



UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

Corso di Architettura dei Sistemi Digitali – Prof. Mazzocca

Anno Accademico 2021-2022

PROGETTI DI ARCHITETTURA DEI SISTEMI DIGITALI



Antonio Romano M63001315
Giuseppe Riccio M63001314
Giuliano di Giuseppe M63001322
Sossio Cirillo M63001329

INDICE

Prefazione.....	6
1 Multiplexer	7
1.1 Progettazione del Multiplexer indirizzabile 16:1	7
1.1.1 Codice VHDL del Multiplexer	9
1.1.2 La simulazione del Multiplexer	12
1.2 Progettazione della Rete di Interconnessione a 16:4	14
1.2.1 Codice VHDL della Rete di Interconnessione 16:4	16
1.2.2 La simulazione della Rete	17
2 Encoder BCD.....	21
2.1 Progettazione del Encoder BCD	21
2.1.1 Codice VHDL del Encoder BCD	22
2.1.2 La simulazione del Encoder BCD	25
2.1.3 Implementazione sulla board tramite i led.....	26
2.1.4 Implementazione sulla board tramite il display	29
3 Riconoscitore a 2 <i>modalità</i>	33
3.1 Progettazione del Riconoscitore di sequenza a 2 modalità.....	33
3.1.1 Schematico del Riconoscitore di sequenza a 2 modalità.....	35
3.1.2 Codice VHDL del Riconoscitore di sequenza a 2 modalità	35
3.1.3 La simulazione del Riconoscitore di sequenza a 2 modalità	41
3.2 Implementazione sulla board	42
3.2.1 Il ButtonDebouncer	42
3.2.2 Top Module.....	44
3.2.3 Implementazione sulla board.....	45
4 Shift register.....	47
4.1 Shift register in generale	47
4.2 Shift register comportamentale	48
4.2.1 Codice VHDL	48
4.2.2 TestBench	49
4.2.3 La simulazione	51
4.3 Shift register strutturale.....	51
4.3.1 Codice VHDL	52
4.3.2 TestBench	58
4.3.3 La simulazione	61
5 Cronometro	62

5.1	Progettazione del Cronometro	62
5.1.1	Codice VHDL del Cronometro	63
5.1.2	La simulazione del Cronometro	67
5.2	Progettazione del Cronometro on Board	69
5.2.1	Codice VHDL del Cronometro on Board.....	70
5.3	Progettazione del Cronometro on Board con Intertempi.....	79
5.4	Progettazione del Cronometro on Board (Modifica dell'orologio contemporanea)	86
6	Sistema di testing	94
6.1	Progettazione del Sistema di Testing	94
6.1.1	Funzionamento	94
6.1.2	Codice VHDL.....	95
6.1.3	Test Bench	101
6.1.4	La simulazione del Sistema di Testing	102
6.2	Implementazione sulla board	102
6.2.1	Modifiche	102
6.2.2	Codice VHDL.....	102
6.2.3	Test sulla Board	106
7	Handshaking	108
7.1	Progettazione Handshaking tra due entità.....	108
7.1.1	Codice VHDL dell'Handshaking	110
7.1.2	La simulazione della Rete	136
7.1.3	Considerazioni finali e possibile scopo futuro	139
8	Processore MIC-1.....	140
8.1	Introduzione.....	140
8.1.1	Architettura Processore Mic-1.....	140
8.2	Soluzione	142
8.2.1	Esercizio A	142
8.2.2	Esercizio B	148
8.2.3	Modo alternativo Esercizio B.....	150
9	Interfaccia seriale.....	151
9.1	Progettazione interfaccia seriale	151
9.1.1	Schematico in RTL Analysis di Vivado	152
9.1.2	Codice VHDL	153
9.1.3	Test Bench	153
9.1.4	Simulazione.....	154
9.1.5	Constraints.....	155
9.1.6	Implementazione sulla board.....	156

9.2	Esercizio “2_UART_MEM”	157
9.2.1	Descrizione del funzionamento di “2_UART_MEM”	157
9.2.2	Codice VHDL	158
9.2.3	Test Bench	165
9.2.4	Simulazione.....	167
10	Switch multistadio	168
10.1	Traccia.....	168
10.1.1	Descrizione del funzionamento dello switch multistadio.....	168
10.1.2	Codice VHDL	169
10.1.3	Testbench.....	174
10.1.4	Simulazione.....	176
10.2	Switch con protocollo di handshaking.....	176
10.2.1	Descrizione del funzionamento dello switch multistadio con handshaking	176
10.2.2	Schematico in RTL Analysis di Vivado	177
10.2.3	Codice VHDL	177
10.2.4	Test Bench	180
10.2.5	Simulazione.....	182
11	Moltiplicatore di Robertson.....	183
11.1	Progettazione del Moltiplicatore di Robertson	183
11.1.1	Descrizione del funzionamento del Moltiplicatore di Robertson....	184
11.1.2	Schematico in RTL Analysis di Vivado	187
11.1.3	Codice VHDL del Moltiplicatore di Robertson.....	188
11.1.4	La simulazione del Moltiplicatore di Robertson.....	194
11.1.5	Implementazione sulla board tramite switch e display – versione esadecimale	197
11.1.6	Implementazione sulla board tramite switch e display – versione decimale	201
11.2	Applicazioni reali e timing analysis del Moltiplicatore di Robertson.....	206
12	Divisore Non-Restoring	208
12.1	Progettazione del Divisore Non-Restoring.....	208
12.1.1	Descrizione del funzionamento del Divisore Non-Restoring	212
12.1.2	Schematico in RTL Analysis di Vivado	214
12.1.3	Codice VHDL del Divisore Non-Restoring	215
12.1.4	La simulazione del Divisore Non-Restoring	225
12.2	Applicazioni reali e timing analysis del Divisore Non-Restoring	227
12.2.1	Descrizione del funzionamento della Calcolatrice.....	228
12.2.2	Schematico in RTL Analysis di Vivado	229

INDICE

12.2.3	Codice VHDL della Calcolatrice.....	230
12.2.4	La simulazione della Calcolatrice.....	233

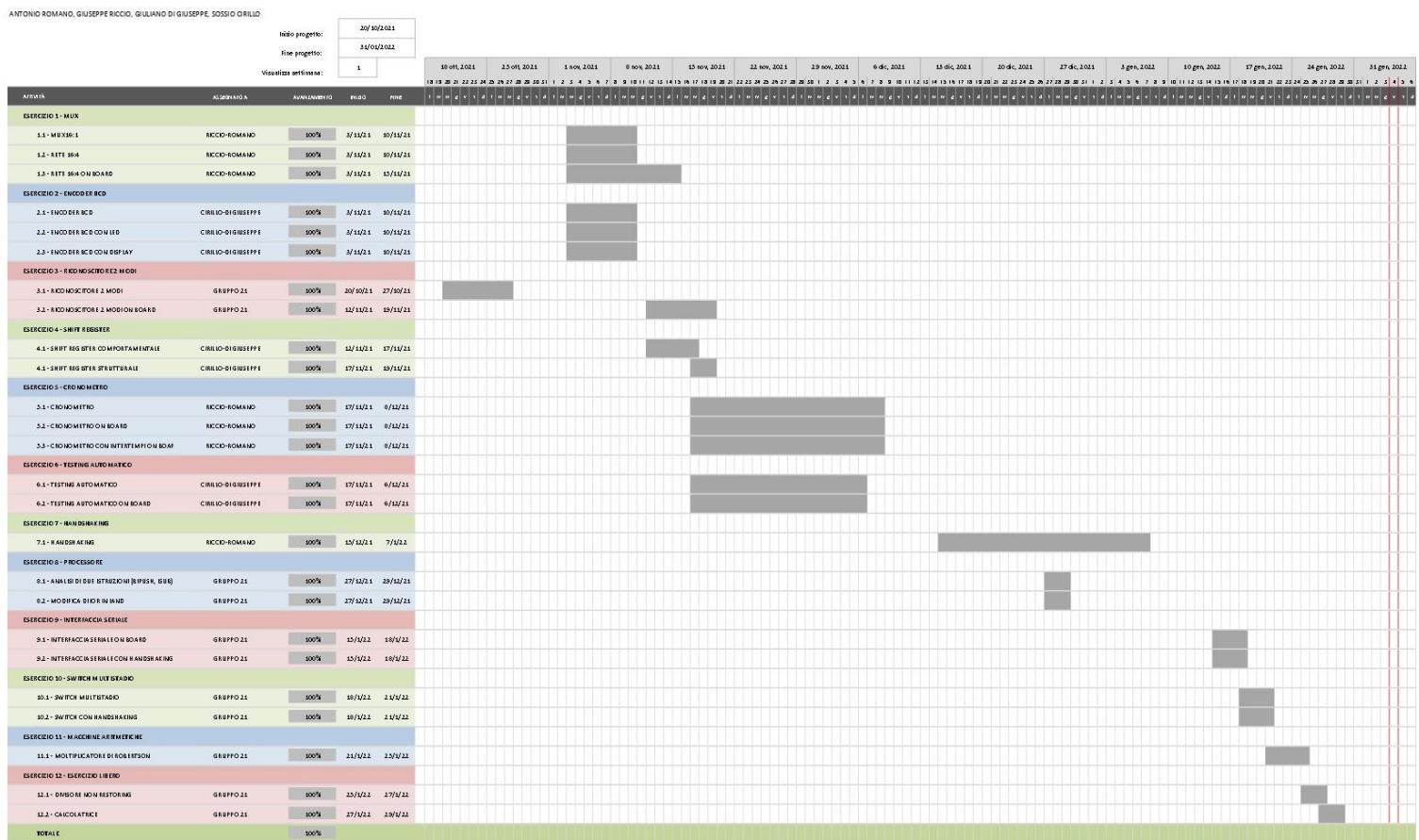


Prefazione

Per la realizzazione del progetto si è seguito un diagramma di Gantt, uno strumento utile per la pianificazione della nostra progettazione. Si è realizzata una **panoramica dei compiti programmati** attraverso la quale tutte le parti interessate sono a conoscenza dei **compiti e delle rispettive scadenze**.

GRUPPO 21 - PROGETTO ASD

ANTONIO ROMANO, GIUSEPPE RICCI, GIULIANO DI GIUSEPPE, OSSO CIRILLO



Ogni settimana, si è organizzato una brainstorming – riunione dove si sono discusse tutte le nostre scelte di realizzazione e motivate in gruppo al fine di essere sempre sicuri di tali scelte.



Lavoro svolto con dedizione e pazienza
ai tempi del **CoronaVirus**.

Gli autori



1 Multiplexer

Si realizza un Multiplexer seguendo le specifiche richieste dal cliente:

- 1.1 Progettare, implementare in VHDL e testare mediante simulazione un **multiplexer indirizzabile 16:1**, utilizzando un approccio di progettazione per composizione a partire da **multiplexer 4:1**.
- 1.2 Utilizzando il componente sviluppato al punto precedente, progettare, implementare in VHDL e testare mediante simulazione una **rete di interconnessione a 16 sorgenti e 4 destinazioni**.
- 1.3 Sintetizzare ed implementare su board il progetto della rete di interconnessione sviluppato al punto 1.2, utilizzando gli switch per fornire gli input di selezione e i led per visualizzare i 4 bit di uscita. Per quanto riguarda i 16 bit dato in input, essi possono essere precaricati nel sistema oppure immessi anch'essi mediante switch, sviluppando in questo secondo caso un'apposita rete di controllo per l'acquisizione.

1.1 Progettazione del Multiplexer indirizzabile 16:1

Per la progettazione del Multiplexer in questione, come dalle specifiche, si utilizza un approccio di progettazione modulare, ovvero implementando componenti più piccoli e componendoli tra di loro. In questo caso, per la realizzazione del Multiplexer indirizzabile 16:1 si utilizzano cinque Multiplexer 4:1, che a loro volta saranno implementati come composizione di tre Multiplexer 2:1 notevoli.

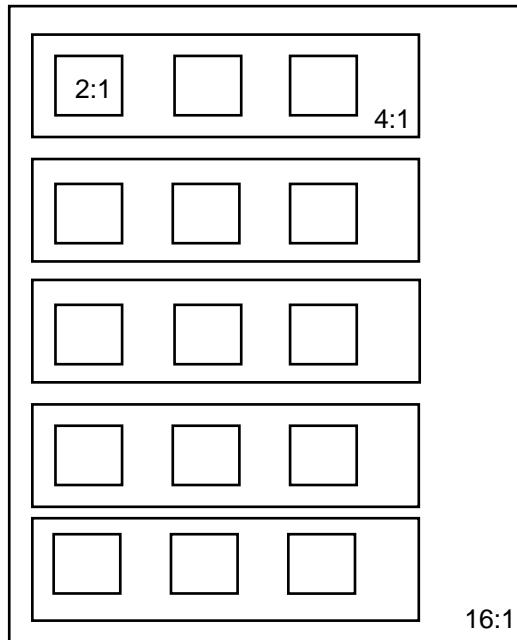


FIG. 1.0 – Concetto grafico

Il **Mux 2:1** è una macchina combinatoria notevole, indirizzabile avente una sola linea di selezione e due ingressi.

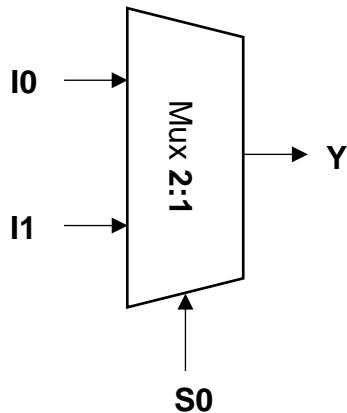


FIG. 1.1 – Mux 2:1

Perché si utilizzano *tre* Mux 2:1 per la realizzazione del Mux 4:1? La risposta è nel seguente grafico:

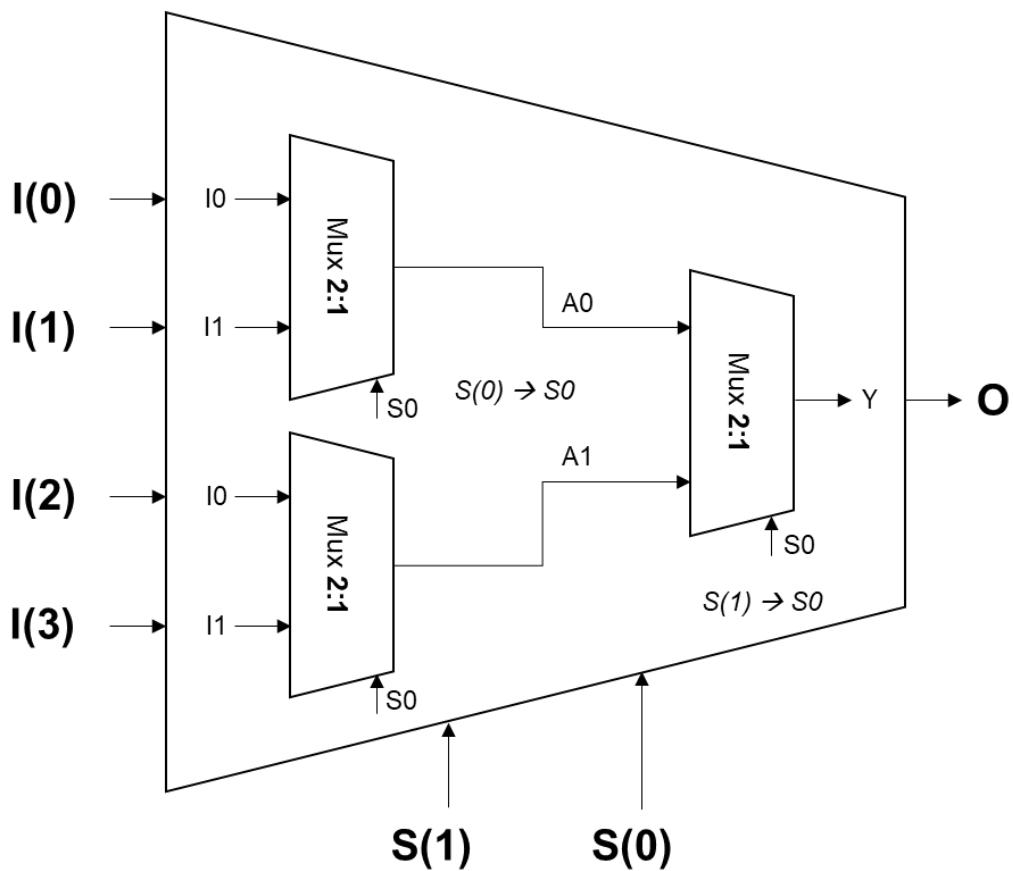


FIG. 1.2 – Mux 4:1

I **quattro** input mappati in ingresso ai primi due Mux 2:1 la cui uscita è determinata dall'input di selezione **S0**. I due segnali in uscita dai primi due Mux 2:1 entrano nell'ultimo Mux 2:1 selezionato dall'ingresso **S1** e l'uscita Y mappata in **O**.

