microistruzioni eseguite dal processore al fine di realizzare l'istruzione IAND. La simulazione dell'esecuzione è riportata in Figura 12.5.

Si noti che l'analisi dell'istruzione IAND è stata eseguita a partire da 295 ns in quanto, nel lasso temporale precedente, sono eseguite delle microistruzioni relative alle istruzioni IJVM BIPUSH e MAIN.

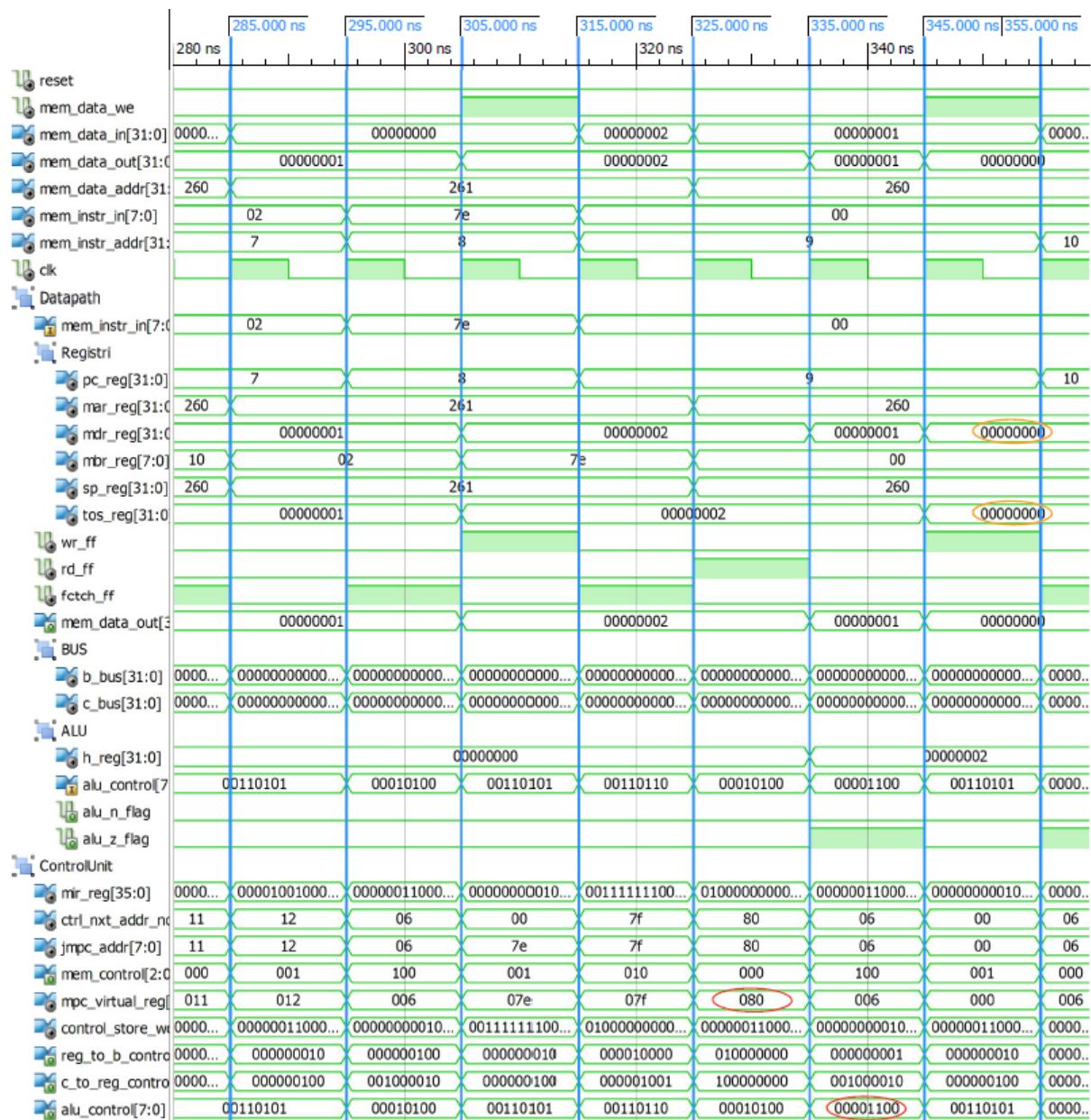


Figura 12.5: Simulazione esecuzione istruzione IAND

Dalla simulazione si osserva che le operazioni eseguite dal processore per realizzare l'istruzione IAND sono del tutto simili a quelle effettuate per realizzare l'istruzione IADD. La principale differenza fra le due micro-procedure sta nel fatto che mentre nel caso precedente i bit del registro ALU_CONTROL sono impostati per effettuare l'operazione di somma, in questo caso sono configurati per effettuare l'operazione di AND bit a bit.

Dalla simulazione si evince inoltre che il punto chiave dell'istruzione IAND risiede nell'impostazione dei bit del registro ALU_CONTROL nell'istante 335 ns. Tale configurazione di bit corrisponde infatti all'esecuzione da parte dell'ALU dell'operazione di AND bit a bit fra gli operandi in ingresso. Il risultato dell'elaborazione è visibile nei registri MDR e TOS all'istante 345 ns.

In Tabella 12.1 sono riportati i valori del registro ALU_CONTROL per le istruzioni IAND ed IOR.

F_0	F_1	EN_A	EN_B	INV_A	INC	Function
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B

Tabella 12.1: Tabella valori registro ALU_CONTROL per istruzioni IAND ed IOR

Per modificare il comportamento della IAND è stato quindi sufficiente individuare l'indirizzo della micro-ROM al quale sono specificate le operazioni effettuate dall'ALU. Per identificare tale indirizzo è stato osservato il valore del MPC un colpo di clock prima rispetto a quando vengono impostati i segnali di controllo dell'ALU per l'esecuzione della AND.

L'indirizzo di interesse è 0x80, corrispondente al valore 128 in decimale. Il valore contenuto nella micro-ROM a tale indirizzo è:

000000110000'00001100'0010000101000000

dove i bit delimitati dagli apici rappresentano i bit di controllo dell'ALU, contenuti in ALU_CONTROL.

Seguendo quanto riportato in Tabella 12.1, per modificare l'istruzione di IAND in modo da farle eseguire la OR bit a bit è stato quindi sufficiente sostituire al valore precedente la stringa di bit:

000000110000'00011100'0010000101000000

Di seguito è riportato un estratto di codice VHDL del componente *control_store*, il quale rappresenta la micro-ROM del processore. Nel codice viene evidenziato il punto in cui sono state effettuate le modifiche esposte.

Estratto di Codice Control Store

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.common_defs.all;

--! Processor control store

--! The control store is a ROM used to store the processor microprogram.
entity control_store is
    port (
        --! Address of the desired word
        address : in ctrl_str_addr_type;
        --! Content of the addressed word
        word    : out ctrl_str_word_type
    );
end entity control_store;

--! Dataflow architecture for the control store
architecture dataflow of control_store is

    -- Constants
    constant words : ctrl_str_type := (
        --BEGIN_WORDS_ENTRY
        0 => "000000110000000000000000000000001001",
        :
        :
    );

```

Apportate tali modifiche al contenuto della micro-ROM, si è osservato il comportamento del processore nel caso di esecuzione dell'istruzione `IAND`. La simulazione dell'esecuzione è riportata in Figura 12.6.

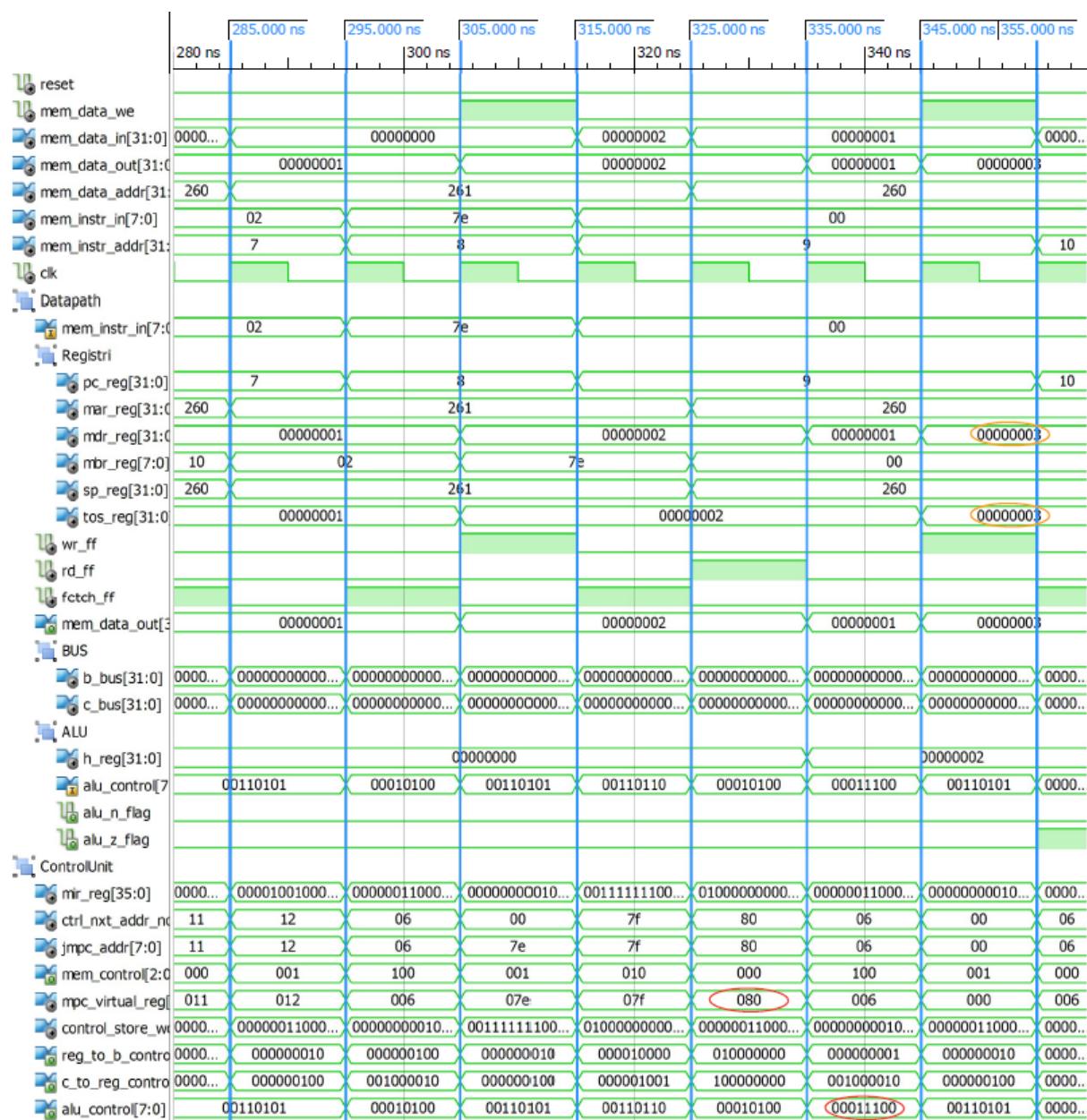


Figura 12.6: Simulazione esecuzione istruzione IOR

È facile osservare come il valore del registro ALU_CONTROL sia stato modificato e come il risultato dell'istruzione di IAND corrisponda effettivamente al risultato dell'IOR. Il risultato dell'elaborazione è visibile nei registri MDR e TOS all'istante 345 ns.

Si riporta, infine, un'analisi temporale più approfondita del contenuto dei registri del processore Mic-1 durante l'esecuzione dell'istruzione IAND.

Analisi temporale istruzione IAND

```
295 ns:    MEM_ISTR_IN = x7E (preleva IAND)
            MPC_VIRTUAL = x006
            REG_TO_B_CONTROL = 000000100 (-> MBR_IN)
            REG_TO_C_CONTROL = 001000010 (-> TOS e MDR_OUT)
```

```

ALU_CONTROL = 00010100 (PASSA B INALTERATO)
MEM_CONTROL = 100 (SCRIVI)
FETCH_FF = 1
305 ns: MIR_REG -> bit JMPC = 1 (MIR_REG presenta il bit JMPC alto)
(MPC = NEXT_ADDR OR MBR = x07E)
MPC_VIRTUAL = x07E (prossima istruzione -> IADD)
MBR_REG = x7E (PRELEVATO PRIMO BYTE ISTRUZIONE)
REG_TO_B_CONTROL = 000000010 (-> PC IN)
REG_TO_C_CONTROL = 000000100 (-> PC OUT)
ALU_CONTROL = 00110101 (B+1)
MEM_CONTROL = 001 (FETCHA)
TOS = MDR = x02
W_FF = 1
315 ns: MEM_ISTR_IN = x00 (PRELEVA SECONDO BYTE)
MPC_VIRTUAL = x07F (inizializza IADD)
REG_TO_B_CONTROL = 000010000 (-> SP IN)
REG_TO_C_CONTROL = 000001001 (-> SP e MAR OUT)
ALU_CONTROL = 00110110 (B-1)
FETCH_FF = 1
325 ns: MEM_CONTROL = 010 (LEGGI DATO ALL'IND. MAR) (-> VIENE DATO IL SEGNALE DI READ)
MPC_VIRTUAL = x080
MBR_REG = x00 (PRELEVATO SECONDO BYTE)
REG_TO_B_CONTROL = 010000000 (-> TOS IN)
REG_TO_C_CONTROL = 100000000 (-> H OUT)
ALU_CONTROL = 00010100 (PASSA B INALTERATO)
READ_FF = 1
335 ns: MPC_VIRTUAL = x006 (GOTO MAIN)
MDR_REG = x01 (OP B) (-> RICEVUTO IL DATO LETTO DOPO 2 Tck)
H_REG = x02 (OP A)
REG_TO_B_CONTROL = 000000001 (-> MDR IN)
REG_TO_C_CONTROL = 001000010 (-> TOS e MDR OUT)
ALU_CONTROL = 00111100 (A AND B) (-> OR: 00011100)
MEM_CONTROL = 100 (SCRIVI)
345 ns: MIR_REG -> bit JMPC = 1 (MIR_REG presenta il bit JMPC alto)
(MPC = NEXT_ADDR OR bit a bit MBR = x00)
MPC_VIRTUAL = x000 (programma TERMINATO)
MBR_REG = x00
TOS = MDR = x00
W_FF = 1

```

12.3.3 Esercizio C

Come richiesto dalla traccia si è proceduto con la descrizione del funzionamento del processore in merito ad un'istruzione di input/output. In particolare è stata scelta di analizzare l'istruzione GSTORE.

A tal fine sono state apportate alcune modifiche all'architettura complessiva del sistema, come riportato in Figura 12.7. Nello specifico sono stati considerati:

- Un processore IJVM.
- Una periferica PWM (Pulse Width Modulation), la quale è dotata di tre registri di controllo, tramite i quali è possibile impostarne i parametri di funzionamento.
- Un componente multiplexer, il quale si fa carico di indirizzare correttamente le richieste di scrittura in memoria da parte del processore.

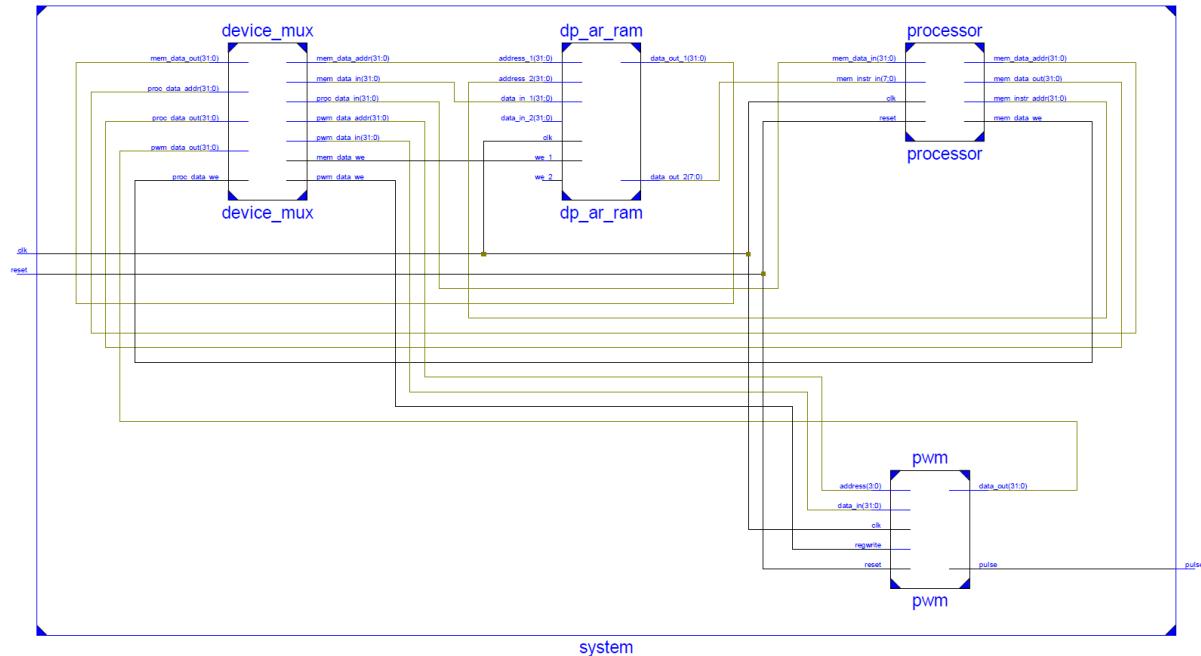


Figura 12.7: Schematico Processore e Periferica PWM

File: device_mux.vhd

Il ruolo chiave nella gestione della comunicazione con i dispositivi di I/O è svolto dal componente `device_mux`. Tale componente riceve in ingresso dal processore il segnale di abilitazione alla scrittura, ed il contenuto dei registri MAR e MDR. I segnali di ingresso sono quindi indirizzati in uscita verso la memoria RAM oppure verso i registri della periferica PWM.

È bene osservare che il processore non riconosce la scrittura su un dispositivo di I/O, ma si limita ad effettuare una normale operazione di scrittura in memoria. È infatti compito del multiplexer indirizzare correttamente la richiesta valutando il decimo bit meno significativo dell'indirizzo a cui scrivere. In particolare, se tale bit è pari ad uno, viene effettuata la scrittura su uno dei registri della periferica PWM, altrimenti viene effettuata una normale scrittura in memoria RAM.

Codice Multiplexer

```
-- Device multiplexer for amic-0 system
-- Alberto Moriconi

library ieee;
use ieee.std_logic_1164.all;

use work.common_defs.all;

entity device_mux is
  port (
    proc_data_we : in std_logic;
    proc_data_out : in reg_data_type;
    proc_data_addr : in reg_data_type;
    mem_data_out : in reg_data_type;
    pwm_data_out : in reg_data_type;
    proc_data_in : out reg_data_type;
    mem_data_we : out std_logic;
```

```

mem_data_in    : out reg_data_type;
mem_data_addr  : out reg_data_type;
pwm_data_we    : out std_logic;
pwm_data_in    : out reg_data_type;
pwm_data_addr  : out reg_data_type
);
end entity device_mux;

architecture dataflow of device_mux is

begin

-- Device multiplexing
with proc_data_addr(9) select mem_data_we <=
  proc_data_we when '0',
  '0'           when others;

with proc_data_addr(9) select mem_data_in <=
  proc_data_out when '0',
  (others => '0') when others;

with proc_data_addr(9) select mem_data_addr <=
  proc_data_addr when '0',
  (others => '0') when others;

with proc_data_addr(9) select pwm_data_we <=
  proc_data_we when '1',
  '0'           when others;

with proc_data_addr(9) select pwm_data_in <=
  proc_data_out when '1',
  (others => '0') when others;

with proc_data_addr(9) select pwm_data_addr <=
  proc_data_addr when '1',
  (others => '0') when others;

with proc_data_addr(9) select proc_data_in <=
  mem_data_out when '0',
  pwm_data_out when others;

end architecture dataflow;

```

Per analizzare il funzionamento dell'istruzione GSTORE si è sviluppato il programma in linguaggio IJVM riportato di seguito:

Istruzioni di I/O in linguaggio IJVM

```

.constant
d_reg 0x200
f_reg 0x204
c_reg 0x208
d 0x2FAF080
f 0x5F5E100
c 0x1
.endconstant

.main
LDC_W d
GSTORE d_reg
LDC_W f
GSTORE f_reg
LDC_W c
GSTORE c_reg
HALT
.endmethod

```

Dal programma IJVM si evince che per compiere operazioni di I/O sono stati definiti preventivamente alcuni valori nell'area di memoria riservata alle costanti. Prima di effettuare un'operazione di scrittura su uno dei registri della periferica PWM è necessario caricare in testa allo stack il valore da scrivere. Nel caso specifico, tale valore viene prelevato dall'area delle costanti e caricato in memoria attraverso l'istruzione LDC_W.

Di seguito è riportata la specifica in linguaggio MAL delle istruzioni IJVM LDC_W e GSTORE.

Specifiche dell'istruzione LDC_W in linguaggio MAL

```
ldc_w = 0x20:  
    PC = PC + 1; fetch  
    H = MBRU << 8  
    H = MBRU OR H  
    MAR = H + CPP; rd; goto iload_cont  
  
iload_cont:  
    MAR = SP = SP + 1  
    PC = PC + 1; fetch; wr  
    TOS = MDR; goto main
```

Specifiche dell'istruzione GSTORE in linguaggio MAL