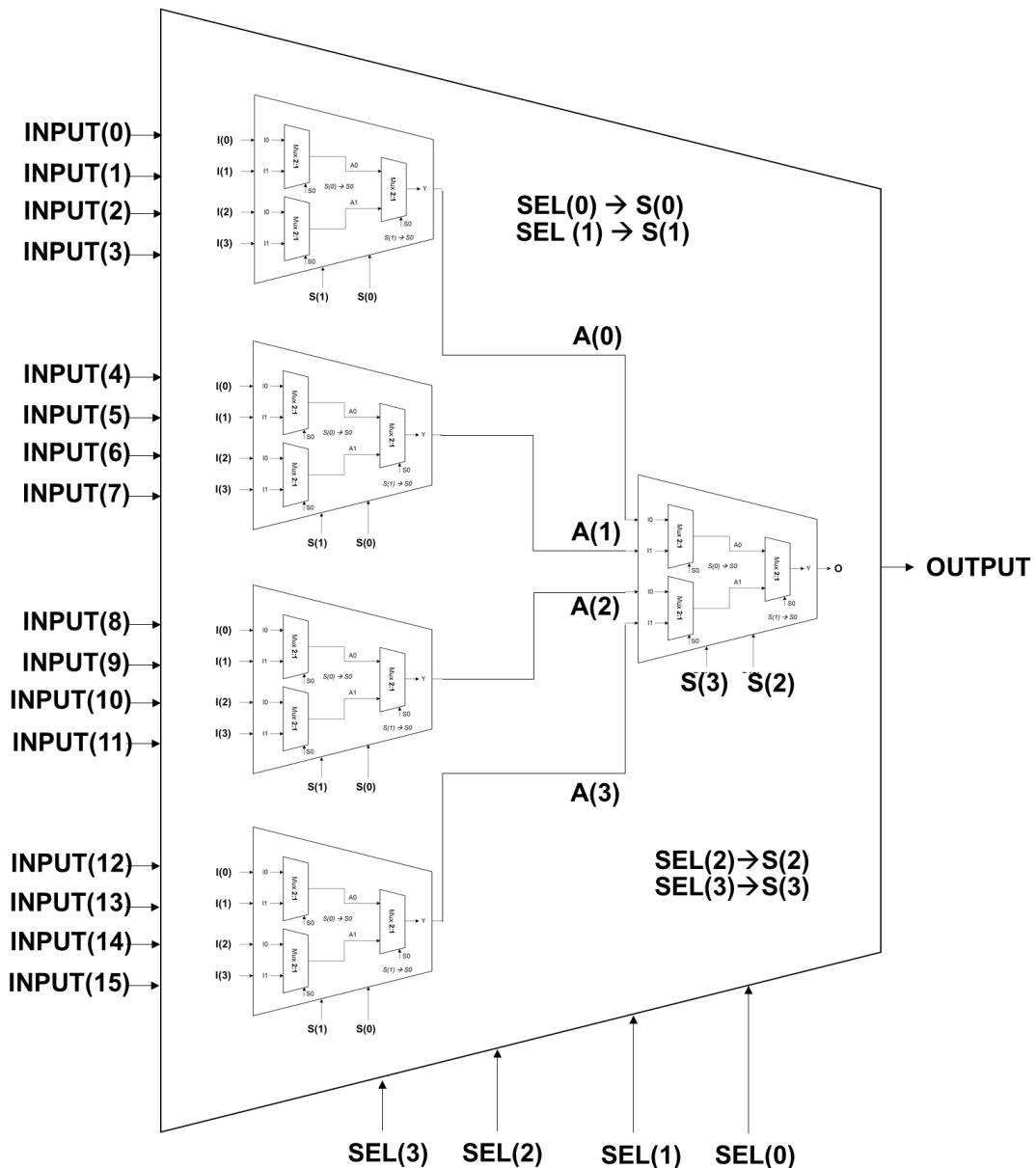


FIG. 1.3 – Mux 16:1

Si fa uso del Mux 4:1 composito per la realizzazione del Mux 16:1. Dalla **FIG 1.3**, si evincono i 16 ingressi di **INPUT(0...15)**, mappati per ogni 4 input ad ogni Mux 4:1 composito. Le uscite sono gestite dai segnali interni **A(0...3)** i quali vanno in ingresso all'ultimo Mux 4:1 composito. L'uscita dell'ultimo Mux 4:1 composito corrisponde all'uscita complessiva del sistema.

1.1.1 Codice VHDL del Multiplexer

Si inizia dal componente notevole, il **Multiplexer 2:1 (multiplexer_2_1.vhd)**:

Si definisce l'entità del MUX 2:1

Multiplexer_2_1.vhd

```
entity multiplexer_2_1 is
  Port ( I0 : in STD_LOGIC;
         I1 : in STD_LOGIC;
         S0 : in STD_LOGIC;
         Y : out STD_LOGIC
       );
end multiplexer_2_1;
```

E l'architettura:

```
architecture behavioral of multiplexer_2_1 is
begin
  Y <= I0 when S0 = '0' else
             I1 when S0 = '1' else
             '-';
end behavioral;
```

Dopo l'implementazione del MUX 2:1, si passa all'implementazione del **Multiplexer 4:1** (**multiplexer_4_1.vhd**) riutilizzando proprio il **multiplexer_2_1**.

Multiplexer_4_1.vhd

```
entity multiplexer_4_1 is
  Port (
    I: in STD_LOGIC_VECTOR (0 to 3);
    S: in STD_LOGIC_VECTOR (1 downto 0);
    O: out STD_LOGIC
  );
end multiplexer_4_1;

architecture structural of multiplexer_4_1 is
  -- riutilizzo del componente multiplexer_2_1
  component multiplexer_2_1 is
    port (
      I0 : in STD_LOGIC;
      I1 : in STD_LOGIC;
      S0 : in STD_LOGIC;
      Y : out STD_LOGIC
    );
  end component;
  -- segnali interni
  signal A : STD_LOGIC_VECTOR (0 to 1);
```



```
begin

-- si definiscono i multiplexer che si interfacciano con
l'ingresso
mux0_1: FOR j in 0 to 1 GENERATE m: multiplexer_2_1

    -- si mappano con gli ingressi indicati dalle figure
    port map(
        I0 => I(j*2),
        I1 => I(j*2 +1),
        S0 => S(0),
        Y => A(j)
    );

end GENERATE;
-- si mappa il multiplexer collegato all'uscita
mux2: multiplexer_2_1

    port map(
        I0 => A(0),
        I1 => A(1),
        S0 => S(1),
        Y => O
    );

end structural;
```

Dopo l'implementazione del MUX 4:1, si passa all'implementazione del **Multiplexer 16:1** (**multiplexer_16_1.vhd**) riutilizzando proprio il **multiplexer_4_1**.

Multiplexer_16_1.vhd

```
entity multiplexer_16_1 is

    Port (
        INPUT: in STD_LOGIC_VECTOR(0 to 15);
        SEL: in STD_LOGIC_VECTOR(3 downto 0);
        OUTPUT: out STD_LOGIC
    );

end multiplexer_16_1;

architecture structural of multiplexer_16_1 is

    -- riutilizzo del componente multiplexer_4_1
    component multiplexer_4_1 is

        Port (
            I: in STD_LOGIC_VECTOR (0 to 3);
            S: in STD_LOGIC_VECTOR (1 downto 0);
            O: out STD_LOGIC
        );
    end component;
```



```

end component;

-- segnali interni
signal A : STD_LOGIC_VECTOR (0 to 3);

begin

-- si definiscono I 4 mux che si interfacciano con l'ingresso
mux0_3: FOR j IN 0 to 3 GENERATE m: multiplexer_4_1

    -- si mappano i segnali con i segnali di ingresso esterni
    Port map (
        I(0 to 3) => INPUT(j*4 to j*4+3),
        S(1 downto 0) => SEL(1 downto 0),
        O => A(j)
    );
end GENERATE;

-- si definisce il 5° mux che si interfaccia con l'uscita
mux4: multiplexer_4_1

    -- si mappano i segnali con i segnali di ingresso interni
    port map (
        I(0 to 3) => A(0 to 3),
        S(1 downto 0) => SEL(3 downto 2),
        O => OUTPUT
    );
end structural;
-----
```

1.1.2 La simulazione del Multiplexer

Come di consueto, la simulazione sarà fatta stimolando l'implementazione del **multiplexer_16_1** attraverso il *Testbench* (**multiplexer_16_1_TB.vhd**)

Multiplexer_16_1_TB.vhd

```

entity multiplexer_16_1_TB is
end multiplexer_16_1_TB;

architecture Behavioral of multiplexer_16_1_TB is

component multiplexer_16_1 is

    Port (
        INPUT: in STD_LOGIC_VECTOR (0 to 15);
        SEL: in STD_LOGIC_VECTOR (3 downto 0);
        OUTPUT: out STD_LOGIC
    );
end component;

begin component;
```

```
    signal inputs : STD_LOGIC_VECTOR (0 to 15);
```



```
signal selections : STD_LOGIC_VECTOR (3 downto 0);
signal outputs : STD_LOGIC;

begin

--Unit Under Test
uut: multiplexer_16_1

    -- mapping dei segnali da simulare
    port map (
        INPUT => inputs,
        SEL => selections,
        OUTPUT => outputs
    );

-- TEST
stim_proc: process

begin

--selezione linea INPUT0
wait for 10 ns;
inputs <= "1000000000000000";
selections <= "0000";
wait for 10 ns;
assert outputs <= '1'
report "Errore sulla Linea 0"
severity failure;

--selezione linea INPUT6
wait for 10 ns;
inputs <= "0000001000000000";
selections <= "0110";
wait for 10 ns;
assert outputs <= '1'
report "Errore sulla Linea 6"
severity failure;

--selezione linea INPUT12
wait for 10 ns;
inputs <= "000000000001000";
selections <= "1100";
wait for 10 ns;
assert outputs <= '1'
report "Errore sulla Linea 12"
severity failure;

--selezione linea INPUT3
wait for 10 ns;
inputs <= "000100000000000";
selections <= "0011";
wait for 10 ns;
assert outputs <= '1'
report "Errore sulla Linea 3"
severity failure;
```



```
--selezione linea INPUT15
wait for 10 ns;
inputs <= "0000000000000001";
selections <= "1111";
wait for 10 ns;
assert outputs <= '1'
report "Errore sulla Linea 15"
severity failure;

end process;
end Behavioral;
```



FIG. 1.4 – Simulazione Multiplexer 16:1

In definitiva, come è possibile notare dalla **FIG 1.4**, osservando la simulazione all'istante 50ns si ha che l'**INPUT** è “00000000001000” e il **SEL** è “1100” ovvero viene selezionata la linea 12 (**c**) e quindi in uscita (**OUTPUT**) si avrà 1 perché tale linea è ad un valore logico alto.

1.2 Progettazione della Rete di Interconnessione a 16:4

Per la realizzazione di tale rete si utilizza il **MUX 16_1**, che con un'opportuna selezione degli ingressi, inoltra il segnale in uscita del multiplexer in ingresso ad un **DEMUX 1_4 (FIG. 1.5)**; che a sua volta, il demultiplexer inoltrerà su una delle quattro possibili destinazioni in modo tale da realizzare una vera e propria rete di interconnessione a 16 ingressi e 4 uscite.

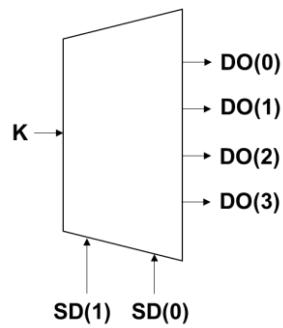


FIG. 1.5 – Demultiplexer 1_4

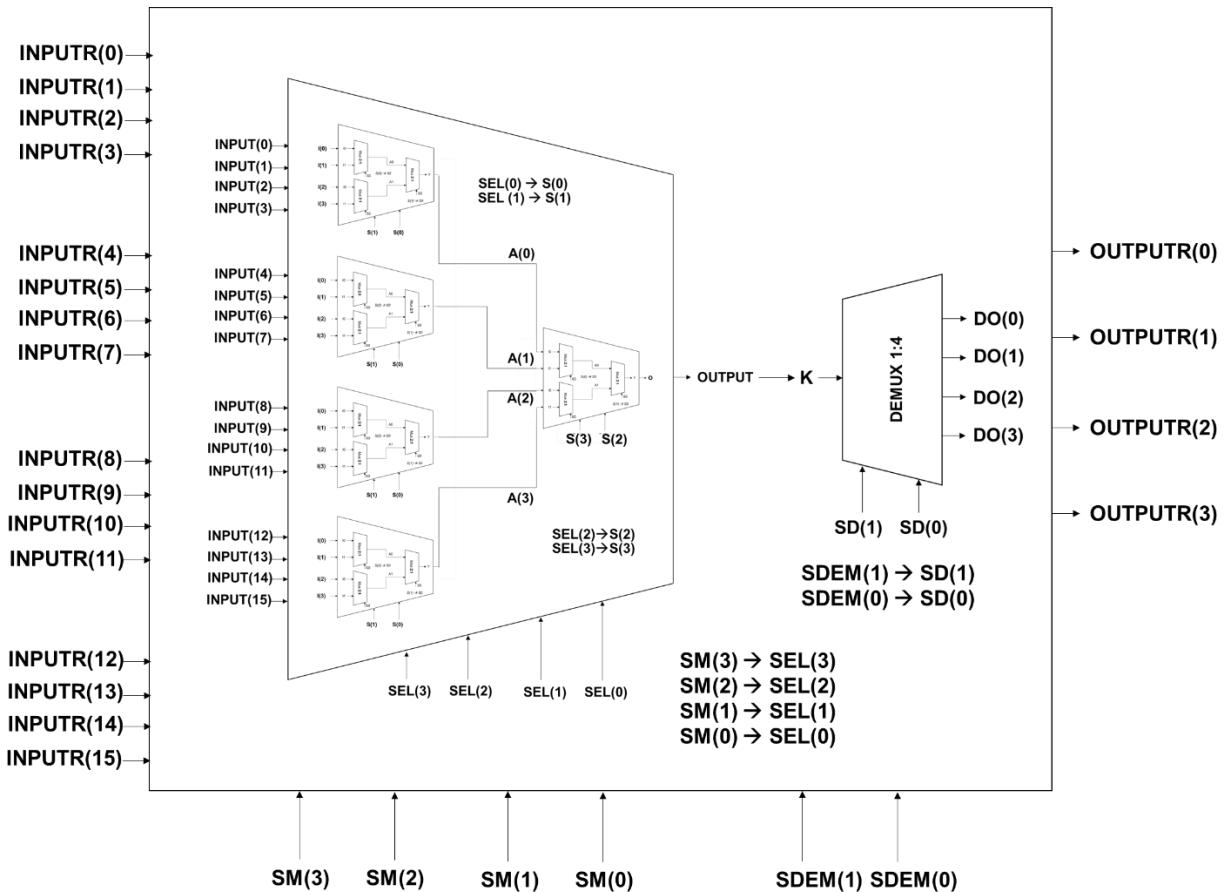


FIG. 1.6 – Rete di Interconnessione a 16 sorgenti e 4 destinazioni

Dalla **FIG 1.6** si trae l'approccio modulare proposto nel paragrafo 1, in cui ogni componente è composizione di altri componenti più semplici, in particolare gli ingressi della rete di interconnessione (**INPUTR[0..15]**) sono mappati agli ingressi del **MUX 16:1** (**INPUT[0..15]**). Di tali ingressi, soltanto uno viene posto in uscita (**OUTPUT**) sulla base del valore di selezione (**SM[0..3]**) indicato. Quest'ultimo è inoltrato all'ingresso del **DEMUX [1..4]** che in accordo con il valore **SDEM[0,1]** seleziona la destinazione di uscita (**OUTPUTR[0..3]**). Si noti dunque che il valore di **OUTPUTR** sarà formato da un valore di 4 bit di cui solo uno sarà alto (es. "0010").