```
gstore = 0xD0:  
    PC = PC + 1; fetch  
    H = MBRU << 8  
    H = MBRU OR H  
    MAR = H + CPP; rd  
    empty  
    MAR = MDR; goto istore_cont  
  
istore_cont:  
    MDR = TOS; wr  
    SP = MAR = SP - 1; rd  
    PC = PC + 1; fetch  
    TOS = MDR; goto main
```

A questo punto è possibile proseguire con l'analisi delle microistruzioni eseguite dal processore al fine di realizzare l'istruzione GSTORE. La simulazione dell'esecuzione è riportata in Figura 12.8.

Si noti che l'analisi è stata eseguita a partire da (415 ns) in quanto, nel lasso temporale precedente, sono eseguite delle microistruzioni relative alle istruzioni IJVM INVOKEVIRTUAL e MAIN.

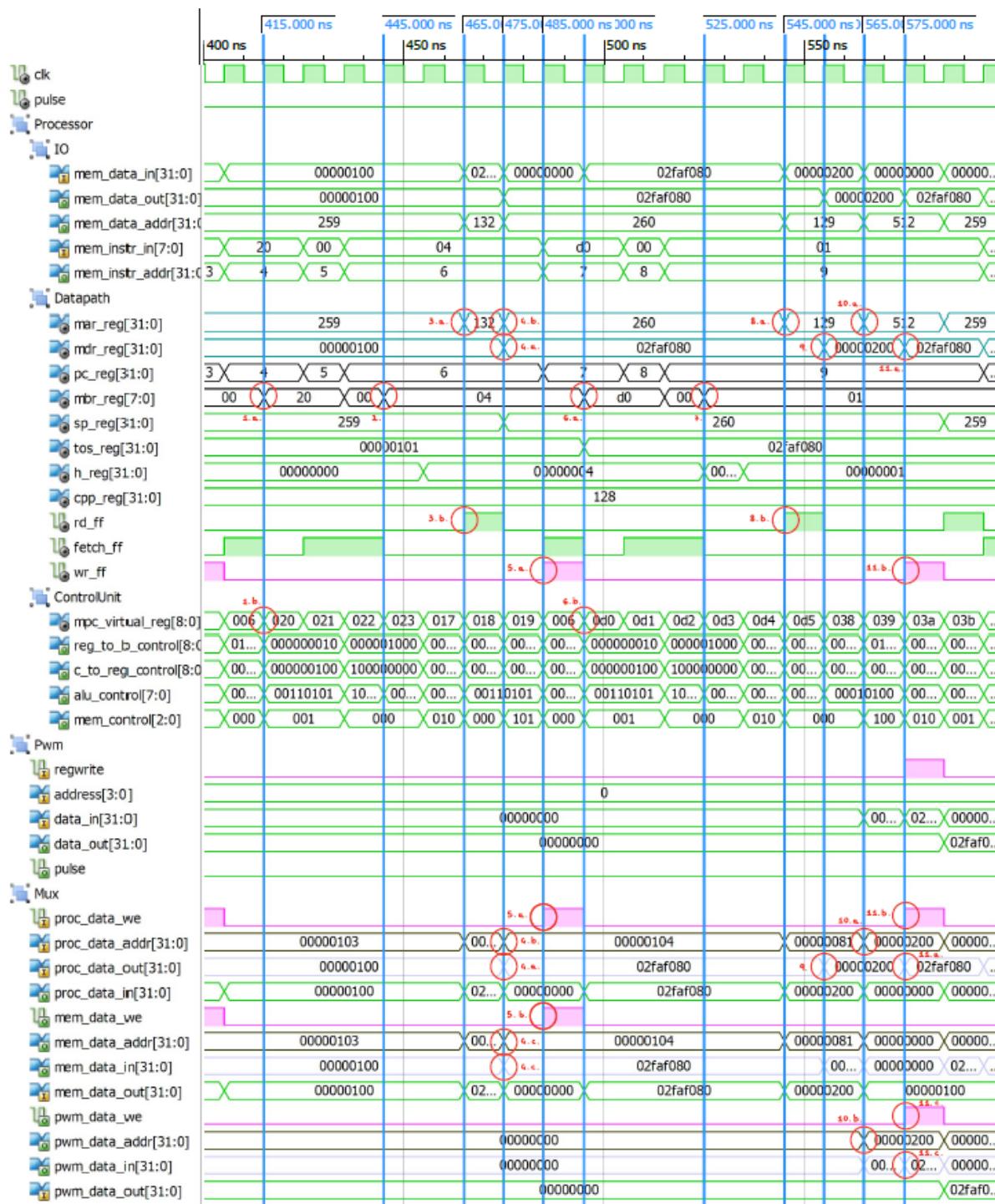


Figura 12.8: Simulazione esecuzione istruzione GSTORE

Dalla simulazione si evincono le seguenti operazioni principali, necessarie alla corretta esecuzione dell'istruzione:

1. (415 ns) LDC_W:

- (a) Viene prelevato il codice operativo dell'istruzione LDC_W (0x20) e caricato nel registro MBR.
 - (b) Viene aggiornato il valore del MPC, in modo da eseguire la microprocedura associata all'istruzione LDC_W.
2. (445 ns) LDC_W: viene prelevato dalla RAM il terzo byte dell'istruzione LDC_W, corrispondente allo spiazzamento relativo dell'indirizzo della costante da caricare nello stack. In altre parole, comando tale spiazzamento al valore del registro CPP, puntatore all'area di memoria delle costanti, si ottiene l'indirizzo di memoria della costante.
3. (465 ns) LDC_W:
- (a) Viene caricato nel registro MAR l'indirizzo della costante da prelevare ottenuto come specificato in precedenza.
 - (b) Viene alzato il segnale rd_ff di richiesta di lettura dalla memoria.
È bene osservare che il multiplexer non viene coinvolto in alcun modo nelle operazioni di lettura.
4. (475 ns) LDC_W:
- (a) Viene aggiornato il registro MDR con il valore della costante, richiesta dalla lettura in memoria.
 - (b) Viene aggiornato il valore del registro MAR, caricando in esso il contenuto dello SP, dopo averlo opportunamente incrementato. In questo modo viene specificato di scrivere in memoria alla locazione puntata dal nuovo SP.
 - (c) Sulla base dell'indirizzo specificato dal processore, i valori dei registri MDR e MAR vengono posto in ingresso alla memoria RAM.
L'indirizzo di memoria a cui è richiesta la scrittura è infatti 260 (0x104), per cui viene riconosciuta la scrittura in memoria RAM.
5. (485 ns) LDC_W:
- (a) Viene alzato il segnale wr_ff di richiesta di scrittura verso la memoria.
 - (b) Sulla base dell'indirizzo specificato dal processore, viene abilitata la scrittura in memoria RAM, piuttosto che sui registri della periferica PMW, alzando il segnale mem_data_we.
6. (495 ns) GSTORE:
- (a) Viene prelevato il codice operativo dell'istruzione GSTORE (0xd0) e caricato nel registro MBR.
 - (b) Viene aggiornato il valore del MPC, in modo da eseguire la microprocedura associata all'istruzione GSTORE.

7. (525 ns) GSTORE: viene prelevato dalla RAM il terzo byte dell'istruzione GSTORE, corrispondente allo spiazzamento relativo dell'indirizzo del registro della periferica PWM. In altre parole, comando tale spiazzamento al valore del registro CPP si ottiene l'indirizzo di memoria in cui è contenuto l'indirizzo del registro della periferica su cui scrivere il valore contenuto in testa allo stack.
8. (545 ns) GSTORE:
 - (a) Viene caricato nel registro MAR l'indirizzo dell'indirizzo del registro della periferica su cui si deve scrivere, ottenuto come specificato in precedenza.
 - (b) Viene alzato il segnale rd_ff di richiesta di lettura dalla memoria.
9. (555 ns) GSTORE: Viene aggiornato il registro MDR con il valore dell'indirizzo del registro della periferica, richiesto dalla lettura in memoria.
10. (565 ns) GSTORE:
 - (a) Viene aggiornato il valore del registro MAR, caricando in esso il contenuto dell'MDR. In questo modo viene specificato di scrivere nel registro di controllo della periferica.
 - (b) Sulla base dell'indirizzo specificato dal processore, il contenuto del registro MAR viene posto in ingresso alla periferica PWM.
L'indirizzo di memoria a cui è richiesta la scrittura è infatti 512 (0x200), per cui viene riconosciuta la scrittura su uno dei registri di controllo della periferica PWM.
11. (575 ns) GSTORE:
 - (a) Viene caricato nel registro MDR il valore in testa allo stack, ovvero la costante precedentemente carica con l'istruzione LDC_W.
 - (b) Viene alzato il segnale wr_ff di richiesta di scrittura verso la memoria.
 - (c) Sulla base dell'indirizzo specificato dal processore, il contenuto del registro MDR viene posto in ingresso alla periferica e viene abilitata la scrittura su uno dei registri della periferica PWM, piuttosto che in memoria RAM, alzando il segnale pwm_data_we.

Capitolo 13

Esercizio 13

13.1 Traccia

Si realizzi un progetto a scelta dello studente. Di seguito si forniscono, a scopo esemplificativo, alcuni possibili progetti presentati durante il corso, che possono essere sviluppati/modificati dagli studenti. Per ognuno dei progetti suggeriti, viene fornita una cartella contenente la documentazione a supporto.

- a. Progettare, implementare in VHDL e sintetizzare un componente per la realizzazione di primitive crittografiche hardware secondo le specifiche fornite nella cartella “CRITTOGRAFIA”. Con riferimento all’utilizzo di una primitiva crittografica di encryption su un flusso continuo di dati, si discutano le possibili modifiche all’architettura hardware proposta che consentirebbero di aumentare il throughput (es. pipelining). Testare il funzionamento del componente.
- b. Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network, descritto nelle dispense fornite nella cartella “SWITCH”. 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.
- c. Progettare ed implementare in VHDL un componente in grado di effettuare il prodotto scalare fra 2 vettori A e B di N numeri interi ciascuno, codificati su M bit in complementi a 2 (con N ed M a scelta dello studente).

- d. Progettare ed implementare la rete neurale come da specifica fornita nella cartella “RETE NEURALE”. Si discutano le possibili modifiche all’architettura hardware proposta che consentirebbero di aumentare il throughput della macchina (es. pipelining).

13.2 Introduzione

Tra gli esercizi proposti, si è scelto di realizzare una rete neurale. Dopo un profondo studio sui principi alla base delle reti neurali, si è passati alla progettazione dell’architettura del dispositivo da implementare.

Data la complessità dell’esercizio, è stato necessario decomporre l’architettura del dispositivo in parte operativa e parte di controllo. Dopo aver definito gli elementi costituenti l’unità operativa è stata realizzata l’unità di controllo.

È bene osservare che a causa di vincoli tecnologici legati alla board di sviluppo sono state riadattate alcune specifiche richieste nella traccia originale. In questo modo, è stato possibile sintetizzare il dispositivo.

13.3 Soluzione

Le reti neurali artificiali sono formalismi matematici ispirati alle reti neurali biologiche. Tali sistemi sono in grado di apprendere l’esecuzione di determinati task, dopo aver osservato preliminarmente alcuni esempi, senza essere programmati per l’esecuzione di uno specifico compito.

Una rete neurale è costituita da un insieme di unità, dette **neuroni**, la cui struttura è illustrata in Figura 13.1. Ciascun neurone riceve un certo numero di ingressi ed esegue la somma riportata nella seguente equazione:

$$h = \sum_i x_i \cdot w_i + b \quad (13.1)$$

dove x_i indica l’i-esimo ingresso, w_i rappresenta il **peso** del neurone associato all’ingresso i-esimo e b rappresenta il **bias** associato al neurone. Al risultato della somma viene poi applicata una particolare funzione, detta **funzione di attivazione**. Il risultato di tale operazione viene quindi fornito in uscita.

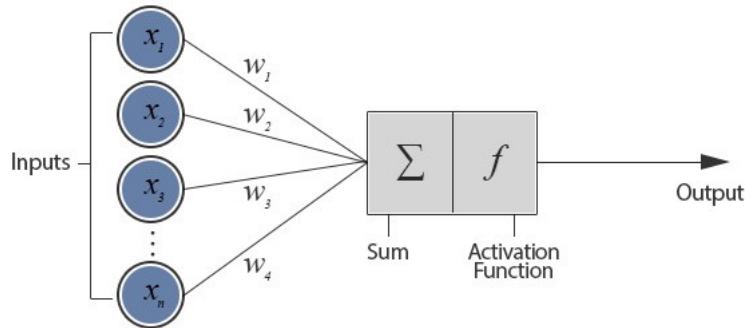


Figura 13.1: Neurone Artificiale

Per ulteriori dettagli si rimanda a [5].

È stata realizzata una rete neurale, la cui architettura è riportata in Figura 13.2. Nel dettaglio, si è supposto che la rete neurale sia già addestrata, pertanto il valore dei pesi è noto.

Per rendere il dispositivo sintetizzabile è stato ridotto il numero di neuroni su ogni layer e il numero di bit su cui sono rappresentati i valori di ingresso. In particolare, sono presenti:

- 2 neuroni nell'*Input Layer*
- 3 neuroni nell'*Hidden Layer*
- 1 neurone nell'*Output Layer*

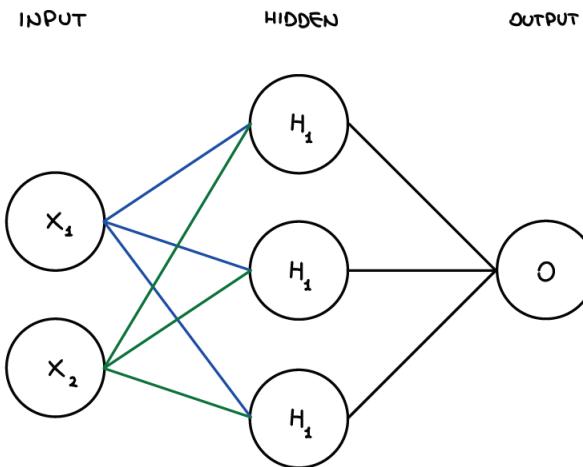


Figura 13.2: Architettura Rete Neurale

Data la complessità dell'esercizio, è stato necessario suddividere l'architettura del componente in due parti, come illustrato in Figura 13.3: unità operativa e unità di

controllo. Le due unità cooperano in modo da garantire il corretto funzionamento della rete neurale. In particolare, i segnali di uscita dell'unità di controllo pilotano l'unità operativa.

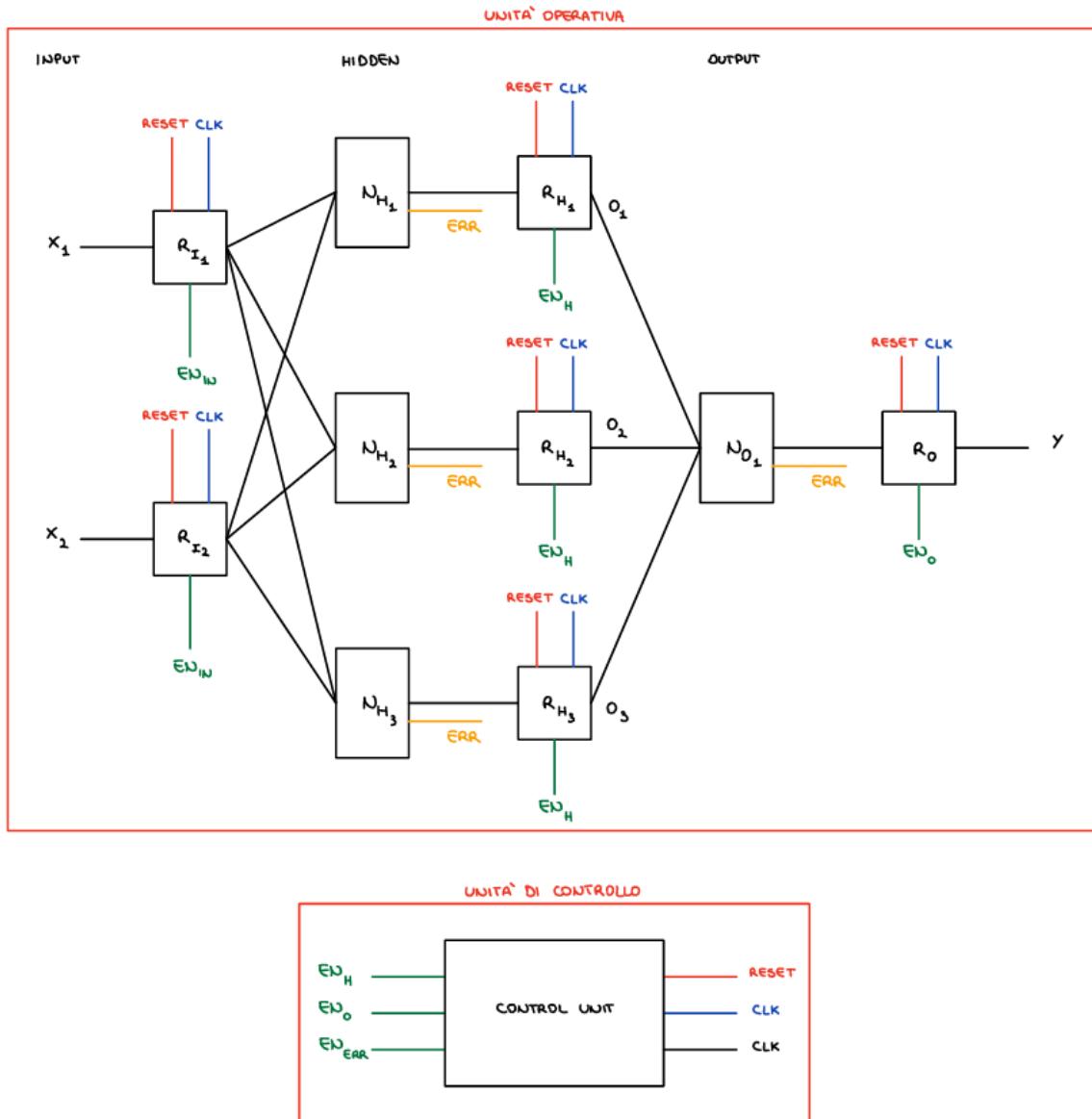


Figura 13.3: Architettura Rete Neurale

Per risolvere problemi di temporizzazione, si è scelto di abilitare l'unità di controllo sul fronte di salita del clock, mentre i componenti dell'unità operativa sul fronte di discesa. In questo modo, non solo l'unità operativa viene abilitata quando i segnali generati dall'unità di controllo sono ormai stabili, ma anche i valori dei registri sono aggiornati quando i loro segnali d'ingresso si sono stabilizzati.

Di seguito sono descritti e riportati rispettivamente i file relativi ad unità operativa e unità di controllo.

13.3.1 Unità Operativa

L'unità operativa, spesso indicata anche con il termine *datapath*, è costituita dall'insieme di componenti necessari per eseguire le operazioni desiderate.

Per la realizzazione della rete neurale, l'unità operativa prevede i seguenti componenti: sei registri parallelo-parallelo RegisterN di dimensioni opportune, tre neuroni per l'hidden layer NeuronHL e un neurone per l'output layer NeuronOL.

File: RegisterN.vhd

Il componente RegisterN (**behavioural**) rappresenta un registro parallelo-parallelo. Per la realizzazione della parte operativa sono stati utilizzati complessivamente sei registri di questo tipo. Nello specifico si hanno:

- Register_X_i: due registri di 4 bit, contenenti i valori di input della rete neurale prelevati dagli switch presenti sulla board. Tali valori sono espressi appunto su 4 bit, in complemento a due.
In particolare, Register_X₁ contiene i bit dei 4 switch più significativi, mentre Register_X₂ contiene i 4 bit meno significativi. Tali registri rappresentano i due neuroni dell'input layer.
- Register_H_i: tre registri di 8 bit, contenenti i valori di uscita dei tre neuroni dell'hidden layer.
- Register_Y: registro di 16 bit, contenente il valore di uscita del neurone dell'output layer.

Tali registri non solo consentono di bufferizzare le uscite di ogni livello della rete neurale al fine di tempificare opportunamente il circuito, ma rendono anche possibile la realizzazione di un'**architettura pipelined**. È infatti possibile caricare nuovi valori di ingresso nei registri dell'input layer e dare avvio alla valutazione dei risultati dell'hidden layer prima ancora che i risultati relativi ad elaborazioni precedenti vengano forniti in uscita.

Il codice relativo al componente RegisterN è uguale a quello riportato nel Paragrafo 10.3.1.

File: NeuronHL.vhd

Il componente NeuronHL (**structural**) rappresenta il generico neurone dell'hidden layer. Esso è in grado di ricevere in input i valori d'uscita dei neuroni dell'input layer e di fornire un valore di uscita in accordo con quanto specificato nella Figura 13.1. Tale componente fornisce inoltre in uscita il segnale ERR, il quale segnala una eventuale condizione di overflow verificatasi nella valutazione dell'uscita. In particolare, se almeno uno dei sommatori presenti nell'architettura segnala la condizione di overflow, il segnale ERR viene alzato.

Per la realizzazione del dispositivo è stata seguita un'architettura multi-livello, riportata in Figura 13.5. Essa è costituita da:

- **Multiplier:** due moltiplicatori su 4 bit, ognuno dei quali è in grado di effettuare il prodotto fra un valore di uscita dal layer precedente ed il corrispettivo peso. Il risultato della moltiplicazione è espresso su 8 bit.
- **Adder:** due sommatori su 8 bit, uno utilizzato per valutare la somma parziale tra i valori ottenuti in uscita dai moltiplicatori, l'altro per valutare la somma tra la somma parziale ed il valore di bias associato al neurone.
- **Multiplexer_2_1_Nbit.vh:** multiplexer 2:1 che realizza la funzione di attivazione. In particolare è stata scelta come funzione di attivazione la funzione **RELU**. Pertanto, il multiplexer riceve in ingresso un valore costante pari a zero ed il risultato dell'ultimo adder. Il segnale di selezione è rappresentato dal bit più significativo del valore in uscita dall'ultimo adder, rappresentante il segno del risultato: se tale bit è alto verrà selezionato il valore nullo, viceversa verrà selezionato il risultato dell'addizione.

Codice NeuronHL

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity NeuronHL is
  generic( NUM      : integer := 1;
           N        : integer := 4);
  port( X1       : in STD_LOGIC_VECTOR(N-1 downto 0);
        X2       : in STD_LOGIC_VECTOR(N-1 downto 0);
        Y        : out STD_LOGIC_VECTOR(2*N-1 downto 0);
        ERR     : out STD_LOGIC);
end NeuronHL;

architecture structural of NeuronHL is

  constant W11      : STD_LOGIC_VECTOR(N-1 downto 0) := "0001";
  constant W12      : STD_LOGIC_VECTOR(N-1 downto 0) := "0001";
  constant B1       : STD_LOGIC_VECTOR(2*N-1 downto 0) := "00000001";
  --constant B1      : STD_LOGIC_VECTOR(2*N-1 downto 0) := "01111111";--test overflow

  constant W21      : STD_LOGIC_VECTOR(N-1 downto 0) := "0001";
  constant W22      : STD_LOGIC_VECTOR(N-1 downto 0) := "0001";
  constant B2       : STD_LOGIC_VECTOR(2*N-1 downto 0) := "00000001";

  constant W31      : STD_LOGIC_VECTOR(N-1 downto 0) := "0001";
  constant W32      : STD_LOGIC_VECTOR(N-1 downto 0) := "0001";
  constant B3       : STD_LOGIC_VECTOR(2*N-1 downto 0) := "00000001";

  COMPONENT Multiplier
  GENERIC( N : integer );
  PORT(
    OP1 : IN std_logic_vector(N-1 downto 0);
    OP2 : IN std_logic_vector(N-1 downto 0);
    RIS : OUT std_logic_vector(2*N-1 downto 0)
  );
  END COMPONENT;

  COMPONENT Adder
  GENERIC( N : integer );
  PORT(
    OP1 : IN std_logic_vector(N-1 downto 0);
    OP2 : IN std_logic_vector(N-1 downto 0);
    RIS : OUT std_logic_vector(N-1 downto 0);
    ERR : OUT std_logic
  );
  END COMPONENT;

  COMPONENT Multiplexer_2_1_Nbit
  GENERIC( N : integer );
  PORT(
    X0 : IN std_logic_vector(N-1 downto 0);
    X1 : IN std_logic_vector(N-1 downto 0);
    S  : IN std_logic;
    Y  : OUT std_logic_vector(N-1 downto 0)
  );
  END COMPONENT;

```

```

signal W1 : STD_LOGIC_VECTOR(N-1 downto 0);
signal W2 : STD_LOGIC_VECTOR(N-1 downto 0);

signal B      : STD_LOGIC_VECTOR(2*N-1 downto 0);

signal P1 : STD_LOGIC_VECTOR(2*N-1 downto 0);
signal P2 : STD_LOGIC_VECTOR(2*N-1 downto 0);

signal S_TEMP : STD_LOGIC_VECTOR(2*N-1 downto 0);

signal ERR_1 : STD_LOGIC;
signal ERR_2 : STD_LOGIC;

signal Y_TEMP : STD_LOGIC_VECTOR(2*N-1 downto 0);

signal GROUND : STD_LOGIC_VECTOR(2*N-1 downto 0);

begin
    GROUND <= (others => '0');

    with NUM select
        W1 <= W11 when 1,
        W21 when 2,
        W31 when 3,
        (others => '0') when others;
    with NUM select
        W2 <= W12 when 1,
        W22 when 2,
        W32 when 3,
        (others => '0') when others;

    with NUM select
        B     <= B1 when 1,
        B2 when 2,
        B3 when 3,
        (others => '0') when others;

    Inst_Multiplier_1: Multiplier
        GENERIC MAP(
            N => N
        )
        PORT MAP(
            OP1 => X1,
            OP2 => W1,
            RIS => P1
        );
    Inst_Multiplier_2: Multiplier
        GENERIC MAP(
            N => N
        )
        PORT MAP(
            OP1 => X2,
            OP2 => W2,
            RIS => P2
        );
    Inst_Adder_1: Adder
        GENERIC MAP(
            N => 2*N
        )
        PORT MAP(
            OP1 => P1,
            OP2 => P2,
            RIS => S_TEMP,
            ERR => ERR_1
        );
    Inst_Adder_2: Adder
        GENERIC MAP(
            N => 2*N
        )
        PORT MAP(
            OP1 => S_TEMP,
            OP2 => B,
            RIS => Y_TEMP,
            ERR => ERR_2
        );
    Inst_Multiplexer_2_1_Nbit: Multiplexer_2_1_Nbit
        GENERIC MAP(
            N => 2*N
        )
        PORT MAP(