1.2.1 Codice VHDL della Rete di Interconnessione 16:4

Si implementa inizialmente l'ulteriore componente della rete, ovvero il **DEMUX 1:4**:

Demultiplexer_1_4.vhd

```

entity demultiplexer_1_4 is
  Port (
    K : in STD_LOGIC;
    SD : in STD_LOGIC_VECTOR (1 downto 0);
    DO : out STD_LOGIC_VECTOR (0 to 3)
  );
end demultiplexer_1_4;

architecture Behavioral of demultiplexer_1_4 is
begin

  -- l'uscita del demultiplexer_1_4 è un vettore di 4 bit pertanto in
  -- correlazione della selezione SD si utilizza la concatenazione tra K
  -- e i valori di bit rimanenti.

  process (SD,K)
  begin
    if SD="00" then
      DO <= K&"000";
    elsif SD="01" then
      DO <= '0'&K&"00";
    elsif SD="10" then
      DO <= "00"&K&'0';
    elsif SD="11" then
      DO <= "000"&K;
    end if;
  end process;
end Behavioral;

```

A questo punto si passa all'implementazione generale della rete di interconnessione:

Rete_16_4.vhd

```

entity rete_16_4 is
  Port (
    -- si inizializza INPUTR con tutti "1" poiché i switch presenti
    -- sulla board non sono sufficienti a rappresentare tutti gli ingressi
    -- della rete in questione (sia quelli delle linee e sia quelli di
    -- selezione complessivi)
    INPUTR : in STD_LOGIC_VECTOR(0 to 15) := "1111111111111111";
    SM : in STD_LOGIC_VECTOR(3 downto 0);
    SDEM : in STD_LOGIC_VECTOR(1 downto 0);
    OUTPUTR : out STD_LOGIC_VECTOR(0 to 3)
  );
end rete_16_4;

architecture structural of rete_16_4 is

```



```

component multiplexer_16_1 is
    Port (
        INPUT: in STD_LOGIC_VECTOR(0 to 15);
        SEL: in STD_LOGIC_VECTOR(3 downto 0);
        OUTPUT: out STD_LOGIC
    );
end component;

component demultiplexer_1_4 is
    Port (
        K : in STD_LOGIC;
        SD : in STD_LOGIC_VECTOR (1 downto 0);
        DO : out STD_LOGIC_VECTOR (0 to 3)
    );
end component;

begin
    mux: multiplexer_16_1
        port map (
            INPUT(0 to 15) => INPUTR(0 to 15),
            SEL(3 downto 0) => SM(3 downto 0),
            OUTPUT => A
        );
    demux: demultiplexer_1_4
        port map(
            K => A,
            SD(1 downto 0) => SDEM(1 downto 0),
            DO(0 to 3) => OUTPUTR(0 to 3)
        );
end structural;

```

1.2.2 La simulazione della Rete

Dall'analisi della simulazione in **FIG 1.7** è possibile notare il funzionamento della rete 16:4 ad esempio all'istante 10 ns sulle 16 linee di input viene selezionata (*selections1*) la nona (1001) il cui valore logico è 1 (il valore che si passa). Tale valore viene riportato in uscita alla rete complessiva sulla linea 2 (scelta casuale) come evidenziato dal valore di *selections2*. Dunque, l'output è 0010 ovvero il valore della linea 9 in ingresso è trasmesso alla linea 2 in uscita. Invece se avessimo selezionato in uscita la linea 0 (istante 30 ns) il corrispondente valore in output sarebbe stato 1000.

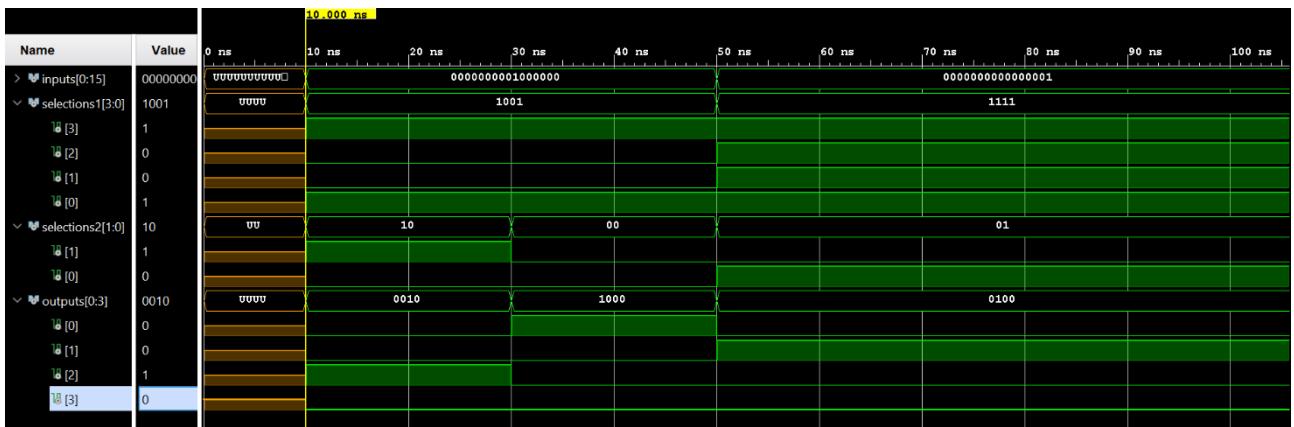


FIG. 1.7 – Simulazione della rete 16:4

rete_16_4_TB.vhd

```

entity rete_16_4_TB is
--  Port ( );
end rete_16_4_TB;
architecture Behavorial of rete_16_4_TB is
component rete_16_4 is
    Port (
        INPUTR : in STD_LOGIC_VECTOR(0 to 15);
        SM : in STD_LOGIC_VECTOR(3 downto 0);
        SDEM : in STD_LOGIC_VECTOR(1 downto 0);
        OUTPUTR : out STD_LOGIC_VECTOR(0 to 3)
    );
end component;
signal inputs: STD_LOGIC_VECTOR(0 to 15);
signal selections1: STD_LOGIC_VECTOR(3 downto 0);
signal selections2: STD_LOGIC_VECTOR(1 downto 0);
signal outputs: STD_LOGIC_VECTOR(0 to 3);
begin
--Unit Under Test
uut: rete_16_4
    Port map (
        INPUTR => inputs,
        SM => selections1,
        SDEM => selections2,
        OUTPUTR => outputs
    );
stim_proc: process
begin
    --selezione INPUT9
    wait for 10 ns;
    inputs <= "0000000001000000";
    selections1 <= "1001";
    selections2 <= "10";
    wait for 10 ns;

    --selezione INPUT9
    wait for 10 ns;
    selections2 <= "00";
    wait for 10 ns;

```



```
--selezione INPUT15
wait for 10 ns;
inputs <= "0000000000000001";
selections1 <= "1111";
selections2 <= "01";
wait for 10 ns;
assert outputs <= "0100"
report "Errore sulla Linea 15"
severity failure;
wait;

end process;
end Behavorial;
```

1.2.3 Implementazione su board Rete 16:4

Per l'implementazione sulla board si è settato tutto ad 1 gli inputs perché il numero di switch a disposizione sulla board è insufficiente per gestire sia gli inputs e sia le selezioni del MUX e del DEMUX presenti nel sistema. Pertanto, si è scelto di settare 6 SWITCH e 4 LED sul *constraint*.

```
##Switches
set_property -dict { PACKAGE_PIN J15    IOSTANDARD LVCMOS33 } [get_ports { SDEM[0] }];
set_property -dict { PACKAGE_PIN L16    IOSTANDARD LVCMOS33 } [get_ports { SDEM[1] }];
set_property -dict { PACKAGE_PIN M13    IOSTANDARD LVCMOS33 } [get_ports { SM[0] }];
set_property -dict { PACKAGE_PIN R15    IOSTANDARD LVCMOS33 } [get_ports { SM[1] }];
set_property -dict { PACKAGE_PIN R17    IOSTANDARD LVCMOS33 } [get_ports { SM[2] }];
set_property -dict { PACKAGE_PIN T18    IOSTANDARD LVCMOS33 } [get_ports { SM[3] }];

## LEDs
set_property -dict { PACKAGE_PIN H17    IOSTANDARD LVCMOS33 } [get_ports { OUTPUTR[0] }];
set_property -dict { PACKAGE_PIN K15    IOSTANDARD LVCMOS33 } [get_ports { OUTPUTR[1] }];
set_property -dict { PACKAGE_PIN J13    IOSTANDARD LVCMOS33 } [get_ports { OUTPUTR[2] }];
set_property -dict { PACKAGE_PIN N14    IOSTANDARD LVCMOS33 } [get_ports { OUTPUTR[3] }];
```



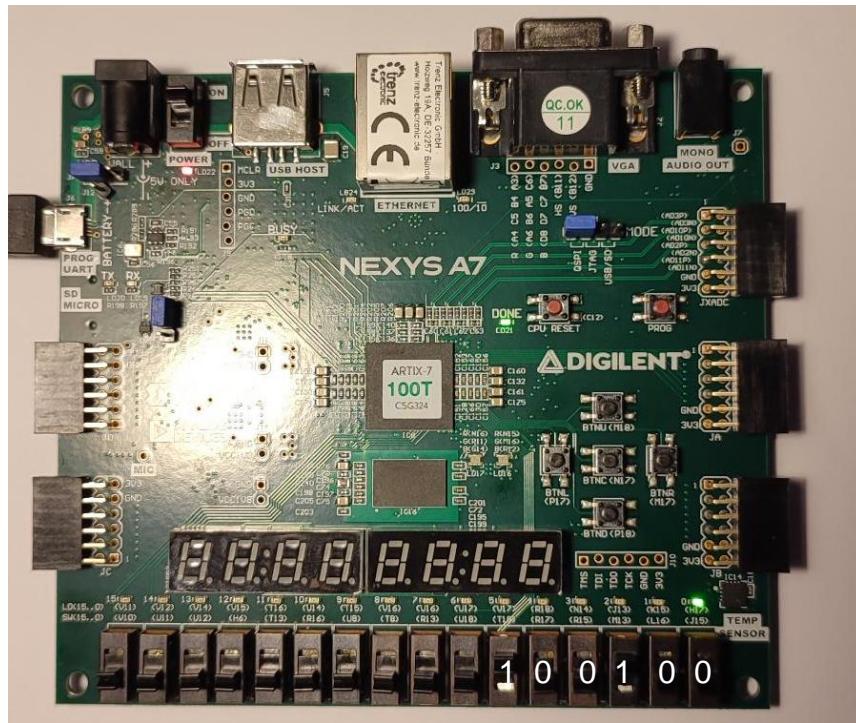


FIG. 1.8 – Uscita LED0 con MUX a 1001 (9) e DEMUX a 00 (0)

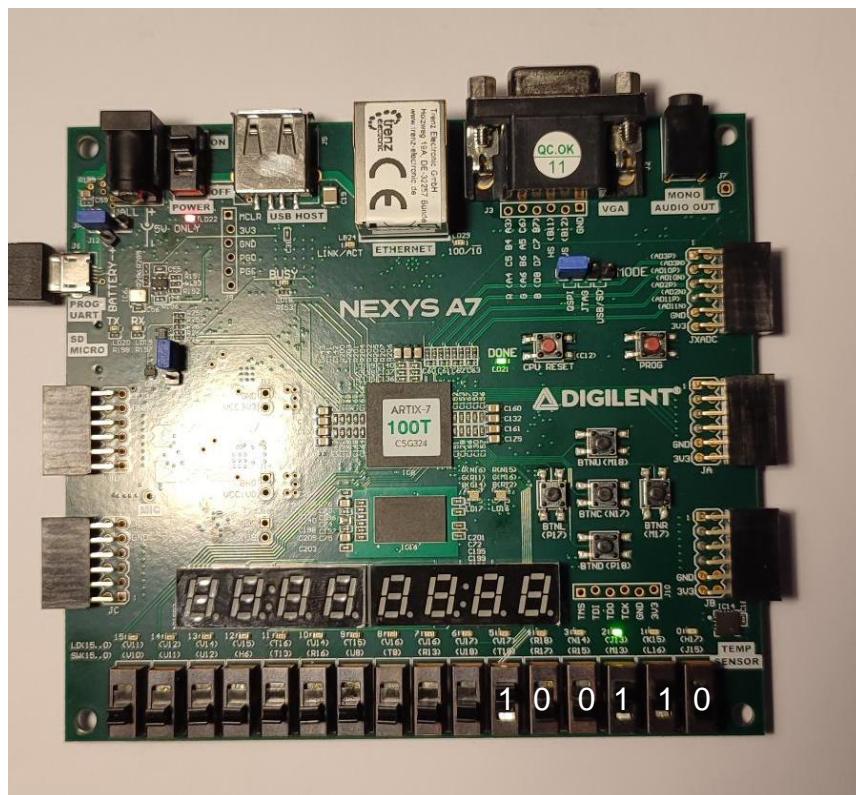


FIG. 1.9 – Uscita LED2 con MUX a 1001 (9) e DEMUX a 10 (2)

2 Encoder BCD

Si realizza un Encoder BCD seguendo le specifiche richieste dal cliente:

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

Input: 0000000001 à Output: 0000 (cifra 0)

Input: 0000000010 à Output: 0001 (cifra 1)

Input: 0000000100 à Output: 0010 (cifra 2)

....

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

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

2.1 Progettazione del Encoder BCD

L'encoder è un dispositivo che riceve in ingresso una parola ad n bit e ha come uscita una parola ad m bit, con $m < n$. L'ingresso è tendenzialmente decodificato mentre l'uscita è codificata; quindi, l'encoder riceve in ingresso n bit di cui solo uno è alto e dà in uscita una parola codificata su $m = \lceil \log_2 n \rceil$ bit.

Per la progettazione di un Encoder BCD si utilizza anche in questo caso un approccio modulare. Si compone, cioè, l'Encoder BCD tramite due componenti. Il primo componente è un arbitro di priorità e il secondo è l'encoder vero e proprio. L'arbitro di priorità ha lo scopo di evitare situazioni in cui, per errore o volutamente, in ingresso all'encoder si presentino n bit in cui più di uno sia alto, infatti l'arbitro ha come ingresso una parola di n bit e come uscita una parola n bit (la quale diventerà l'ingresso dell'encoder) ma, se in ingresso c'è più di un bit alto, allora l'arbitro andrà a considerare come valido solamente il bit più significativo, riportando in uscita quindi una parola di n bit in cui solo quello più significativo sarà alto.

Lo schema del Priority Encoder è mostrato in Figura 2.1.

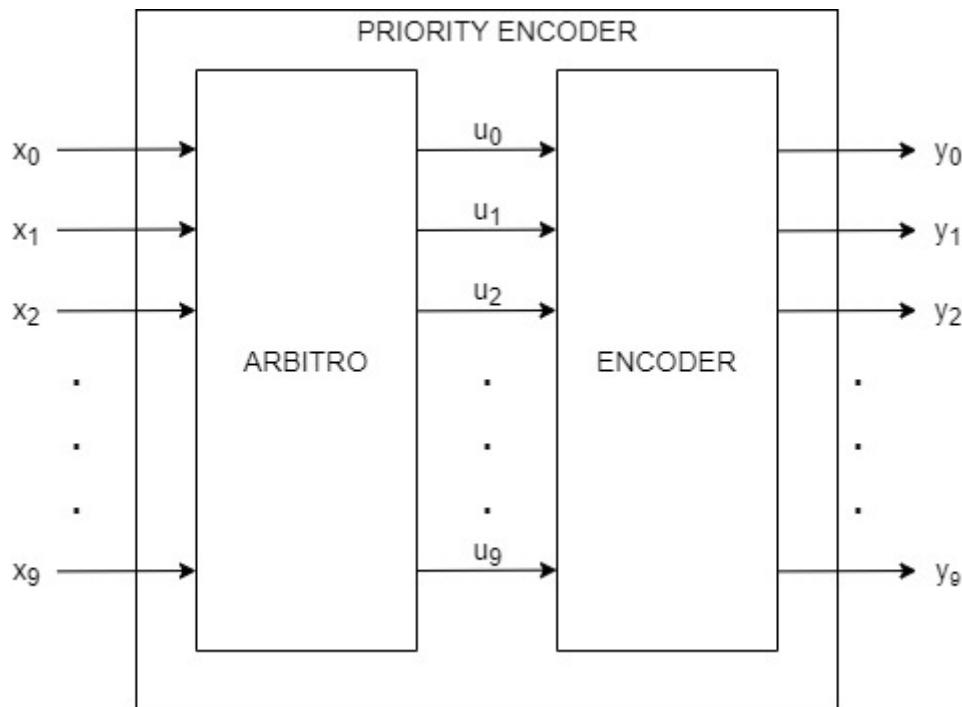


FIG. 2.1

Se la board a nostra disposizione non avesse avuto un numero sufficiente di switch su cui mappare i vari ingressi, le soluzioni potevano essere svariate:

- Creare un processo che gestisse gli input in più passi, ad esempio attraverso i bottoni, assegnando ad esempio $10/n$ segnali alla volta in base agli switch disponibili.
- Biforcare gli ingressi, cioè creare un dispositivo che prende un filo in ingresso e ne costruisce due.
- Precaricare nella memoria un insieme di sequenze.

Così come è stato implementato, il sistema comprende 10 ingressi mappati su altrettanti switch e 4 uscite, mappate prima su altrettanti led e successivamente visualizzabile tramite una singola cifra sul display a 7 segmenti.

2.1.1 Codice VHDL del Encoder BCD

CODICE ARBITRO

Nel codice dell'arbitro, si definisce come deve essere l'output in modo tale che, anche se più di uno degli ingressi è alto, l'uscita avrà comunque solo e soltanto un bit alto. Viene trattato con un modello di tipo Dataflow.

Arbiter.vhd

```

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

```

```

architecture Dataflow of Arbiter is

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

```

```
end Dataflow;
```

CODICE ENCODER

Nel codice dell'encoder si specifica l'uscita codificata in base all'ingresso decodificato. Anch'esso è trattato col modello Dataflow.

Encoder.vhd

```

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

architecture Dataflow of Encoder is

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

```

CODICE PRIORITY ENCODER

Nel priority encoder invece, si usa un modello strutturale, in quanto si utilizzano i componenti prima specificati (Arbiter e Encoder) utilizzando proprio il costrutto COMPONENT. Si vanno poi a mappare i vari ingressi/uscite dei vari componenti nel modo illustrato nello schema.

PriorityEncoder.vhd

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

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

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

    signal u : STD_LOGIC_VECTOR(9 downto 0);
begin

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

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

end Structural;
```



2.1.2 La simulazione del Encoder BCD

CODICE TESTBENCH

Il codice usato per testare il Priority Encoder è il seguente:

PriorityEncoder_TB.vhd

```

ENTITY PriorityEncoder_TB IS
END PriorityEncoder_TB;

ARCHITECTURE Behavioral OF PriorityEncoder_TB IS

    COMPONENT PriorityEncoder
        PORT (
            X : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
            Y : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
        );
    END COMPONENT;

    SIGNAL input : STD_LOGIC_VECTOR(9 DOWNTO 0) := "0000000000";
    SIGNAL output : STD_LOGIC_VECTOR(3 DOWNTO 0);

BEGIN

    uut : PriorityEncoder PORT MAP(
        X => input,
        Y => output
    );

    stim_proc : PROCESS
    BEGIN
        WAIT FOR 100 ns;

        input <= "1010101010";
        WAIT FOR 50 ns;
        input <= "0101011111";
        WAIT FOR 50 ns;
        input <= "0000101010";
        WAIT FOR 50 ns;
        input <= "0010111111";
        WAIT FOR 50 ns;
        input <= "0001110111";
        WAIT FOR 50 ns;
        input <= "0000000001";
        WAIT FOR 50 ns;
        input <= "0000010101";
        WAIT FOR 50 ns;
        input <= "0000000010";
        WAIT FOR 50 ns;
        input <= "1000000000";
        WAIT FOR 50 ns;
        input <= "0000000100";
        WAIT FOR 50 ns;
        input <= "0000000000";
        WAIT;
    
```



```

    END PROCESS;
END Behavioral;
-----
```

Come si può vedere, sono stati inseriti sia input con un solo bit alto, sia input con più di un bit alto, per testare il funzionamento dell'arbitro.

Il risultato è mostrato in Figura 2.2 nell'intervallo 100-340ns e in Figura 2.3 nell'intervallo 340-600ns:

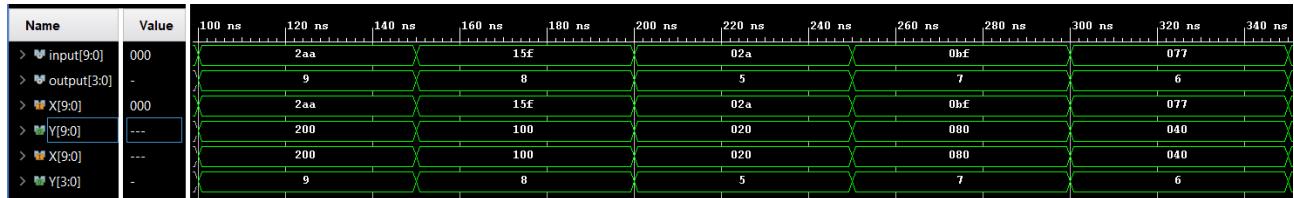


FIG. 2.2

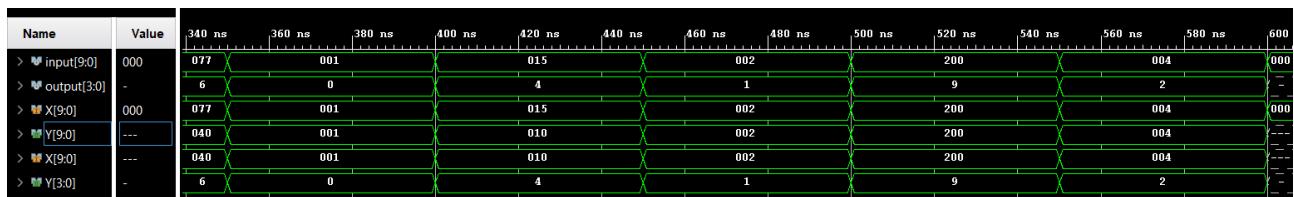


FIG. 2.3

Dalla simulazione si deve specificare che input[9:0] si riferisce all'input immesso dall'utente e output[3:0] si riferisce all'output dell'intero Priority Encoder, successivamente X[9:0] si riferisce all'input dell'arbitro (che corrisponde all'input dell'utente) e Y[9:0] si riferisce all'output dell'arbitro, che corrisponde al successivo X[9:0] che sarebbe l'input dell'encoder, ed infine c'è Y[3:0] che è l'uscita dell'encoder interno, che corrisponde all'uscita finale del Priority Encoder.

Dopo un'attesa di 100ns, vengono inseriti gli input uno ad uno con intervallo di 50ns.

Il primo input è 2aa, cioè "1010101010", cioè il primo input presenta più bit alti. Come si può notare, l'uscita dell'arbitro risulta essere invece 200, cioè "1000000000", ovvero va a considerare solo il primo bit più significativo, azzerando tutti i successivi, esattamente come dovrebbe fare. Siccome ad essere alto è il nono bit più significativo, allora in uscita si presenta la configurazione "9", che in binario sarebbe "1001".

Gli input successivi confermano la correttezza della macchina, sia che essi siano configurati esattamente, cioè con un solo bit alto, sia in caso contrario.

2.1.3 Implementazione sulla board tramite i led

Per poter implementare il Priority Encoder sulla board, bisogna decidere su che switch mappare gli ingressi e su che led mappare le uscite. Si decide arbitrariamente di mappare i 10 ingressi sugli switch da V10 a U18 mentre le uscite verranno mappate sui led da N14 a H17. Le linee da decommentare e modificare nel *constraints* sono:

##Switches

```

set_property -dict { PACKAGE_PIN U18    IO_STANDARD LVCMOS33 } [get_ports { X[9] }];
#IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13    IO_STANDARD LVCMOS33 } [get_ports { X[8] }];
#IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IO_STANDARD LVCMOS18 } [get_ports { X[7] }];
#IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IO_STANDARD LVCMOS18 } [get_ports { X[6] }];
#IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16    IO_STANDARD LVCMOS33 } [get_ports { X[5] }];
#IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13    IO_STANDARD LVCMOS33 } [get_ports { X[4] }];
#IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6    IO_STANDARD LVCMOS33 } [get_ports { X[3] }];
#IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12    IO_STANDARD LVCMOS33 } [get_ports { X[2] }];
#IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11    IO_STANDARD LVCMOS33 } [get_ports { X[1] }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10    IO_STANDARD LVCMOS33 } [get_ports { X[0] }];
#IO_L21P_T3_DQS_14 Sch=sw[15]

```

LEDs

```

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

```

Come esempio, riportiamo quelli di Figura 2.4 e 2.5. In particolare, possiamo vedere in Figura 2.4 che la configurazione di ingresso risulta avere solo un bit alto, cioè quello più significativo mappato sullo switch U18. La configurazione di ingresso è quindi 1000000000. In questo caso non è necessario l'intervento dell'arbitro e l'uscita risulta corretta, 1001.

Per vedere se l'arbitro funziona o meno, vediamo la Figura 2.5, in cui, mantenendo U18 alzato, sono stati alzati anche gli switches U12, H6, T13. Notiamo che la configurazione dell'uscita non varia; ciò vuol dire che l'arbitro considera solo il bit più significativo se in ingresso ci sono più bit alti. Il Priority Encoder funziona correttamente.

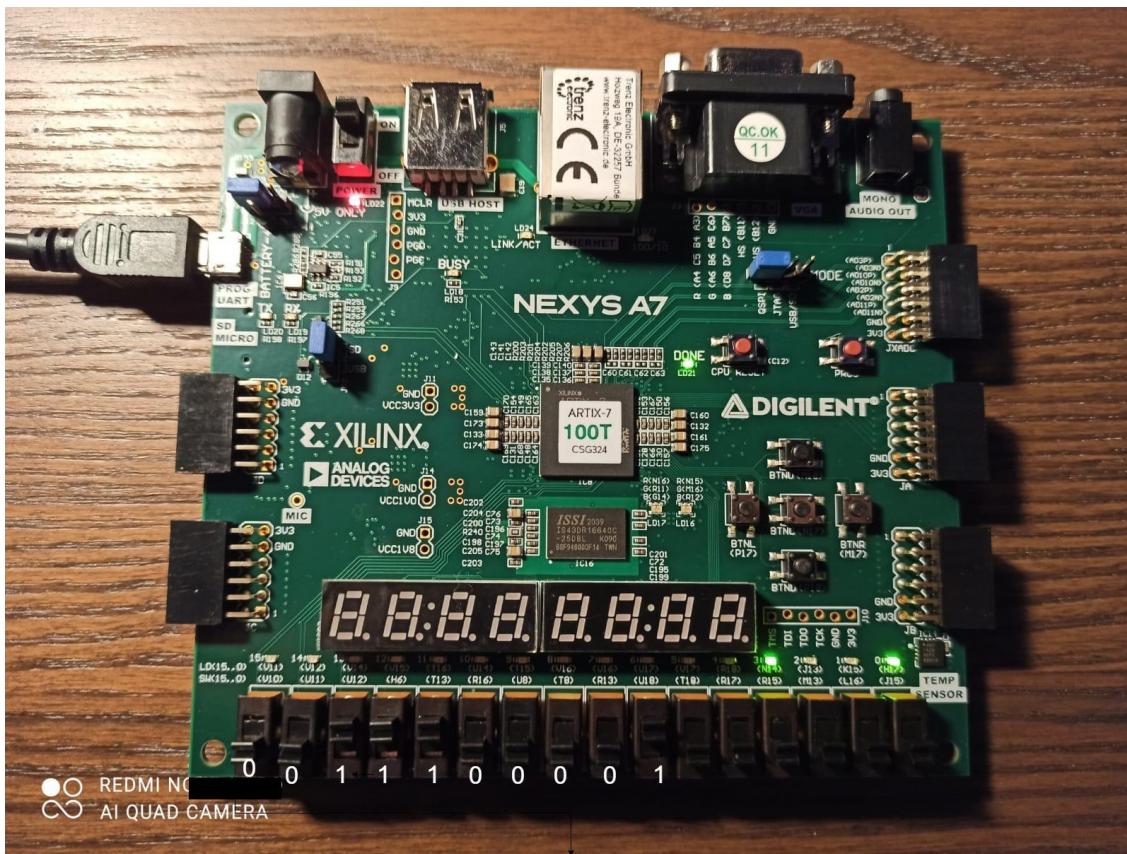


FIG. 2.4

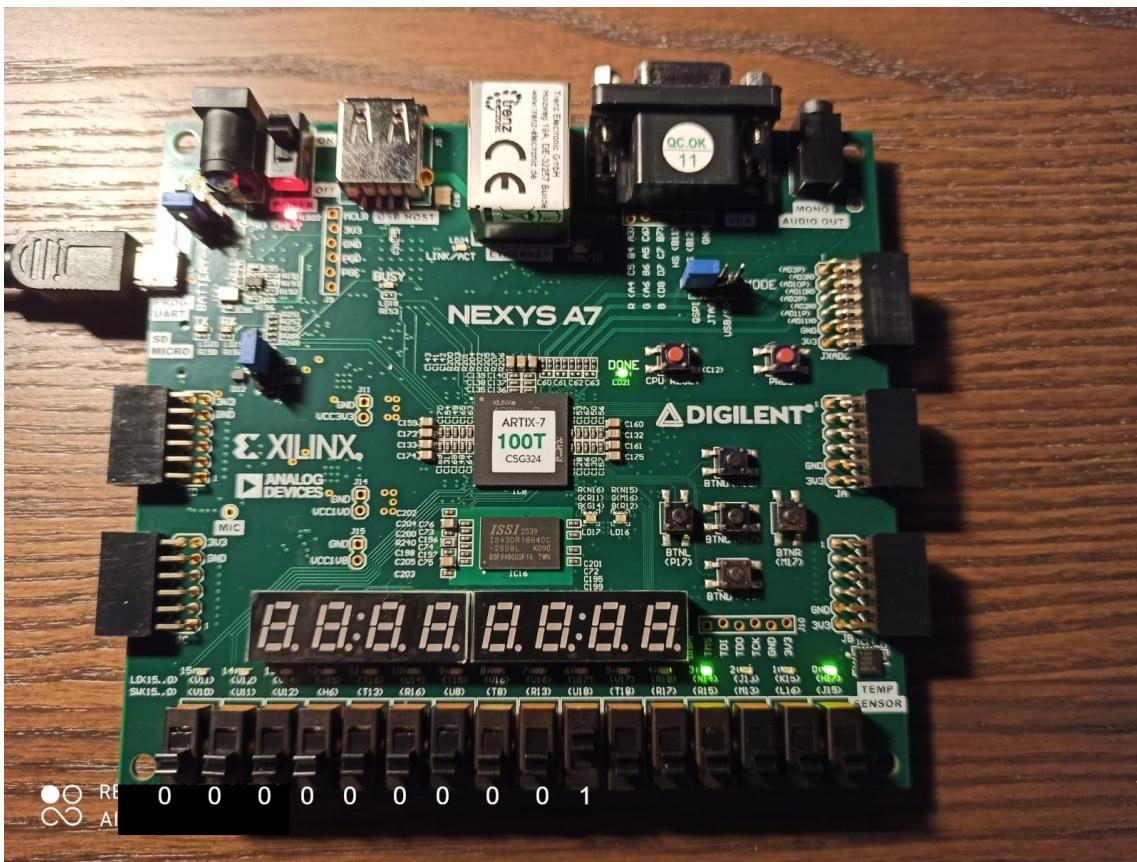


FIG. 2.5

2.1.4 Implementazione sulla board tramite il display

Per poter implementare l'utilizzo del display sulla board, è necessario modificare la composizione del progetto e aggiungere una nuova entità, denominata proprio *display*. Ad ogni numero in ingresso si associa la giusta configurazione degli anodi e dei catodi.

Nota: il seguente metodo di utilizzo del display va bene esclusivamente se si deve utilizzare una solo cifra. Se il progetto avesse preveduto l'utilizzo di più cifre, l'utilizzo del display sarebbe dovuto essere gestito diversamente.

Display.vhd