```

```

      X0 => Y_TEMP,
      X1 => GROUND,
      S => Y_TEMP(2*N-1),
      Y => Y
    );
    ERR <= ERR_1 or ERR_2;
end structural;

```

File: NeuronOL.vhd

Il componente NeuronOL (**structural**) rappresenta il neurone dell'output layer. Esso è in grado di ricevere in input i valori d'uscita dei neuroni dell'hidden layer e di fornire un valore di uscita in accordo con quanto specificato nella Figura 13.1. Tale componente fornisce inoltre in uscita il segnale ERR, il quale segnala una eventuale condizione di overflow verificatasi nella valutazione dell'uscita. In particolare, se almeno uno dei sommatori presenti nell'architettura segnala la condizione di overflow, il segnale ERR viene alzato.

Per la realizzazione del dispositivo è stata seguita un'architettura multi-livello, riportata in Figura 13.6. Essa è costituita da:

- **Multiplier**: tre moltiplicatori su 8 bit, ognuno dei quali è in grado di effettuare il prodotto fra un valore di uscita dal layer precedente ed il corrispettivo peso. Il risultato della moltiplicazione è espresso su 16 bit.
 - **Adder**: tre sommatori su 16 bit, due utilizzati per valutare la somma parziale tra i valori ottenuti in uscita dai moltiplicatori, l'altro per valutare la somma tra la somma parziale ed il valore di bias associato al neurone.
 - **Multiplexer_2_1_Nbit.vh**: multiplexer 2:1 che realizza la funzione di attivazione. In particolare è stata scelta come funzione di attivazione la funzione **RELU**. Pertanto, il multiplexer riceve in ingresso un valore costante pari a zero ed il risultato dell'ultimo adder. Il segnale di selezione è rappresentato dal bit più significativo del valore in uscita dall'ultimo adder, rappresentante il segno del risultato: se tale bit è alto verrà selezionato il valore nullo, viceversa verrà selezionato il risultato dell'addizione.

Codice NeuronOL

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity NeuronOL is
    generic( N : integer := 4 );
    port( H1 : in STD_LOGIC_VECTOR(2*N-1 downto 0);
          H2 : in STD_LOGIC_VECTOR(2*N-1 downto 0);
          H3 : in STD_LOGIC_VECTOR(2*N-1 downto 0);
          Y  : out STD_LOGIC_VECTOR(4*N-1 downto 0);
          ERR : out STD_LOGIC );
end NeuronOL;

architecture Behavioral of NeuronOL is

    constant W1 : STD_LOGIC_VECTOR(2*N-1 downto 0) := "000000001";
    constant W2 : STD_LOGIC_VECTOR(2*N-1 downto 0) := "000000001";
    constant W3 : STD_LOGIC_VECTOR(2*N-1 downto 0) := "000000000";
    constant B  : STD_LOGIC_VECTOR(4*N-1 downto 0) := "00000000000000000000000000000000";

    COMPONENT Multiplier
    GENERIC( N : integer );

```

```

PORT(
    OP1 : IN std_logic_vector(N-1 downto 0);
    OP2 : IN std_logic_vector(N-1 downto 0);
    RIS : OUT std_logic_vector(2*N-1 downto 0)
);
END COMPONENT;

COMPONENT Adder
GENERIC( N : integer);
PORT(
    OP1 : IN std_logic_vector(N-1 downto 0);
    OP2 : IN std_logic_vector(N-1 downto 0);
    RIS : OUT std_logic_vector(N-1 downto 0);
    ERR : OUT std_logic
);
END COMPONENT;

COMPONENT Multiplexer_2_1_Nbit
GENERIC( N : integer);
PORT(
    X0 : IN std_logic_vector(N-1 downto 0);
    X1 : IN std_logic_vector(N-1 downto 0);
    S : IN std_logic;
    Y : OUT std_logic_vector(N-1 downto 0)
);
END COMPONENT;

signal P1 : STD_LOGIC_VECTOR(4*N-1 downto 0);
signal P2 : STD_LOGIC_VECTOR(4*N-1 downto 0);
signal P3 : STD_LOGIC_VECTOR(4*N-1 downto 0);

signal S_TEMP1 : STD_LOGIC_VECTOR(4*N-1 downto 0);
signal S_TEMP2 : STD_LOGIC_VECTOR(4*N-1 downto 0);

signal ERR_1 : STD_LOGIC;
signal ERR_2 : STD_LOGIC;
signal ERR_3 : STD_LOGIC;

signal Y_TEMP : STD_LOGIC_VECTOR(4*N-1 downto 0);
signal GROUND : STD_LOGIC_VECTOR(4*N-1 downto 0);

begin
    GROUND <= (others => '0');

    Inst_Multiplier_1: Multiplier
        GENERIC MAP(
            N => 2*N
        )
        PORT MAP(
            OP1 => H1,
            OP2 => W1,
            RIS => P1
        );

    Inst_Multiplier_2: Multiplier
        GENERIC MAP(
            N => 2*N
        )
        PORT MAP(
            OP1 => H2,
            OP2 => W2,
            RIS => P2
        );

    Inst_Multiplier_3: Multiplier
        GENERIC MAP(
            N => 2*N
        )
        PORT MAP(
            OP1 => H3,
            OP2 => W3,
            RIS => P3
        );

    Inst_Adder_1: Adder
        GENERIC MAP(
            N => 4*N
        )
        PORT MAP(
            OP1 => P1,
            OP2 => P2,
            RIS => S_TEMP1,
            ERR => ERR_1
        );

```

```

Inst_Adder_2: Adder
  GENERIC MAP (
    N => 4*N
  )
  PORT MAP (
    OP1 => S_TEMP1,
    OP2 => P3,
    RIS => S_TEMP2,
    ERR => ERR_2
  );
  Inst_Adder_3: Adder
  GENERIC MAP (
    N => 4*N
  )
  PORT MAP (
    OP1 => S_TEMP2,
    OP2 => B,
    RIS => Y_TEMP,
    ERR => ERR_3
  );
  Inst_Multiplexer_2_1_Nbit: Multiplexer_2_1_Nbit
  GENERIC MAP (
    N => 4*N
  )
  PORT MAP (
    X0 => Y_TEMP,
    X1 => GROUND,
    S => Y_TEMP(4*N-1),
    Y => Y
  );
  ERR <= ERR_1 or ERR_2 or ERR_3;
end Behavioral;

```

File: Multiplier.vhd

Il componente Multiplier (**data-flow**) è un moltiplicatore su N bit, dimensionato opportunamente a seconda del livello del neurone in cui viene istanziato.

Tale dispositivo è stato realizzato utilizzando l'operatore aritmetico * contenuto nella libreria IEEE.STD_LOGIC_ARITH. In questo modo, il moltiplicatore è stato riconosciuto dallo strumento di sintesi ed associato al componente omologo presente sulla scheda. In aggiunta, tale scelta è stata dettata da esigenze pratiche, in quanto, avendo realizzato precedentemente un moltiplicatore MAC nel Capitolo 9, sarebbe stato necessario sviluppare un moltiplicatore in grado di operare con numeri *signed*.

Codice Multiplier

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.STD_LOGIC_UNSIGNED.all;

entity Multiplier is
  generic( N : integer := 8);
  port(
    OP1      : in   STD_LOGIC_VECTOR(N-1 downto 0);
    OP2      : in   STD_LOGIC_VECTOR(N-1 downto 0);
    RIS      : out  STD_LOGIC_VECTOR(2*N-1 downto 0));
end Multiplier;

architecture dataflow of Multiplier is

  signal RIS_TEMP : STD_LOGIC_VECTOR(2*N-1 downto 0);

begin
  RIS_TEMP <= signed(OP1) * signed(OP2);
  RIS <= RIS_TEMP;

```

```
end architecture;
```

File: Adder.vhd

Il componente Adder (**data-flow**) è un sommatore su N bit, dimensionato opportunamente a seconda del livello del neurone in cui viene istanziato.

Tale dispositivo è stato realizzato utilizzando l'operatore aritmetico + contenuto nella libreria IEEE.STD_LOGIC_ARITH. In questo modo, il sommatore è stato riconosciuto dallo strumento di sintesi ed associato al componente omologo presente sulla scheda. Tale scelta è stata dettata da considerazioni analoghe a quelle effettuate per il moltiplicatore.

Rispetto al moltiplicatore, viene fornito in uscita un ulteriore segnale di errore, che sta ad indicare la condizione di overflow. Tale condizione si verifica nel momento in cui, dati due addendi positivi, viene restituito un valore negativo, e viceversa.