```

ENTITY display IS
  PORT (
    number : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    anodi : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    catodi : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END display;

ARCHITECTURE Behavioral OF display IS

BEGIN

  PROCESS (number)

    BEGIN

      CASE number IS
        WHEN "0000" =>
          catodi <= "11000000"; --0
          anodi <= "11111110";
        WHEN "0001" =>
          catodi <= "11111001"; --1
          anodi <= "11111110";
        WHEN "0010" =>
          catodi <= "10100100"; --2
          anodi <= "11111110";
        WHEN "0011" =>
          catodi <= "10110000"; --3
          anodi <= "11111110";
        WHEN "0100" =>
          catodi <= "10011001"; --4
          anodi <= "11111110";
        WHEN "0101" =>
          catodi <= "10010010"; --5
          anodi <= "11111110";
        WHEN "0110" =>
          catodi <= "10000010"; --6
          anodi <= "11111110";
        WHEN "0111" =>
          catodi <= "11111000"; --7
          anodi <= "11111110";
        WHEN "1000" =>
          catodi <= "10000000"; --8
    
```



```

        anodi <= "11111110";
WHEN "1001" =>
    catodi <= "10010000"; --9
    anodi <= "11111110";
WHEN "1010" =>
    catodi <= "10100000"; --a
    anodi <= "11111110";
WHEN "1011" =>
    catodi <= "10000011"; --b
    anodi <= "11111110";
WHEN "1100" =>
    catodi <= "11000110"; --c
    anodi <= "11111110";
WHEN "1101" =>
    catodi <= "10100001"; --d
    anodi <= "11111110";
WHEN "1110" =>
    catodi <= "10000110"; --e
    anodi <= "11111110";
WHEN "1111" =>
    catodi <= "10001110"; --f
    anodi <= "11111110";
WHEN OTHERS =>
    catodi <= "11000000"; --x
    anodi <= "11111101";
END CASE;

END PROCESS;
END Behavioral;

```

Un ulteriore modifica che bisogna fare riguarda le porte di ingresso dell'entità *PriorityEncoder*, che devono essere aggiornate aggiungendo un vettore per i catodi e uno per gli anodi.

```

ENTITY PriorityEncoder IS
    PORT (
        X : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
        CAT : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        AN : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END PriorityEncoder;

```

Ovviamente, l'entità display deve essere aggiunta come componente all'interno dell'**ARCHITECTURE** del Priority Encoder, con le seguenti modifiche da apportare al codice VHDL prima mostrato:

```

SIGNAL u : STD_LOGIC_VECTOR(9 DOWNTO 0);
SIGNAL uu : STD_LOGIC_VECTOR(3 DOWNTO 0);

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

```



```

    X => u,
    Y => uu
);

D : display
PORT MAP(
    number => uu,
    anodi => AN,
    catodi => CAT
);

```

Infine, bisogna modificare il file *constraints* commentando le righe inerenti all'utilizzo dei led e decommentando ed associando correttamente quelle relative al display.

##7 segment display

```

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

```



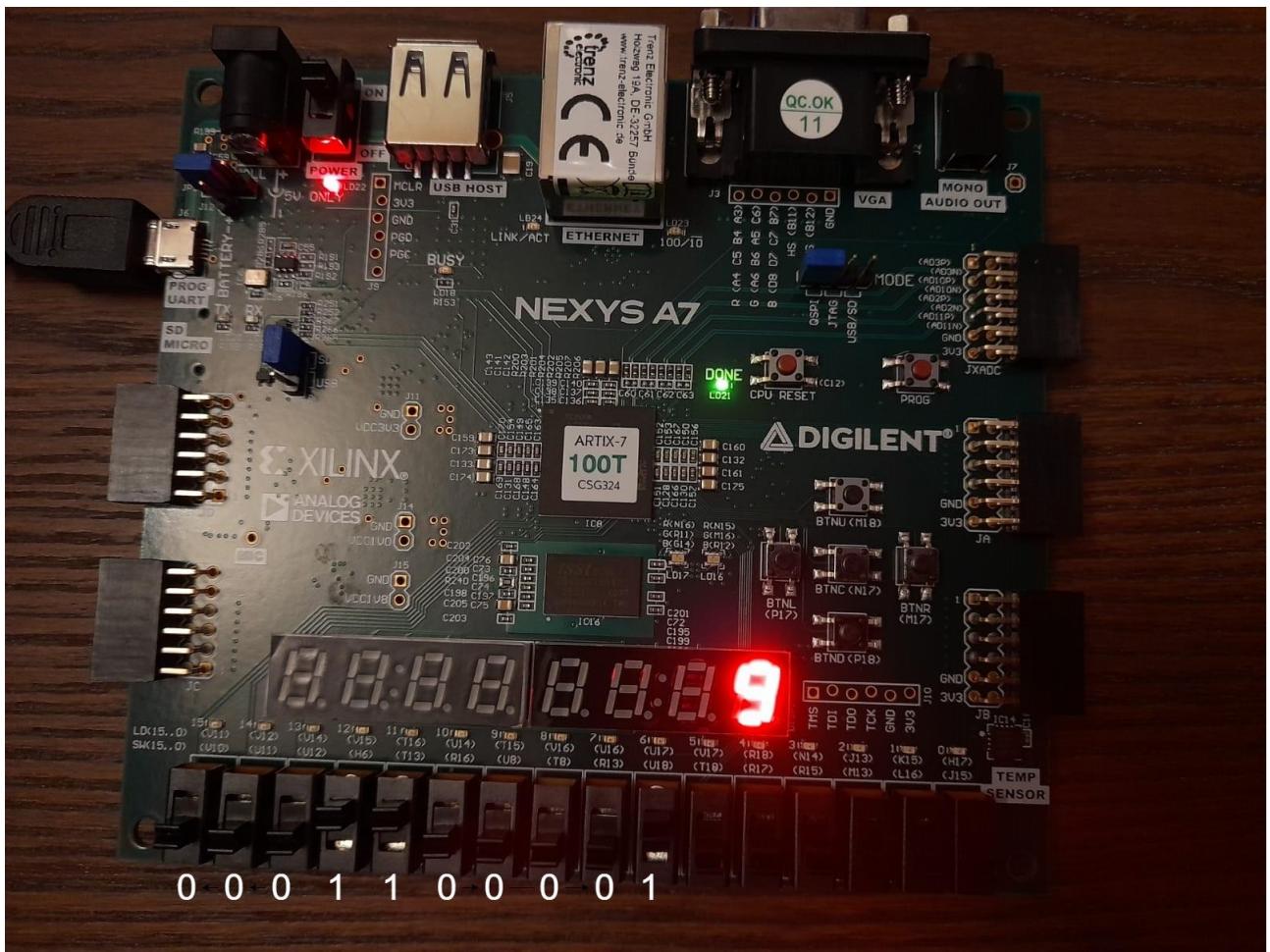


FIG. 2.6

Riportiamo un esempio nella figura 2.6 dell'utilizzo del display, in cui è possibile vedere che il decimo switch è alto; infatti, il display visualizza il numero 9, e continua a visualizzare 9 anche se sono alti altri switch di peso inferiore; ciò significa che l'arbitro funziona correttamente.

3 Riconoscitore a 2 modalità

Si realizza un Riconoscitore di sequenza a 2 modalità seguendo le specifiche richieste dal cliente:

- **3.1** - Progettare, implementare in VHDL e testare mediante **simulazione** una macchina in grado di riconoscere la sequenza **1001**. La macchina prende in ingresso un segnale binario i che rappresenta il dato, un segnale A di temporizzazione e un segnale M di modo, che ne disciplina il funzionamento, e fornisce un'uscita Y alta quando la sequenza viene riconosciuta. In particolare,
 - se **$M=0$** , la macchina valuta i bit seriali in ingresso a gruppi di 4,
 - se **$M=1$** , la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta.
- **3.2** - Sintetizzare e implementare su board la rete sviluppata al punto precedente, utilizzando uno switch $S1$ per codificare l'input i e uno switch $S2$ per codificare il modo M , in combinazione con due pulsanti $B1$ e $B2$ utilizzati rispettivamente per acquisire l'input da $S1$ e $S2$ in sincronismo con il segnale di temporizzazione A , che deve essere ottenuto a partire dal clock del board. Infine, l'uscita Y può essere codificata utilizzando un led.

3.1 Progettazione del Riconoscitore di sequenza a 2 modalità

Per la realizzazione della macchina, si utilizza **un approccio comportamentale** dove viene descritto il comportamento della macchina come definito dai relativi automi. Le specifiche del progetto richiedono la valutazione dei bit in "2 modi" determinato dal segnale M di modo fornendo un'uscita Y alta quando la sequenza viene riconosciuta.

Dalle indicazioni delle specifiche, la sequenza da riconoscere è la **1001**, pertanto si costruiscono le due macchine sequenziali:

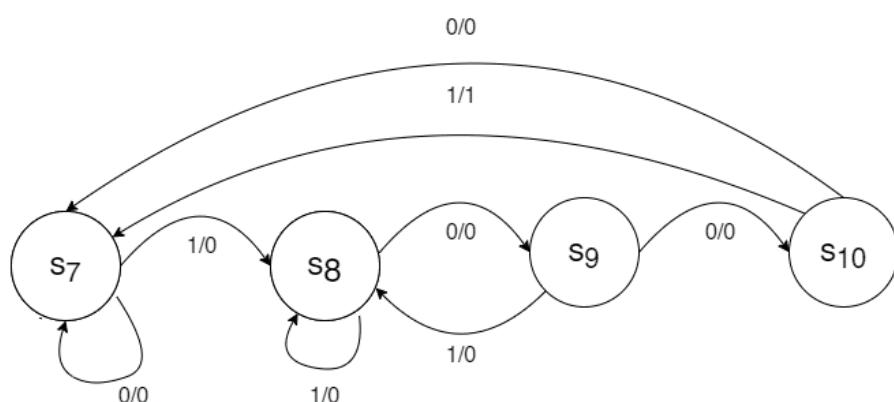


FIG. 3.1 - Automa che valuta i bit seriali in ingresso uno alla volta

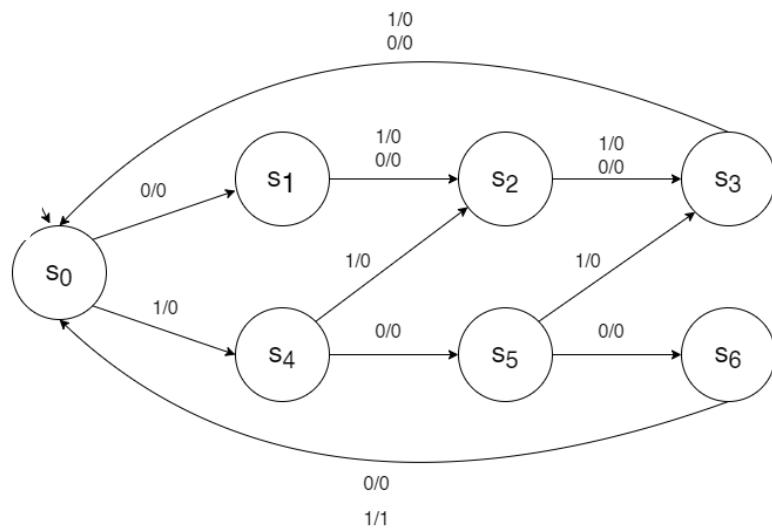


FIG. 3.2 - Automa che valuta i bit seriali in ingresso a gruppi di 4 bit.

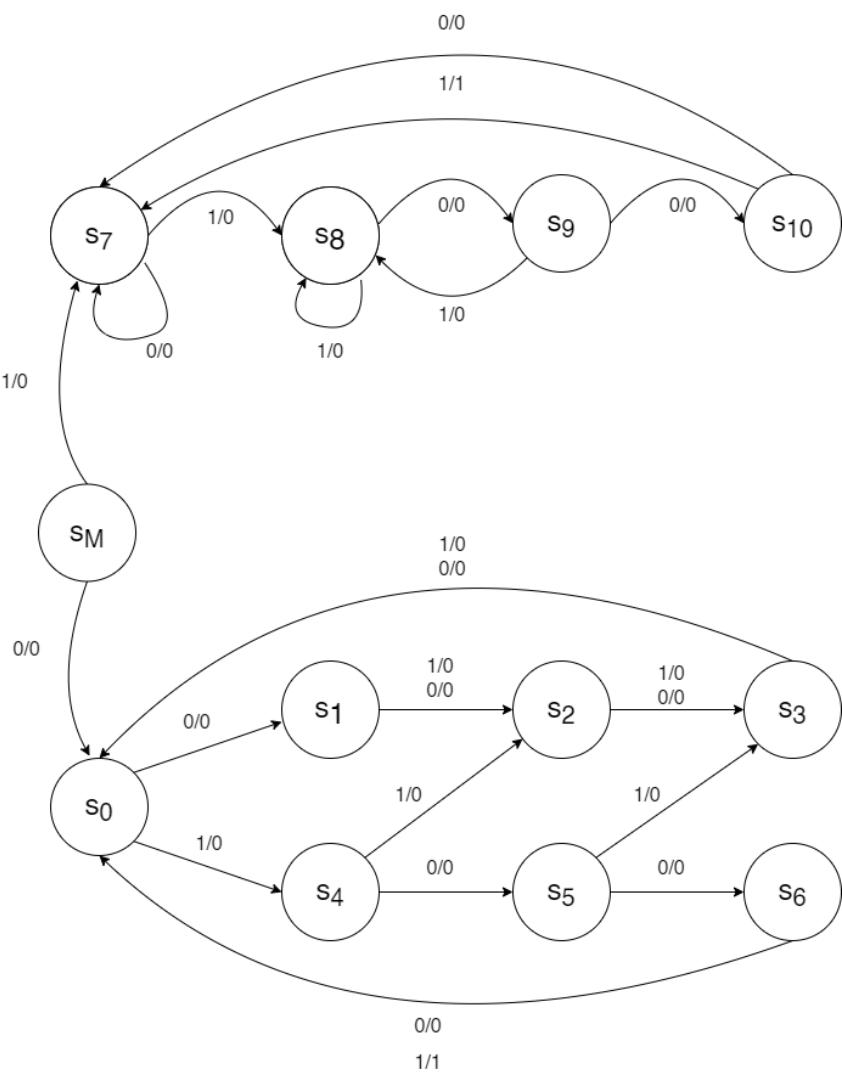


FIG. 3.3 - Automa con modo

Dalla **FIG 3.3** si evince dunque il funzionamento del riconoscitore. Partendo dallo stato iniziale **SM**, lo stato che determina il modo dell'automa, se $M = 1$, si valuta la sequenza **un bit alla volta**, se $M = 0$ si valuta la sequenza **a gruppi di 4 bit**.

Dalle **FIGURE** si nota inoltre che è stato utilizzato il modello di **Mealy** dove la variazione dell'uscita a fronte di ingresso e di stato è:

$$w: Q \times I \rightarrow U$$

3.1.1 Schematico del Riconoscitore di sequenza a 2 modalità

Il modello del seguente riconoscitore è diviso da una rete combinatoria (la parte del sistema che calcolerà le funzioni di transizione da uno stato all'altro) e l'elemento di memoria (la retroazione) utile per la gestione SINCRONA, tramite RST (Reset) di M (Modo) tempificato dal CLK (Clock).

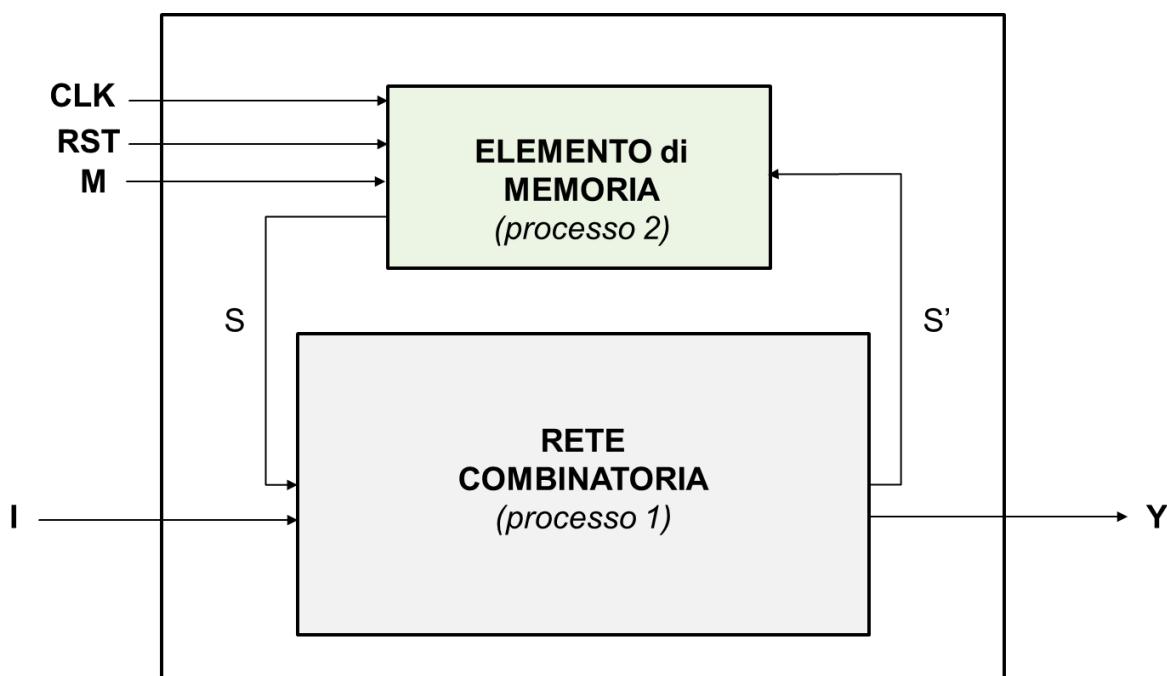


FIG. 3.4 – Schematico del riconoscitore di sequenza

3.1.2 Codice VHDL del Riconoscitore di sequenza a 2 modalità

Dallo schematico in **FIG 3.4**, si estrapolano i “process” da implementare in VHDL, nel caso in esame si sono utilizzati dunque due processi, uno per la parte combinatoria ed uno per la parte di memoria. L’implementazione è dunque, divisa in: Design Sources per l’implementazione dei processi e in Simulation Sources per il TestBench.

Si definisce **l’entità** composta da 4 porte in ingresso (I , RST , M , CLK) e l’uscita (Y).

Riconoscitore_Mealy_2_Modi.vhd

```
-- L'entità
entity Riconoscitore_Mealy_2_Modi_01 is
    port( i: in std_logic;
          RST, CLK: in std_logic;
          M: in std_logic;
          Y: out std_logic
        );
end Riconoscitore_Mealy_2_Modi_01;
```

Nell'**architecture** si enumerano i 10 diversi stati che compongono entrambi gli automi e si pone come stato iniziale, come già detto, lo stato SM.

```
-- architettura
architecture Behavioral of Riconoscitore_Mealy_2_Modi_01 is

-- enumerazione stati
type stato is (SM, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10);

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

Il processo **combinatorio** sarà così costituito:

```
begin
stato_uscita: process(stato_corrente, i)
begin

    -- se M = 0 : Riconoscitore di Sequenza con valutazione di
    sequenza per gruppi di 4 bit
    -- se M = 1 : Riconoscitore di Sequenza con valutazione di
    sequenza ad ogni singolo bit

        case stato_corrente is
            when S0 =>
                if(i = '0' ) then
                    stato_prossimo <= S1;
                    Y <= '0';
                else
                    stato_prossimo <= S4;
                    Y <= '0';
                end if;

            when S1 =>
                if( i = '0' or i = '1' ) then
                    stato_prossimo <= S2;
                    Y <= '0';
                end if;

            when S2 =>
                if( i = '0' or i = '1' ) then
                    stato_prossimo <= S3;
```



```

        Y <= '0';
    end if;

when S3 =>
    if( i = '0' or i = '1' ) then
        stato_prossimo <= S0;
        Y <= '0';
    end if;

when S4 =>
    if( i = '0' ) then
        stato_prossimo <= S5;
        Y <= '0';
    else
        stato_prossimo <= S2;
        Y <= '0';
    end if;

when S5 =>
    if( i = '0' ) then
        stato_prossimo <= S6;
        Y <= '0';
    else
        stato_prossimo <= S3;
        Y <= '0';
    end if;

when S6 =>
    if( i = '0' ) then
        stato_prossimo <= S0;
        Y <= '0';
    else
        stato_prossimo <= S0;
        Y <= '1';
    end if;

when S7 =>
    if( i = '0' ) then
        stato_prossimo <= S7;
        Y <= '0';
    else
        stato_prossimo <= S8;
        Y <= '0';
    end if;

when S8 =>
    if( i = '0' ) then
        stato_prossimo <= S9;
        Y <= '0';
    else
        stato_prossimo <= S8;
        Y <= '0';
    end if;

when S9 =>
    if( i = '0' ) then

```

```

        stato_prossimo <= S10;
        Y <= '0';
    else
        stato_prossimo <= S8;
        Y <= '0';
    end if;

    when S10 =>
        if( i = '0' ) then
            stato_prossimo <= S7;
            Y <= '0';
        else
            stato_prossimo <= S7;
            Y <= '1';
        end if;

    when others =>
        stato_prossimo <= S7;
        Y <= '0';
    end case;
end process;

```

Si è usato il costrutto **CASE-WHEN** per definire cosa fa la macchina sequenziale in ogni stato. Ogni valore dell'ingresso 'i' fa evolvere la macchina verso un nuovo stato specificando dunque una determinata uscita Y.

Il processo **di memorizzazione** sarà:

```

mem: process (CLK)
begin
    if(CLK'event and CLK = '1') then
        if(RST = '1') then
            stato_corrente <= SM;
            if(M = '0') then
                stato_corrente <= S0;
            else
                stato_corrente <= S7;
            end if;
        else
            stato_corrente <= stato_prossimo;
        end if;
    end if;
end process;

end Behavioral;

```

L'istruzione consente di cambiare la modalità del riconoscitore soltanto quando il **RST** è abilitato a 1. Se M = 0, si imposta lo stato corrente del riconoscitore a 4 bit, se M = 1, si imposta lo stato corrente del riconoscitore a 1 bit.

L'assegnazione **stato_corrente <= stato_prossimo;** consente di retroazionare lo stato prossimo nello stato corrente.

Il TB ci permette di simulare il funzionamento della macchina in particolare si può apprezzare il comportamento del Riconoscitore nei casi in cui M = 0 o 1 e come le abilitazioni del RST influiscano sul cambio di modalità della macchina.

```

entity Riconoscitore_Mealy_2_Modi_01_TB is
-- Port ();
end Riconoscitore_Mealy_2_Modi_01_TB;

architecture Behavioral of Riconoscitore_Mealy_2_Modi_01_TB is

COMPONENT Riconoscitore_Mealy_2_Modi_01_TB
PORT(
    i : IN std_logic;
    CLK,RST,M : IN std_logic;
    Y : OUT std_logic
);
END COMPONENT;

-- Inputs
signal i : std_logic := '0';
signal CLK : std_logic := '0';
signal RST : std_logic := '0';
signal M : std_logic := '1';

--Outputs
signal Y : std_logic;

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

begin

    -- Instantiate the Unit Under Test (UUT)
    -- Qui si specifica quale architecture simulare di quelle
    definite nel progetto corrente
    uut: entity work.Riconoscitore_Mealy_2_Modi_01(Behavioral) port map(
        i => i,
        CLK => CLK,
        RST => RST,
        M => M,
        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;

        -- start da 200 (100 + CLK_period*10)
        wait for CLK_period*10;
    end process;

```



```
-- insert stimulus here
-- sequenza di bit in ingresso 0010010000101001

    i<='0';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='1';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='1';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='1';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;

--C'è il cambio di M e viene abilitato il reset (RST<='1')

M<='0';
RST<='1';
wait for 10 ns;
RST<='0';

    i<='0';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='1';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='1';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='1';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='1';
    wait for 10 ns;

-- C'è il cambio di M e viene abilitato il reset (RST<='1')

M<='1';
RST<='1';
wait for 10 ns;
RST<='0';

    i<='0';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='1';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
    i<='1';
    wait for 10 ns;
    i<='0';
    wait for 10 ns;
```

```

        i<='0';
        wait for 10 ns;
        i<'1';

        wait;
    end process;

end Behavioral;

```

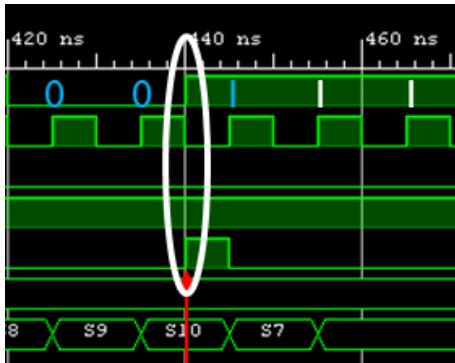
3.1.3 La simulazione del Riconoscitore di sequenza a 2 modalità

Seguendo il TestBench, la simulazione sarà:

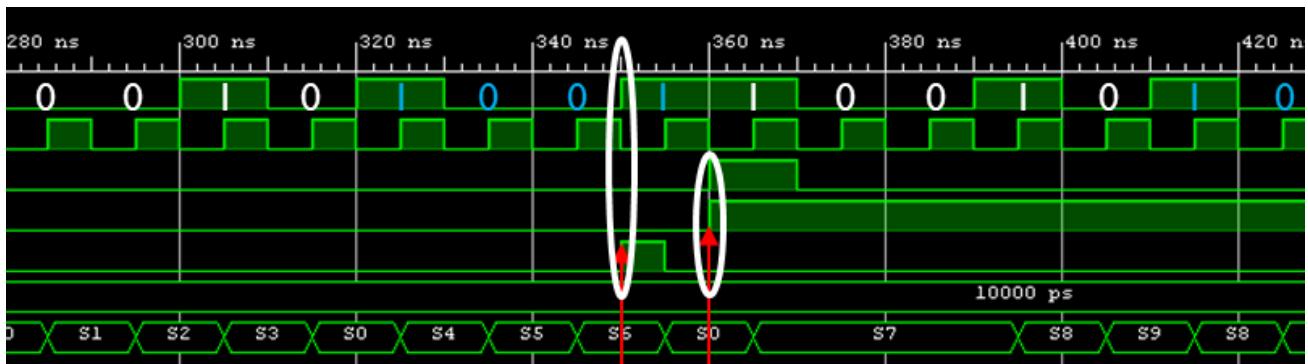


A **250ns** si ha la transizione di Y: 0 → 1 (M=1)

A **270ns** si ha la transizione di M: 1 → 0 e RST: 0 → 1
(RST torna a 0 dopo 10ns)



A **440ns** si ha la transizione di Y: 0 → 1 (M=1)



A **350ns** si ha la transizione di Y: 0 → 1 (M=0)

A **360ns** si ha la transizione di M: 0 → 1 e
RST: 0 → 1
(RST torna a 0 dopo 10ns)

3.2 Implementazione sulla board

3.2.1 Il ButtonDebouncer

Per l'implementazione sulla board, vedendo le specifiche, si prevede l'utilizzo di un led per la visualizzazione del risultato, di uno switch per il valore di M, uno switch per l'input e due bottoni per dire all'automa quando prelevare gli ingressi e quando prelevare il modo.

Per l'utilizzo della board, è necessario utilizzare un dispositivo aggiuntivo, il ButtonDebouncer per eliminare il rumore presente ai lati dei segnale effettivo del bottone. Il bottone è gestito considerando 2 stati: **PRESSED** e **NOT_PRESSED**. Quando il bottone non è premuto si trova nello stato **NOT_PRESSED**, non appena viene premuto, esso transita nello stato **PRESSED** e ci rimane finché il bottone è premuto e un certo contatore non arriva ad un dato valore M. Questo valore M non è

presente nelle documentazioni ufficiali, ma deve essere ricavato empiricamente facendo delle prove sul funzionamento. Il segnale di uscita dal ButtonDebouncer sarà un segnale del bottone pulito, cioè sarà quasi impulsivo.