Codice Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.STD_LOGIC_ARITH.all;

entity Adder is
    generic( N : integer := 8);
    port(
        OP1      : in STD_LOGIC_VECTOR(N-1 downto 0);
        OP2      : in STD_LOGIC_VECTOR(N-1 downto 0);
        RIS      : out STD_LOGIC_VECTOR(N-1 downto 0);
        ERR      : out STD_LOGIC);
end Adder;

architecture dataflow of Adder is

    signal RIS_TEMP : STD_LOGIC_VECTOR(N-1 downto 0);

begin
    RIS_TEMP <= signed(OP1) + signed(OP2);
    RIS <= RIS_TEMP;
    ERR <= '1' when ((OP1(N-1) = '0' and OP2(N-1) = '0' and RIS_TEMP(N-1) = '1') or
                      (OP1(N-1) = '1' and OP2(N-1) = '1' and RIS_TEMP(N-1) = '0')) 
              else '0';
end dataflow;
```

File: Multiplexer_2_1_Nbit.vhd

Il codice relativo al componente Multiplexer_2_1_Nbit è uguale a quello riportato nel Paragrafo 1.3.

File: Datapath.vhd

I componenti dell'unità operativa sono istanziati ed opportunamente connessi all'interno del modulo Datapath.

In aggiunta ai componenti appena descritti, è presente un'ulteriore flip-flop, descritto attraverso un costrutto process, attivo sul fronte di discesa del segnale di clock. Tale flip-flop si occupa di bufferizzare il segnale di errore da fornire in uscita al datapath dato dalla OR tra i segnali di errore in uscita da ciascun neurone.

Codice Datapath

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Datapath is
    generic( N : integer := 4);
    port(
        CLOCK      : IN std_logic;
        EN_IN      : IN std_logic;
        EN_HI      : IN std_logic;
        EN_OUT     : IN std_logic;
        EN_ERR     : IN std_logic;
        RESET      : IN std_logic;
        X1         : IN std_logic_vector(N-1 downto 0);
        X2         : IN std_logic_vector(N-1 downto 0);
        Y          : OUT std_logic_vector(4*N-1 downto 0);
        ERR        : OUT std_logic
    );
end Datapath;

architecture Structural of Datapath is

COMPONENT NeuronHL
    GENERIC( N : integer;
             NUM : integer
    );
    PORT(
        X1         : IN std_logic_vector(N-1 downto 0);
        X2         : IN std_logic_vector(N-1 downto 0);
        Y          : OUT std_logic_vector(2*N-1 downto 0);
        ERR        : OUT std_logic
    );
END COMPONENT;

COMPONENT NeuronOL
    GENERIC( N : integer);
    PORT(
        H1         : IN std_logic_vector(2*N-1 downto 0);
        H2         : IN std_logic_vector(2*N-1 downto 0);
        H3         : IN std_logic_vector(2*N-1 downto 0);
        Y          : OUT std_logic_vector(4*N-1 downto 0);
        ERR        : OUT std_logic
    );
END COMPONENT;

COMPONENT RegisterN
    GENERIC( N : integer);
    PORT(
        CLOCK : IN std_logic;
        RESET : IN std_logic;
        ENABLE : IN std_logic;
        X : IN std_logic_vector(N-1 downto 0);
        Y : OUT std_logic_vector(N-1 downto 0)
    );
END COMPONENT;

signal X1_REG : std_logic_vector(N-1 downto 0);
signal X2_REG : std_logic_vector(N-1 downto 0);

signal Y_H_REG_0 : std_logic_vector(2*N-1 downto 0);
signal Y_H_REG_1 : std_logic_vector(2*N-1 downto 0);
signal Y_H_REG_2 : std_logic_vector(2*N-1 downto 0);

signal Y_O_TEMP_0 : std_logic_vector(2*N-1 downto 0);
signal Y_O_TEMP_1 : std_logic_vector(2*N-1 downto 0);
signal Y_O_TEMP_2 : std_logic_vector(2*N-1 downto 0);

signal Y_O_REG : std_logic_vector(4*N-1 downto 0);
signal ERR_TEMP : std_logic_vector(3 downto 0);

signal ERR_REG : std_logic;

begin

--Istanziazione componenti
Register_X1: RegisterN
GENERIC MAP( N => N )
PORT MAP (
    CLOCK => CLOCK,
    RESET => RESET,
    ENABLE => EN_IN,
    X => X1,
    Y => X1_REG

```

```

) ;

Register_X2: RegisterN
GENERIC MAP( N => N )
PORT MAP(
    CLOCK => CLOCK,
    RESET => RESET,
    ENABLE => EN_IN,
    X => X2,
    Y => X2_REG
) ;

Inst_NeuronHL_0: NeuronHL
GENERIC MAP(
    N => N,
    NUM => 1
)
PORT MAP(
    X1 => X1_REG,
    X2 => X2_REG,
    Y => Y_H_REG_0,
    ERR => ERR_TEMP(0)
) ;

Inst_NeuronHL_1: NeuronHL
GENERIC MAP(
    N => N,
    NUM => 2
)
PORT MAP(
    X1 => X1_REG,
    X2 => X2_REG,
    Y => Y_H_REG_1,
    ERR => ERR_TEMP(1)
) ;

Inst_NeuronHL_2: NeuronHL
GENERIC MAP(
    N => N,
    NUM => 3
)
PORT MAP(
    X1 => X1_REG,
    X2 => X2_REG,
    Y => Y_H_REG_2,
    ERR => ERR_TEMP(2)
) ;

Register_H0: RegisterN
GENERIC MAP( N => 2*N )
PORT MAP(
    CLOCK => CLOCK,
    RESET => RESET,
    ENABLE => EN_HI,
    X => Y_H_REG_0,
    Y => Y_O_TEMP_0
) ;

Register_H1: RegisterN
GENERIC MAP( N => 2*N )
PORT MAP(
    CLOCK => CLOCK,
    RESET => RESET,
    ENABLE => EN_HI,
    X => Y_H_REG_1,
    Y => Y_O_TEMP_1
) ;

Register_H2: RegisterN
GENERIC MAP( N => 2*N )
PORT MAP(
    CLOCK => CLOCK,
    RESET => RESET,
    ENABLE => EN_HI,
    X => Y_H_REG_2,
    Y => Y_O_TEMP_2
) ;

Inst_NeuronOL: NeuronOL
GENERIC MAP(
    N => N
)
PORT MAP(
    H1 => Y_O_TEMP_0,
    H2 => Y_O_TEMP_1,

```

```

H3 => Y_O_TEMP_2,
Y => Y_O_REG,
ERR => ERR_TEMP (3)
);

Register_Y: RegisterN
GENERIC MAP ( N => 4*N )
PORT MAP (
    CLOCK => CLOCK,
    RESET => RESET,
    ENABLE => EN_OUT,
    X => Y_O_REG,
    Y => Y
);
begin
    if(RESET = '1') then
        ERR_REG <= '0';
    elsif(falling_edge(CLOCK)) then
        if(EN_ERR = '1') then
            ERR_REG <= ERR_TEMP (0) or ERR_TEMP (1) or ERR_TEMP (2) or ERR_TEMP (3);
        end if;
    end if;
    end process ERR_PROC;
    ERR <= ERR_REG;
end Structural;

```

13.3.2 Unità di Controllo

File: ControlUnit.vhd

L'unità di controllo è il componente responsabile della gestione dell'unità operativa.

Per la realizzazione della rete neurale è stata implementata la rete sequenziale il cui automa è riportato in Figura 13.4.

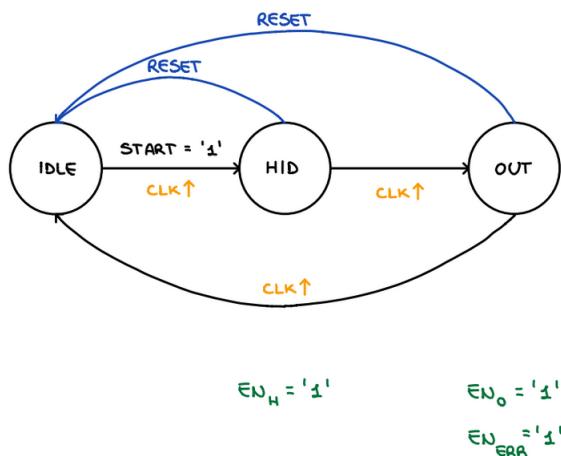


Figura 13.4: Macchina a Stati

L'automa evolve dunque attraverso i seguenti stati:

- IDLE_S: stato di inattività della macchina.
- HID_S: stato in cui vengono abilitati i registri a valle dei neuroni dell'hidden layer.

- OUT_S: stato in cui vengono abilitati i registri a valle dei neuroni dell'output layer.

È opportuno sottolineare che, se il segnale di RESET è asserito, l'automa si riporta nello stato IDLE_S qualunque sia lo stato in cui si trova.

È bene osservare inoltre che, non avendo specificato alcuna codifica per gli stati dell'automa, la codifica sarà effettuata automaticamente dallo strumento di sintesi.

Codice Control Unit

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ControlUnit is
    port(    CLOCK          : in STD_LOGIC;
              RESET          : in STD_LOGIC;
              START          : in STD_LOGIC;
              EN_HI          : out STD_LOGIC;
              EN_OUT         : out STD_LOGIC;
              EN_ERR         : out STD_LOGIC
            );
end ControlUnit;

architecture Behavioral of ControlUnit is

type STATE_TYPE is (
    IDLE_S,
    HID_S,
    OUT_S
);

signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;

begin
    CURR_STATE_PROC : process(CLOCK, RESET)
    begin
        if(RESET = '1') then
            CURRENT_STATE <= IDLE_S;
        elsif(rising_edge(CLOCK)) then
            if(START = '1' and CURRENT_STATE = IDLE_S) then
                CURRENT_STATE <= HID_S;
            else CURRENT_STATE <= NEXT_STATE;
            end if;
        end if;
    end process CURR_STATE_PROC;

    NEXT_STATE_PROC : process(CURRENT_STATE)
    begin
        EN_HI      <= '0';
        EN_OUT     <= '0';
        EN_ERR     <= '0';

        case CURRENT_STATE is
            when IDLE_S =>
                NEXT_STATE <= IDLE_S;

            when HID_S =>
                EN_HI <= '1';
                NEXT_STATE <= OUT_S;