ButtonDebouncer.vhd

```

ENTITY ButtonDebouncer IS
    GENERIC (
        CLK_period : INTEGER := 10; -- periodo del clock della board
10 nanosecondi
        btn_noise_time : INTEGER := 500000000 --intervallo di tempo in
cui si ha l'oscillazione del bottone
        --assumo che duri 500ms=500000microsec=500000000ns
    );
    PORT (
        RST : IN STD_LOGIC;
        CLK : IN STD_LOGIC;
        BTN : IN STD_LOGIC;
        CLEARED_BTN : OUT STD_LOGIC);
END ButtonDebouncer;

ARCHITECTURE Behavioral OF ButtonDebouncer IS

    SIGNAL BTN_state : stato := NOT_PRESSED;

    CONSTANT max_count : INTEGER := btn_noise_time/CLK_period; --
500000000/10= conto 50000000 colpi di clock

BEGIN

    deb : PROCESS (CLK)
        VARIABLE count : INTEGER := 0;

    BEGIN
        IF rising_edge(CLK) THEN

            IF (RST = '1') THEN
                BTN_state <= NOT_PRESSED;
                CLEARED_BTN <= '0';
            ELSE
                CASE BTN_state IS
                    WHEN NOT_PRESSED =>
                        CLEARED_BTN <= '0';
                        IF (BTN = '1') THEN
                            BTN_state <= PRESSED;
                        ELSE
                            BTN_state <= NOT_PRESSED;
                        END IF;
                    WHEN PRESSED =>
                        IF (count = max_count - 1) THEN
                            count := 0;
                            CLEARED_BTN <= '1';
                            BTN_state <= NOT_PRESSED;
                        ELSE
                            count := count + 1;
                        END IF;
                END CASE;
            END IF;
        END IF;
    END BEGIN;

```



```

        BTN_state <= PRESSED;
    END IF;
    WHEN OTHERS =>
        BTN_state <= NOT_PRESSED;
    END CASE;
END IF;
END IF;
END PROCESS;
END Behavioral;

```

3.2.2 Top Module

Dopo il ButtonDebouncer, bisogna creare una entità che contenga al suo interno sia il riconoscitore, sia il Debouncer. Questa entità la chiamiamo Riconoscitore_selector_onboard e il codice è il seguente:

Riconoscitore_selector_onboard.vhd

```

ENTITY Riconoscitore_selector_onboard IS
PORT (
    clock_in : IN STD_LOGIC;
    reset_in : IN STD_LOGIC; --bottone che abilita il cambio di
modalità del riconoscitore
    I_strobe_in : IN STD_LOGIC; --bottone che abilita la lettura
dell'input I dallo switch
    Input : IN STD_LOGIC; --input della sequenza di bit (SWITCH)
    Mode : IN STD_LOGIC; --modalità del riconoscitore (SWITCH)
    Output : OUT STD_LOGIC --output del riconoscitore
);
END Riconoscitore_selector_onboard;

```

ARCHITECTURE Structural **OF** Riconoscitore_selector_onboard **IS**

```

COMPONENT ButtonDebouncer
GENERIC (
    CLK_period : INTEGER := 10; -- periodo del clock della
board in nanosecondi
    btn_noise_time : INTEGER := 500000000 --intervallo di
tempo in cui si ha l'oscillazione del bottone
    --assumo che duri 500ms=500000microsec=500000000ns
);
PORT (
    RST : IN STD_LOGIC;
    CLK : IN STD_LOGIC;
    BTN : IN STD_LOGIC;
    CLEARED_BTN : OUT STD_LOGIC);
END COMPONENT;
COMPONENT Riconoscitore_Mealy_2_Modi_01 IS
PORT (
    i : IN STD_LOGIC;
    RST, CLK : IN STD_LOGIC;
    READ_I : IN STD_LOGIC;

```



```

        M : IN STD_LOGIC;
        Y : OUT STD_LOGIC
    );
END COMPONENT;
SIGNAL reset_n, read_strobeI : STD_LOGIC;

BEGIN

    reset_n <= NOT reset_in; --visto che utilizzo il bottone CPU_reset
della board, che è attivo-basso,
--devo convertire il segnale di reset

debouncerI : ButtonDebouncer GENERIC MAP(
    CLK_period => 10, -- periodo del clock della board pari a 10ns
    btn_noise_time => 500000000 --intervallo di tempo in cui si ha
l'oscillazione del bottone
    --assumo che duri 500ms=500000microsec=500000000ns
)
PORT MAP(
    RST => reset_n,
    CLK => clock_in,
    BTN => I_strobe_in,
    CLEARED_BTN => read_strobeI
);
riconoscitore : Riconoscitore_Mealy_2_Modi_01
PORT MAP(
    i => Input,
    RST => reset_n,
    CLK => clock_in,
    READ_I => read_strobeI,
    M => Mode,
    Y => Output
);
END Structural;

```

3.2.3 Implementazione sulla board

Per caricare l'automa sulla board, bisogna modificare opportunamente il solito file *constraints* della propria board. In questo esercizio, si è deciso di usare i primi due switch sulla destra per implementare rispettare l'input e il modo M e si è usato il primo led sulla destra per visualizzare l'output; successivamente, come bottoni, si è deciso di utilizzare il bottone CPU_reset per il reset e il bottone M18 per la selezione dell'input (attenzione: il bottone CPU_reset è 0 attivo, ciò vuol dire che funziona in maniera inversa rispetto agli altri bottoni, proprio per questo nel codice si è dovuto negare il segnale relativo a questo bottone tramite il comando `reset_n <= not reset_in;`).

```

##Switches
set_property -dict { PACKAGE_PIN J15    IO_STANDARD LVCMOS33 } [get_ports {
Input }];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16    IO_STANDARD LVCMOS33 } [get_ports {
Mode }];
#IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]

## LEDs
set_property -dict { PACKAGE_PIN H17    IO_STANDARD LVCMOS33 } [get_ports {
Output }];
#IO_L18P_T2_A24_15 Sch=led[0]

```



##Buttons

```
set_property -dict { PACKAGE_PIN C12      IOSTANDARD LVCMOS33 } [get_ports { reset_in }];
#IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn
#set_property -dict { PACKAGE_PIN N17      IOSTANDARD LVCMOS33 } [get_ports { BTNC }];
#IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18      IOSTANDARD LVCMOS33 } [get_ports { I_strobe_in }];
#IO_L4N_T0_D05_14 Sch=btnu
```

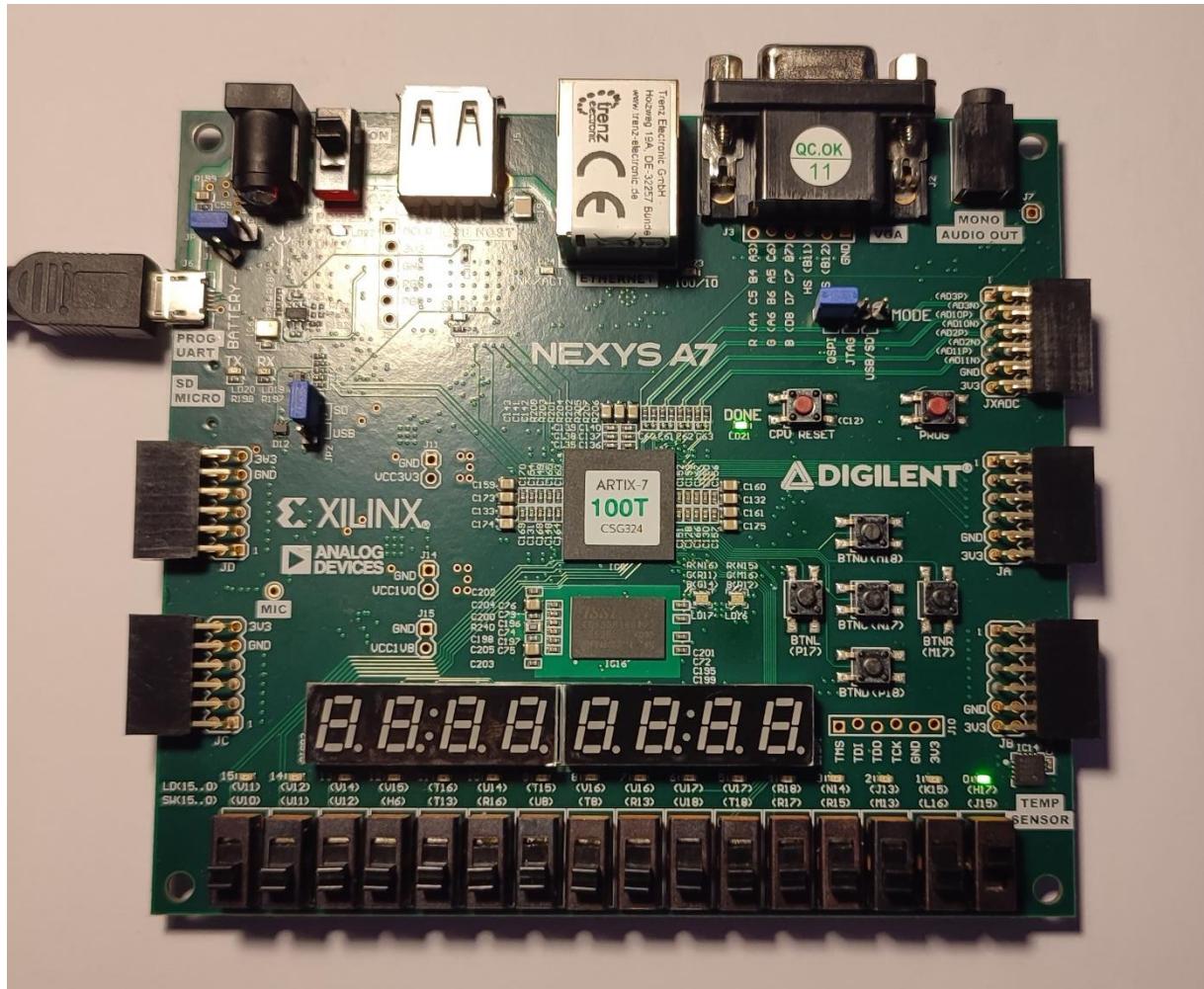


FIG 3.5: Verifica su board con M=0 ed inserendo la sequenza 1001

4 Shift register

Si realizza uno Shift register seguendo le specifiche richieste dal cliente:

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

Nota: il numero di bit del registro X e i valori che può assumere il parametro Y possono essere scelti dallo studente (ad es. $X=8$ e $Y=\{1,2\}$).

4.1 Shift register in generale

Uno shift register (o registro a scorrimento) con configurazione serie-serie è un registro che prende in ingresso un bit alla volta, lo salva nella prima locazione e contemporaneamente fa scorrere tutti gli altri bit al suo interno e dà in uscita il bit che era presente nell'ultima locazione. Tendenzialmente lo shift register ad n bit è formato da n flip flop di tipo D. Esistono però altre configurazioni dello shift register in base al tipo di ingresso e di uscita che si ha (ovvero siano essi seriali o paralleli). Per la realizzazione, si è deciso di usare una configurazione serie-serie e il numero di bit del registro è stato configurato a $n=8$.

Particolarità dell'implementazione di sudetto shift register è la possibilità di poter shiftare in entrambe le direzioni e di poter shiftare anche di più posizioni (sempre in presenza però di un solo bit di input e dando in uscita un solo bit).

Se per lo shift in entrambe le direzioni non c'è nessuna ambiguità, stessa cosa non si può dire per gli shift multipli, infatti sorge il quesito di cosa fare con i bit uscenti dagli shift successivi al primo, dato che l'output deve essere comunque uno solo e deve corrispondere al bit uscente dal primo shift. Si è deciso di fare in modo che i bit uscenti dagli shift successivi al primo vengano reinseriti all'interno dello shift register in modo **quasi-circolare**; infatti, non vengono inseriti a partire dalla prima locazione ma a partire dalla seconda, questo perché si è pensato che eventualmente, per qualche applicazione particolare, si voglia che anche con lo shift register configurato con più shift, rimanga in prima posizione il bit appena inserito. Un disegno esplicativo del comportamento dello shift register è riportato in Figura 4.1.

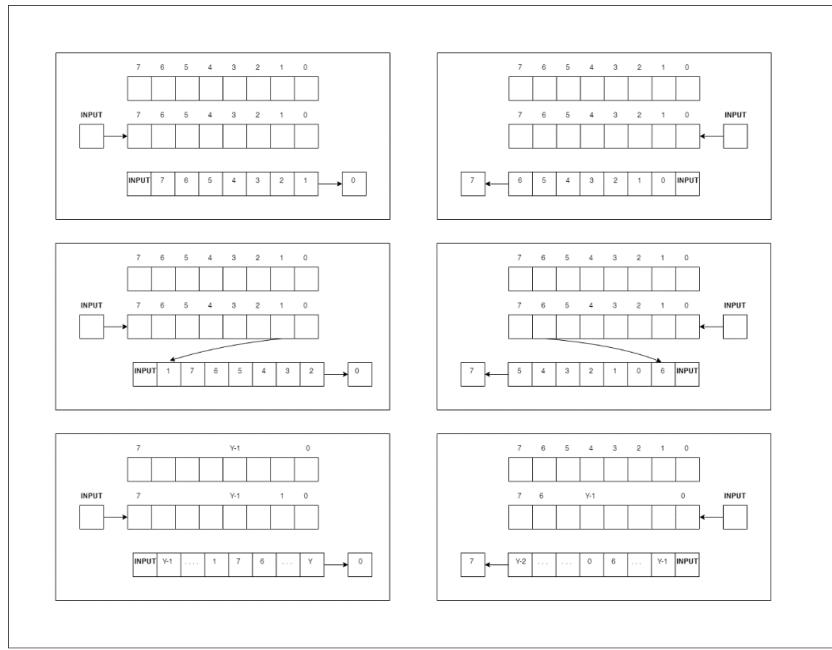


FIG. 4.1

4.2 Shift register comportamentale

Per la progettazione dello Shift register con approccio comportamentale, abbiamo utilizzato una struttura che ha in input i bit:

- clk => il clock del sistema
- input => il bit da dover inserire
- sel => per la selezione della direzione in cui shiftare
- reset => per resettare lo shift register.

Inoltre, è stato utilizzato il costrutto generic, tramite il quale è possibile impostare il numero di shift da voler effettuare, modificando proprio il valore y, in particolare:

- quando Y è 1, il registro viene shiftato di 1, e in base al bit di selezione viene determinata la posizione del valore che va in output e la posizione in cui verrà inserito il valore in input
- quando Y è maggiore di 1, avremo che nel caso sel=0 gli N-Y bit a sinistra vengono shiftati di Y posizioni, mentre Y-1 bit a sinistra vengono inseriti dopo l'ultimo inserito

4.2.1 Codice VHDL

Shift_Register.vhd

```
ENTITY Shift_Register IS
  GENERIC (y : POSITIVE := 2);
  PORT (
    clk, input, sel, reset : IN STD_LOGIC;
    output : OUT STD_LOGIC);
END Shift_Register;
```



```
ARCHITECTURE Behavioral OF Shift_Register IS

  SIGNAL tmp : STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN
  PROCESS (clk)
  BEGIN
    IF (clk'event AND clk = '1') THEN
      IF (reset = '1') THEN
        tmp <= "00000000";
      ELSE
        IF (sel = '0') THEN --shifta a destra
          output <= tmp(0);
          IF (y = 1) THEN
            tmp <= input & tmp(7 DOWNTO 1);
          ELSE
            tmp <= input & tmp(y - 1 DOWNTO 1) &
tmp(7 DOWNTO y);
          END IF;

        ELSE --shifta a sinistra
          output <= tmp(7);
          IF (y = 1) THEN
            tmp <= tmp(6 DOWNTO 0) & input;
          ELSE
            tmp <= tmp(7 - y DOWNTO 0) & tmp(6
DOWNTO 8 - y) & input;
          END IF;
        END IF;
      END IF;
    END PROCESS;

  END Behavioral;
```

4.2.2 TestBench

Shift_Register_TB.vhd

```
ENTITY Shift_Register_TB IS

END Shift_Register_TB;

ARCHITECTURE Behavioral OF Shift_Register_TB IS

  COMPONENT Shift_Register_TB
    GENERIC (y : POSITIVE := 2);
    PORT (
      clk, input, sel, reset : IN STD_LOGIC;
      output : OUT STD_LOGIC);
  END COMPONENT;
  --input
  SIGNAL input : STD_LOGIC := '0';
```



```
SIGNAL reset : STD_LOGIC := '0';
SIGNAL clk : STD_LOGIC := '0';
SIGNAL sel : STD_LOGIC := '0';

--output
SIGNAL output : STD_LOGIC := '0';

--clock period definition
CONSTANT clk_period : TIME := 10ns;

BEGIN

uut : ENTITY work.Shift_Register(Behavioral)
  GENERIC MAP(y => 2)
  PORT MAP(
    input => input,
    reset => reset,
    clk => clk,
    sel => sel,
    output => output
  );
-- 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
PROCESS
BEGIN
  -- hold reset state for 100 ns.
  WAIT FOR 100 ns;

  -- start da 200 (100 + CLK_period*10)
  WAIT FOR CLK_period * 10;

  -- insert stimulus here

  input <= '0';
  WAIT FOR 10 ns;
  input <= '0';
  WAIT FOR 10 ns;
  input <= '1';
  WAIT FOR 10 ns;
  input <= '0';
  WAIT FOR 10 ns;
  input <= '0';
  WAIT FOR 10 ns;
  input <= '1';
  WAIT FOR 10 ns;
  input <= '0';
  WAIT FOR 10 ns;
  input <= '0';
  WAIT FOR 10 ns;
  input <= '0';
  WAIT FOR 10 ns;
```



```

    input <= '0';
WAIT FOR 10 ns;
    input <= '1';
WAIT FOR 10 ns;
    input <= '0';
WAIT FOR 10 ns;
    input <= '1';
WAIT FOR 10 ns;
    input <= '0';
WAIT FOR 10 ns;
    input <= '0';
WAIT FOR 10 ns;
    input <= '1';
WAIT FOR 10 ns;

WAIT;
END PROCESS;

END Behavioral;