            when OUT_S =>
                EN_OUT <= '1';
                EN_ERR <= '1';
                NEXT_STATE <= IDLE_S;
        end case;
    end process NEXT_STATE_PROC;
end Behavioral;

```

13.4 Schematici

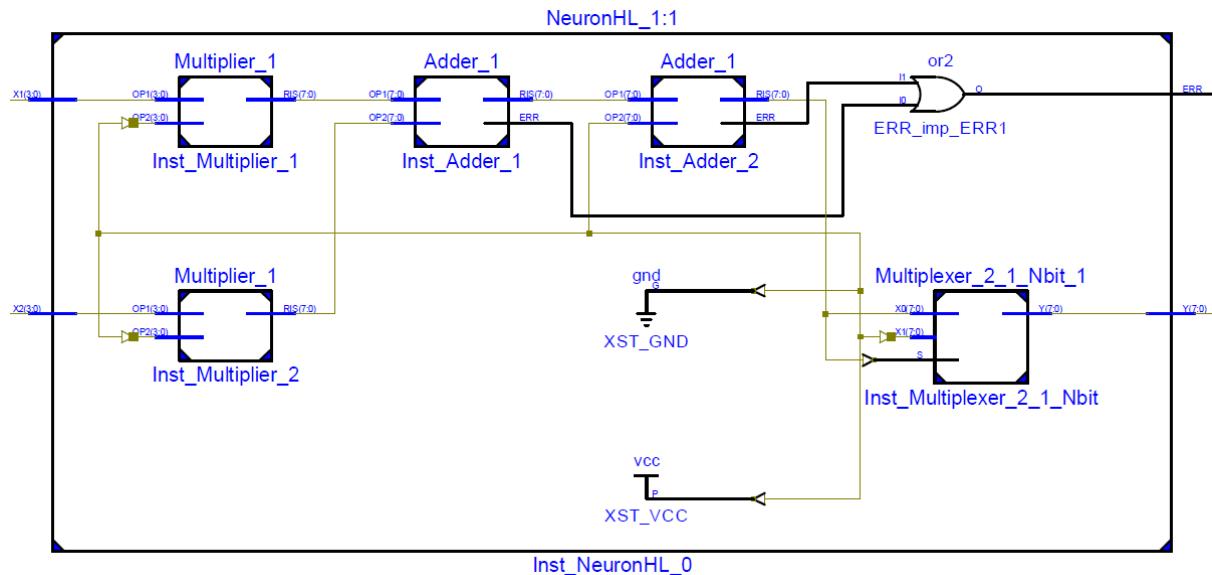


Figura 13.5: Schematico Neurone Hidden Layer

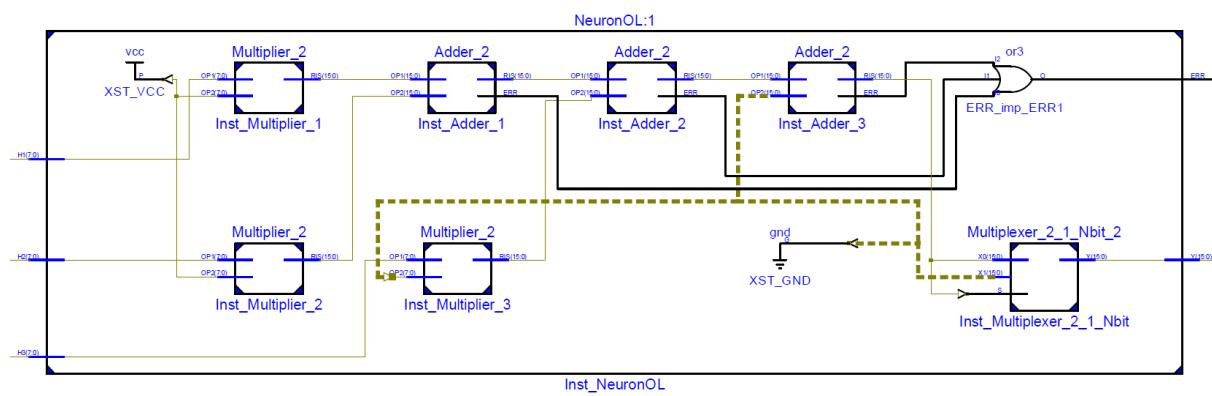


Figura 13.6: Schematico Neurone Output Layer

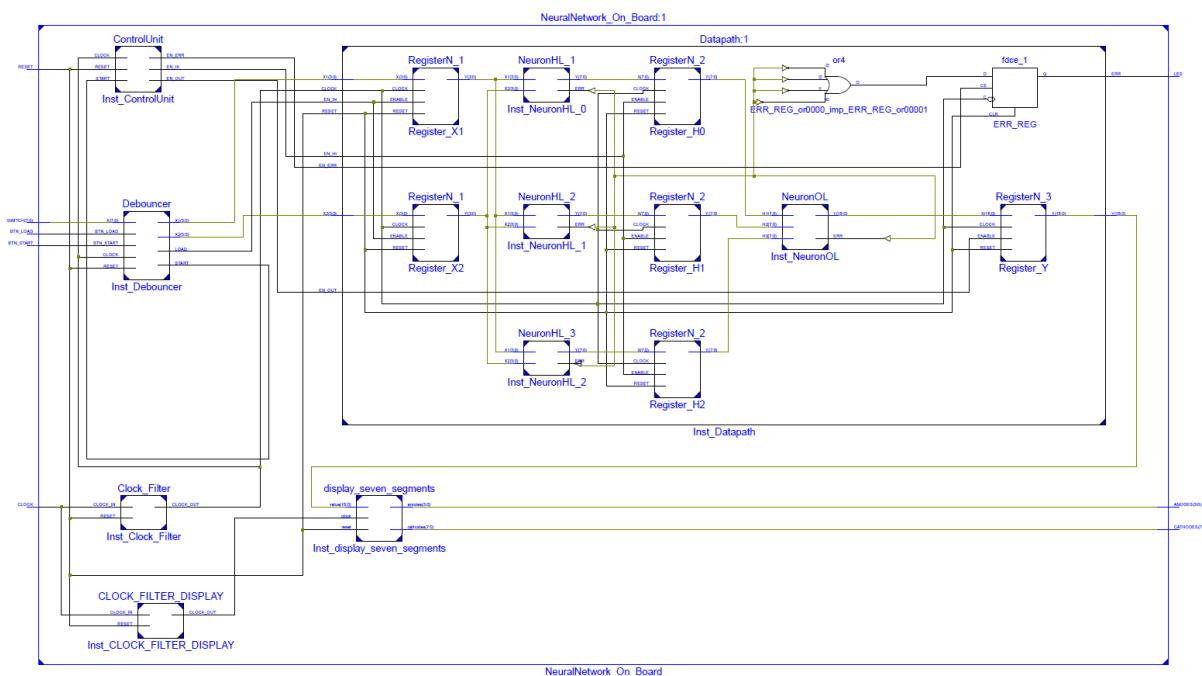


Figura 13.7: Schematico Neural Network On Board

13.5 Sintesi

Come richiesto dalla traccia, è stata effettuata la sintesi del dispositivo sulla board. A tal fine sono stati prodotti ulteriori moduli VHDL.

Il dispositivo sintetizzato sulla board presenta ancora una volta una struttura multi-livello, come riportato in Figura 13.7. Al livello più alto si trova il *top-level-module*, ovvero **NeuralNetwork_On_Board** (**structural**). Tale componente definisce i segnali di ingresso e uscita del dispositivo e descrive le connessioni tra i componenti appartenenti al livello inferiore:

- **Datapath (structural)**: sulla base di quanto esposto in precedenza, rappresenta l'unità operativa della rete neurale.
- **ControlUnit (behavioural)**: sulla base di quanto esposto in precedenza, rappresenta l'unità di controllo della rete neurale.
- **clock_filter (behavioural)**: componente responsabile della divisione in frequenza del segnale di clock, del tutto analogo a quello descritto nel Paragrafo 5.5. Tuttavia, a differenza del precedente, tale componente è stato realizzato in modo da generare in uscita un segnale avente un *duty cycle* del 50%.
- **Debouncer (behavioural)**: ricevuti in ingresso i segnali di clock, reset e un segnale di abilitazione alla lettura dagli switch, gestisce l'acquisizione degli ingressi della rete neurale.

Tale componente è analogo a quello descritto nel Paragrafo 5.5. Tuttavia, a differenza del precedente riceve un ulteriore segnale di ingresso che sta ad indicare l'inizio dell'elaborazione della rete.

- `clock_filter_Display (behavioural)`: componente responsabile della divisione in frequenza del segnale di clock, del tutto identico a quello descritto nel Paragrafo 5.5.
- `display_seven_segments (structural)`: riceve in ingresso un valore numerico rappresentato su 16 bit e restituisce in uscita le configurazioni di anodi e catodi tali da mostrare su quattro cifre del display la codifica in esadecimale il risultato del prodotto. Tale componente è a sua volta costituito da:
 - `cathodes_manager (behavioural)`: componente responsabile della gestione dei catodi.
 - `anodes_manager (behavioural)`: componente responsabile della gestione degli anodi.
 - `counter_mod4 (behavioural)`: contatore modulo 4.

File: NeuralNetwork_On_Board.vhd

Codice Neural Network On Board

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity NeuralNetwork_On_Board is
  GENERIC(  N : integer := 4);
  port(    CLOCK : in STD_LOGIC;
            RESET : in STD_LOGIC;
            SWITCH : in std_logic_vector(7 downto 0);
            BTN_LOAD : in STD_LOGIC;
            BTN_START : in STD_LOGIC;
            LED : out STD_LOGIC;
            ANODES : out std_logic_vector(3 downto 0);
            CATHODES : out std_logic_vector(7 downto 0));
end NeuralNetwork_On_Board;

architecture structural of NeuralNetwork_On_Board is
  COMPONENT ControlUnit
  PORT(
    CLOCK : IN std_logic;
    RESET : IN std_logic;
    START : IN std_logic;
    EN_HI : OUT std_logic;
    EN_OUT : OUT std_logic;
    EN_ERR : OUT std_logic
  );
  END COMPONENT;

  COMPONENT Datapath
  GENERIC(  N : integer);
  PORT(
    CLOCK : IN std_logic;
    EN_IN : IN std_logic;
    EN_HI : IN std_logic;
    EN_OUT : IN std_logic;
    EN_ERR : IN std_logic;
    RESET : IN std_logic;
    X1 : IN std_logic_vector(3 downto 0);
    X2 : IN std_logic_vector(3 downto 0);
    Y : OUT std_logic_vector(15 downto 0);
    ERR : OUT std_logic
  );
  END COMPONENT;

  COMPONENT Debouncer
  PORT(

```

```

CLOCK : IN std_logic;
RESET : IN std_logic;
X : IN std_logic_vector(7 downto 0);
BTN_LOAD : IN std_logic;
BTN_START : IN std_logic;
X1 : OUT std_logic_vector(3 downto 0);
X2 : OUT std_logic_vector(3 downto 0);
START : OUT std_logic;
LOAD : OUT std_logic
);
END COMPONENT;

COMPONENT Clock_Filter
generic(
    CLOCK_FREQUENCY_IN : integer;
    CLOCK_FREQUENCY_OUT : integer);
PORT(
    CLOCK_IN : IN std_logic;
    RESET : IN std_logic;
    CLOCK_OUT : OUT std_logic
);
END COMPONENT;

COMPONENT CLOCK_FILTER_DISPLAY
generic(
    CLOCK_FREQUENCY_IN : integer;
    CLOCK_FREQUENCY_OUT : integer);
PORT(
    CLOCK_IN : IN std_logic;
    RESET : IN std_logic;
    CLOCK_OUT : OUT std_logic
);
END COMPONENT;

COMPONENT display_seven_segments
PORT(
    clock : IN std_logic;
    reset : IN std_logic;
    value : IN std_logic_vector(15 downto 0);
    anodes : OUT std_logic_vector(3 downto 0);
    cathodes : OUT std_logic_vector(7 downto 0)
);
END COMPONENT;

signal CLOCK_FX : std_logic;
signal CLOCK_FX_DISPLAY : std_logic;
signal START_TEMP : std_logic;
signal RESET_N : std_logic;
signal EN_IN_TEMP : std_logic;
signal EN_HI_TEMP : std_logic;
signal EN_OUT_TEMP : std_logic;
signal EN_ERR_TEMP : std_logic;
signal X1_TEMP : std_logic_vector(N-1 downto 0);
signal X2_TEMP : std_logic_vector(N-1 downto 0);
signal Y_TEMP : std_logic_vector(4*N-1 downto 0);

begin
Inst_ControlUnit: ControlUnit PORT MAP (
    CLOCK => CLOCK_FX,
    RESET => RESET,
    START => START_TEMP,
    EN_HI => EN_HI_TEMP,
    EN_OUT => EN_OUT_TEMP,
    EN_ERR => EN_ERR_TEMP
);
Inst_Datapath: Datapath
GENERIC MAP(N => 4)
PORT MAP(
    CLOCK => CLOCK_FX,
    EN_IN => EN_IN_TEMP,
    EN_HI => EN_HI_TEMP,
    EN_OUT => EN_OUT_TEMP,
    EN_ERR => EN_ERR_TEMP,
    RESET => RESET,
    X1 => X1_TEMP,
    X2 => X2_TEMP,
    Y => Y_TEMP,
    ERR => LED
);
Inst_Debouncer: Debouncer PORT MAP (
    CLOCK => CLOCK_FX,
    RESET => RESET,
    X => SWITCH,
    BTN_LOAD => BTN_LOAD,
    BTN_START => BTN_START,

```

```

X1 => X1_TEMP,
X2 => X2_TEMP,
START => START_TEMP,
LOAD => EN_IN_TEMP
);

Inst_Clock_Filter: Clock_Filter
GENERIC MAP
    CLOCK_FREQUENCY_IN => 50000000,
    CLOCK_FREQUENCY_OUT => 500
)
PORT MAP
    CLOCK_IN => CLOCK,
    RESET => RESET,
    CLOCK_OUT => CLOCK_FX
);

Inst_CLOCK_FILTER_DISPLAY: CLOCK_FILTER_DISPLAY
GENERIC MAP
    CLOCK_FREQUENCY_IN => 50000000,
    CLOCK_FREQUENCY_OUT => 500
)
PORT MAP
    CLOCK_IN => CLOCK,
    RESET => RESET,
    CLOCK_OUT => CLOCK_FX_DISPLAY
);

Inst_display_seven_segments: display_seven_segments PORT MAP
    clock => CLOCK_FX_DISPLAY,
    reset => RESET,
    value => Y_TEMP,
    anodes => ANODES,
    cathodes => CATHODES
);
end structural;

```

File: clock_Filter.vhd

Il codice relativo al componente `clock_Filter` è uguale a quello riportato nel Paragrafo 10.5.

File: Debouncer.vhd

Codice Debouncer

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Debouncer is
    generic (N : integer := 4);
    port (CLOCK           : in STD_LOGIC;
          RESET            : in STD_LOGIC;
          X                : in STD_LOGIC_VECTOR(2*N-1 downto 0);
          BTN_LOAD         : in STD_LOGIC;
          BTN_START        : in STD_LOGIC;
          X1               : out STD_LOGIC_VECTOR(N-1 downto 0);
          X2               : out STD_LOGIC_VECTOR(N-1 downto 0);
          START            : out STD_LOGIC;
          LOAD             : out STD_LOGIC
        );
end Debouncer;

architecture Behavioral of Debouncer is

constant CONST : integer := 0;
begin
    DEB_PROC : process (CLOCK, RESET)
    variable i : integer := 0;
    begin
        if(RESET = '1') then
            X1 <= (others =>'0');

```

```
X2      <= (others => '0') ;
LOAD   <= '1';
START  <= '0';
elsif(rising_edge(CLOCK)) then
  if(BTN_LOAD = '1') then
    X1 <= X(2*N-1 downto N);
    X2 <= X(N-1 downto 0);
    LOAD <= '1';
    i := 1;
  elsif(BTN_START = '1' and i = 1) then
    START <= '1';
    LOAD <= '0';
    i := 0;
  else
    LOAD <= '0';
    START <= '0';
  end if;
end if;
end process DEB_PROC;

end Behavioral;
```

File: Clock_filter_display.vhd

Il codice relativo al componente clock_filter_display è uguale a quello riportato nel Paragrafo 5.5.

File: seven_segment_array.vhd

Il codice relativo al componente seven_segment_array è uguale a quello riportato nel Paragrafo 8.5.

File: counter_mod4.vhd

Il codice relativo al componente counter_mod4 è uguale a quello riportato nel Paragrafo 8.5.

File: cathodes_manager.vhd

Il codice relativo al componente cathodes_manager è uguale a quello riportato nel Paragrafo 8.5.

File: anodes_manager.vhd

Il codice relativo al componente anodes_manager è uguale a quello riportato nel Paragrafo 8.5.

13.6 Simulazione

File: NeuralNetwork_On_Board_tb.vhd

Codice Testbench NeuralNetwork_On_Board

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY NeuralNetwork_On_Board_tb IS
END NeuralNetwork_On_Board_tb;

ARCHITECTURE behavior OF NeuralNetwork_On_Board_tb IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT NeuralNetwork_On_Board
        PORT(
            CLOCK : IN std_logic;
            RESET : IN std_logic;
            SWITCH : IN std_logic_vector(7 downto 0);
            BTN_LOAD : IN std_logic;
            BTN_START : IN std_logic;
            LED : OUT std_logic;
            ANODES : OUT std_logic_vector(3 downto 0);
            CATHODES : OUT std_logic_vector(7 downto 0)
        );
    END COMPONENT;

    --Inputs
    signal CLOCK : std_logic := '0';
    signal RESET : std_logic := '0';
    signal SWITCH : std_logic_vector(7 downto 0) := (others => '0');
    signal BTN_LOAD : std_logic := '0';
    signal BTN_START : std_logic := '0';

    --Outputs
    signal LED : std_logic;
    signal ANODES : std_logic_vector(3 downto 0);
    signal CATHODES : std_logic_vector(7 downto 0);

    -- Clock period definitions
    constant CLOCK_period : time := 20 ns;

BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: NeuralNetwork_On_Board PORT MAP (
        CLOCK => CLOCK,
        RESET => RESET,
        SWITCH => SWITCH,
        BTN_LOAD => BTN_LOAD,
        BTN_START => BTN_START,
        LED => LED,
        ANODES => ANODES,
        CATHODES => CATHODES
    );

    -- Clock process definitions
    CLOCK_process :process
    begin
        CLOCK <= '0';
        wait for CLOCK_period/2;
        CLOCK <= '1';
        wait for CLOCK_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;

        wait for CLOCK_period*10;

        -- insert stimulus here
        SWITCH <= "00100010"; -- 2 2 --> 10
        BTN_LOAD <= '1';

        wait for 12 ms;
    end process;

```

```
BTN_LOAD    <= '0';
BTN_START   <= '1';

wait for 12 ms;
BTN_START   <= '0';
wait for 50 ms;
SWITCH <= "00100001"; -- 2 1 --> 8
BTN_LOAD    <= '1';

wait for 12 ms;
BTN_LOAD    <= '0';
BTN_START   <= '1';

wait for 12 ms;
BTN_START   <= '0';
wait for 50 ms;
SWITCH <= "00010001"; -- 1 1 --> 6
BTN_LOAD    <= '1';

wait for 12 ms;
BTN_LOAD    <= '0';
BTN_START   <= '1';

wait for 12 ms;
BTN_START   <= '0';
wait for 50 ms;
SWITCH <= "01110111"; -- 7 7 --> 30
BTN_LOAD    <= '1';

wait for 12 ms;
BTN_LOAD    <= '0';
BTN_START   <= '1';

wait for 12 ms;
BTN_START   <= '0';
wait;
end process;

END;
```

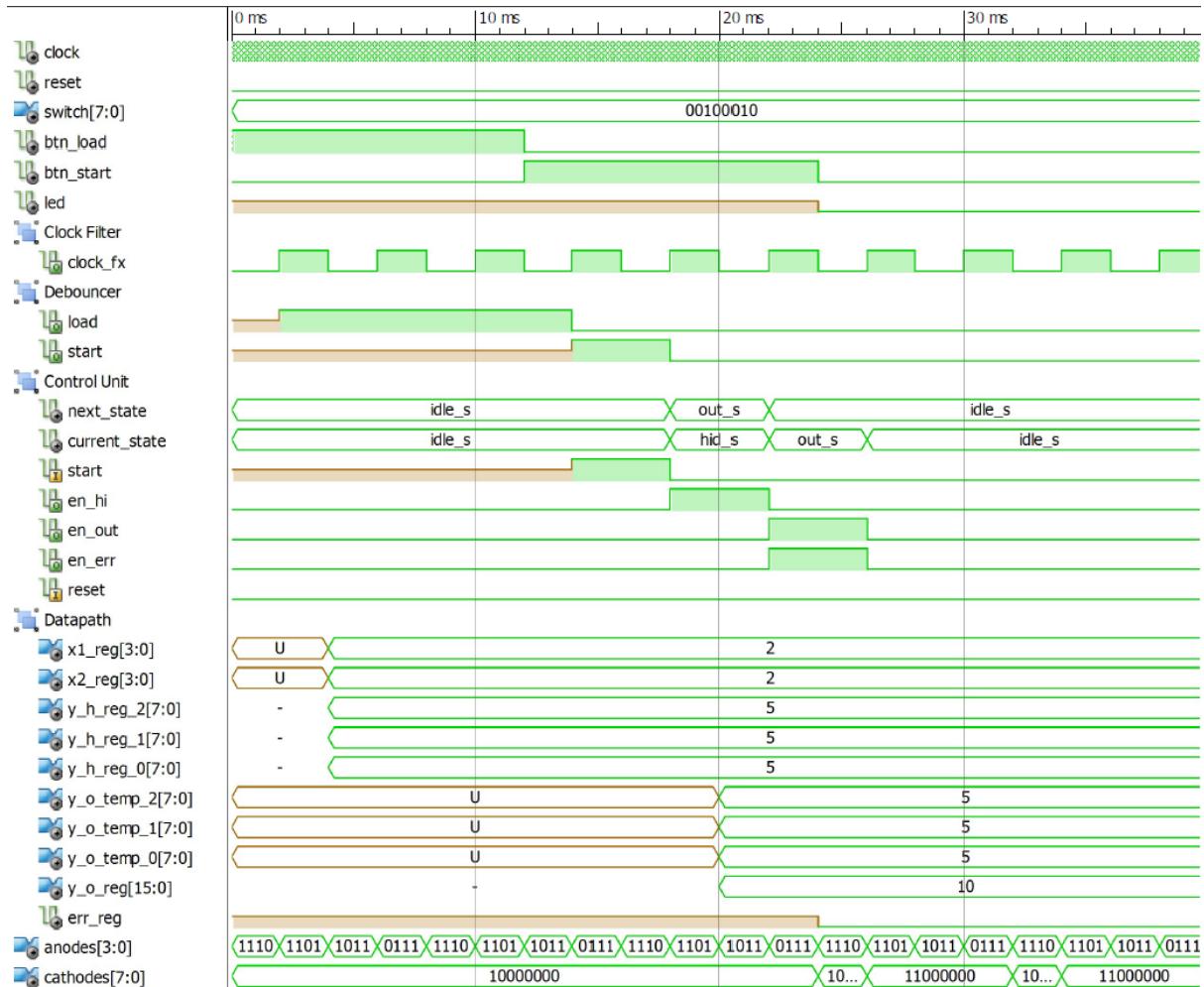


Figura 13.8: Simulazione Neural Network On Board

Bibliografia

- [1] Antonino Mazzeo. *Latch e Flip Flop - Corso di Architettura dei sistemi di elaborazione*. URL: <https://www.docenti.unina.it/webdocenti-be/allegati/materiale-didattico/34141967>.
- [2] *Algoritmo Double Dabble in Wikipedia*. URL: https://en.wikipedia.org/wiki/Double_dabble.
- [3] Giovanni Cozzolino Nicola Mazzocca. *Comunicazione tra dispositivi tramite interfaccia seriale*. URL: <https://www.docenti.unina.it/webdocenti-be/allegati/%20materiale-didattico/34149055>.
- [4] Alberto Moriconi Nicola Mazzocca. *Il processore Mic-1*. URL: <https://www.docenti.unina.it/webdocenti-be/allegati/materiale-didattico/34150737>.
- [5] Stefano Marrone Nicola Mazzocca. *Machine Learning on Embedded Systems*. URL: <https://www.docenti.unina.it/webdocenti-be/allegati/materiale-didattico/34149166>.