```

4.2.3 La simulazione

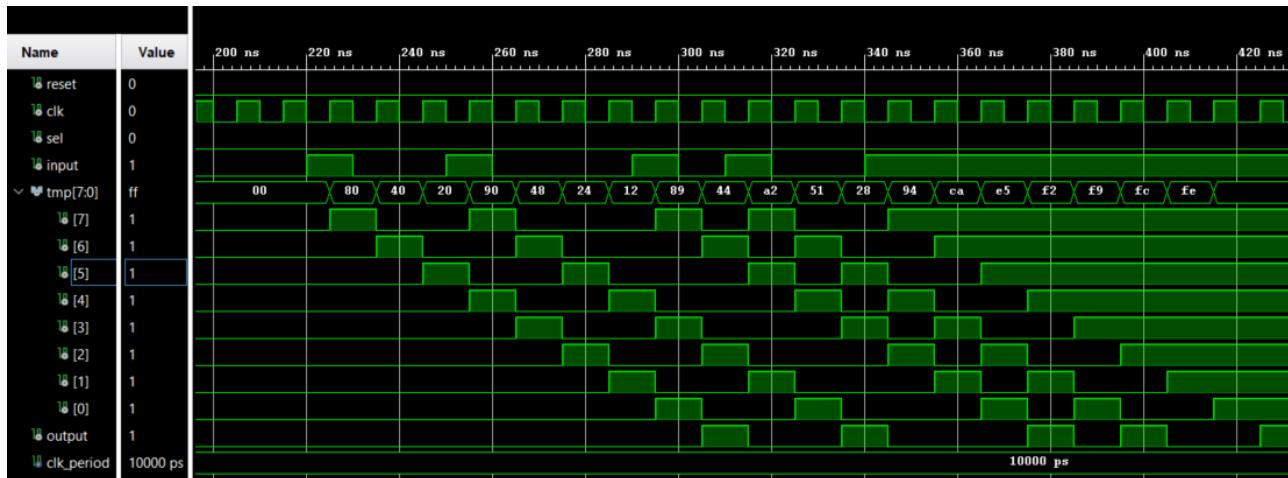


FIG. 4.2

4.3 Shift register strutturale

Per la progettazione dello Shift register con approccio strutturale abbiamo utilizzato i seguenti componenti:

- Multiplexer 8:1
- Flip Flop D

In particolar modo, lo shift register non è composto semplicemente da 8 flipflopD, ma da 8 componenti chiamati D_and_Mux, composti ognuno da un flipflopD e un Mux 8:1.

Per l'implementazione di tipo strutturale, non è stato possibile rendere parametrizzabile il numero di shift da voler compiere, quindi si è deciso di limitare la scelta ad un singolo shift (cioè funzionamento normale) e un doppio shift.

4.3.1 Codice VHDL

Vediamo più nel dettaglio i singoli componenti.

Flip_Flop_D.vhd

```
ENTITY Flip_Flop_D IS
  PORT (
    D : IN STD_LOGIC;
    Clock : IN STD_LOGIC;
    Reset_N : IN STD_LOGIC;
    Q : OUT STD_LOGIC);
END Flip_Flop_D;

ARCHITECTURE Behavioral OF Flip_Flop_D IS

BEGIN

  PROCESS (Clock)
  BEGIN
    IF (Clock'EVENT AND Clock = '1') THEN
      IF (Reset_N = '0') THEN
        Q <= '0';
      ELSE
        Q <= D;
      END IF;
    END IF;
  END PROCESS;

END Behavioral;
```

L'implementazione Flip_Flop_D è quella classica di un flipflopD con comportamento asincrono.

Mux_8_1.vhd

```
ENTITY Mux_8_1 IS
  PORT (
    w : IN STD_LOGIC_VECTOR (4 DOWNTO 0); -- w(0) = store, w(1) =
    s_right_1, w(2) = s_left_1, w(3) = s_right_2, w(4) = s_left_2
    m : IN STD_LOGIC_VECTOR (2 DOWNTO 0); --il primo bit di
    selezione m(2) sarebbe y, mentre gli altri due indicano il modo di shift
    f : OUT STD_LOGIC);
END Mux_8_1;

ARCHITECTURE Behavioral OF Mux_8_1 IS

BEGIN

  WITH m SELECT
    f <= w(1) WHEN "001",
    w(2) WHEN "010",
    w(3) WHEN "101",
    w(4) WHEN "110",
```



```
w(0) WHEN OTHERS;  
END Behavioral;
```

L'implementazione del Mux 8:1 utilizza solamente 5 delle 8 possibili configurazioni in ingresso, in particolar modo i bit di ingresso sono configurati nel seguente modo:

- w(0) = store, cioè semplicemente indica che il valore memorizzato nel flipflopD corrispondente deve rimanere inalterato
- w(1) = s_right_1, cioè un singolo shift a destra
- w(2) = s_left_1, cioè un singolo shift a sinistra
- w(3) = s_right_2, cioè un doppio shift a destra
- w(4) = s_left_2, cioè un doppio shift a sinistra.

I bit di selezione sono ovviamente 3 e, siccome non è possibile utilizzare tutte le 8 configurazioni dato che sono solo 5 i bit in ingresso al Mux, si è deciso di dare ai bit di selezione il seguente significato:

- m(2) indica y, ovvero m(2)=0 => y=1, cioè un solo shift, invece m(2)=1 => y=2, ovvero 2 shift
- m(1) indica lo shift a destra, quindi m(1)=1 si indica lo shift a destra
- m(0) indica lo shift a sinistra, quindi m(0)=1 si indica lo shift a sinistra

D_and_mux.vhd

```
LIBRARY IEEE;  
ENTITY D_and_Mux IS  
    PORT (  
        in_put : IN STD_LOGIC_VECTOR (4 DOWNTO 0);  
        Sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);  
        clk : IN STD_LOGIC;  
        clearN : IN STD_LOGIC;  
        Output : OUT STD_LOGIC);  
END D_and_Mux;  
  
ARCHITECTURE Structural OF D_and_Mux IS  
  
    COMPONENT Mux_8_1 IS  
        PORT (  
            w : IN STD_LOGIC_VECTOR (4 DOWNTO 0);  
            m : IN STD_LOGIC_VECTOR (2 DOWNTO 0); --il primo bit di  
selezione m(2) sarebbe y, mentre gli altri due indicano il modo di shift  
            f : OUT STD_LOGIC);  
    END COMPONENT;  
  
    COMPONENT Flip_Flop_D IS  
        PORT (  
            D : IN STD_LOGIC;  
            Clock : IN STD_LOGIC;  
            Reset_N : IN STD_LOGIC;  
            Q : OUT STD_LOGIC);  
    END COMPONENT;
```



```

SIGNAL mux_d : STD_LOGIC;

BEGIN

    mux : Mux_8_1
    PORT MAP (
        w => in_put,
        m => sel,
        f => mux_d
    );

    FFD : Flip_Flop_D
    PORT MAP (
        D => mux_d,
        Q => Output,
        Clock => clk,
        Reset_N => clearN
    );
END Structural;

```

Il flipflopD e il Mux vanno poi uniti a formare un unico componente, che chiamiamo D_and_Mux. Questo è il componente base che formerà lo shift register. Fondamentalmente, il Mux andrà a selezionare uno dei 5 ingressi per il flipflopD, dove ogni ingresso sta ad indicare una modalità di funzionamento. In base al valore dei bit di selezione, in uscita al Mux ci sarà un bit diverso (corrispondente al bit esatto per la modalità scelta) e quel bit andrà in ingresso al flipflopD e si memorizzerà al suo interno, fornendolo poi in uscita.

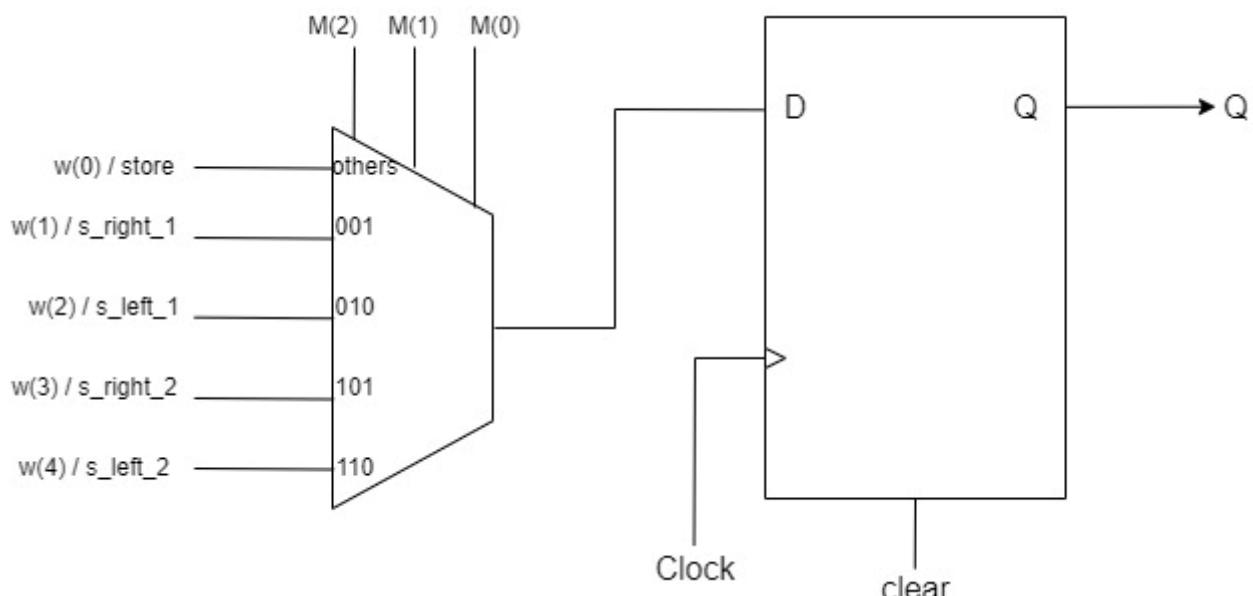


FIG. 4.3

Shift_register.vhd

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Shift_Register IS
    PORT (
        input : IN STD_LOGIC;
        outs : OUT STD_LOGIC;
        clk_1 : IN STD_LOGIC;
        sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        clr_1 : IN STD_LOGIC
    );
END Shift_Register;

ARCHITECTURE Structural OF Shift_Register IS

    COMPONENT D_and_Mux IS
        PORT (
            in_put : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
            Sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
            clk : IN STD_LOGIC;
            clearN : IN STD_LOGIC;
            Output : OUT STD_LOGIC);
    END COMPONENT;
    COMPONENT Flip_Flop_D IS
        PORT (
            D : IN STD_LOGIC;
            Clock : IN STD_LOGIC;
            Reset_N : IN STD_LOGIC;
            Q : OUT STD_LOGIC);
    END COMPONENT Flip_Flop_D;

    SIGNAL sh_left : STD_LOGIC_VECTOR (6 DOWNTO 0);
    SIGNAL sh_right : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL store : STD_LOGIC_VECTOR (7 DOWNTO 0);
    SIGNAL uscita : STD_LOGIC;
    --viene inserito questo segnale per inserire il valore di uscita
    --all'interno di un flip flop che lo tiene memorizzato.

BEGIN
    stage7 : D_and_Mux PORT MAP(
        in_put(0) => store(7),
        in_put(1) => input,
        in_put(2) => sh_left(6),
        in_put(3) => input,
        in_put(4) => sh_left(5),
        clk => clk_1,
        Sel => sel,
        clearN => clr_1,
        Output => sh_right(7));

    stage6 : D_and_Mux PORT MAP(
        in_put(0) => store(6),
        in_put(1) => sh_right(7),
```



```
in_put(2) => sh_left(5),
in_put(3) => sh_right(1),
in_put(4) => sh_left(4),
clk => clk_1,
Sel => sel,
clearN => clr_1,
Output => sh_right(6));

stage5 : D_and_Mux PORT MAP(
    in_put(0) => store(5),
    in_put(1) => sh_right(6),
    in_put(2) => sh_left(4),
    in_put(3) => sh_right(7),
    in_put(4) => sh_left(3),
    clk => clk_1,
    Sel => sel,
    clearN => clr_1,
    Output => sh_right(5));

stage4 : D_and_Mux PORT MAP(
    in_put(0) => store(4),
    in_put(1) => sh_right(5),
    in_put(2) => sh_left(3),
    in_put(3) => sh_right(6),
    in_put(4) => sh_left(2),
    clk => clk_1,
    Sel => sel,
    clearN => clr_1,
    Output => sh_right(4));

stage3 : D_and_Mux PORT MAP(
    in_put(0) => store(3),
    in_put(1) => sh_right(4),
    in_put(2) => sh_left(2),
    in_put(3) => sh_right(5),
    in_put(4) => sh_left(1),
    clk => clk_1,
    Sel => sel,
    clearN => clr_1,
    Output => sh_right(3));

stage2 : D_and_Mux PORT MAP(
    in_put(0) => store(2),
    in_put(1) => sh_right(3),
    in_put(2) => sh_left(1),
    in_put(3) => sh_right(4),
    in_put(4) => sh_left(0),
    clk => clk_1,
    Sel => sel,
    clearN => clr_1,
    Output => sh_right(2));

stage1 : D_and_Mux PORT MAP(
    in_put(0) => store(1),
    in_put(1) => sh_right(2),
    in_put(2) => sh_left(0),
```

```
    in_put(3) => sh_right(3),
    in_put(4) => sh_left(6),
    clk => clk_1,
    Sel => sel,
    clearN => clr_1,
    Output => sh_right(1));

stage0 : D_and_Mux PORT MAP(
    in_put(0) => store(0),
    in_put(1) => sh_right(1),
    in_put(2) => input,
    in_put(3) => sh_right(2),
    in_put(4) => input,
    clk => clk_1,
    Sel => sel,
    clearN => clr_1,
    Output => sh_right(0));

--si inserisce questo FLIPFlop per tenere memorizzato il valore di
uscita in modo che questo possa essere
--in output al successivo colpo di clock
ff_uscita : Flip_Flop_D PORT MAP(
    D => uscita,
    Clock => clk_1,
    Reset_N => clr_1,
    Q => outs);

store(7) <= sh_right(7);

store(6) <= sh_right(6);
sh_left(6) <= sh_right(6);

store(5) <= sh_right(5);
sh_left(5) <= sh_right(5);

store(5) <= sh_right(5);
sh_left(5) <= sh_right(5);

store(4) <= sh_right(4);
sh_left(4) <= sh_right(4);

store(3) <= sh_right(3);
sh_left(3) <= sh_right(3);

store(2) <= sh_right(2);
sh_left(2) <= sh_right(2);

store(1) <= sh_right(1);
sh_left(1) <= sh_right(1);

store(0) <= sh_right(0);
sh_left(0) <= sh_right(0);
uscita <= sh_right(0) WHEN sel = "001" OR sel = "101"
    ELSE sh_right(7) WHEN sel = "010" OR sel = "110"
    ELSE 'U';

END Structural;
```

Lo schematico risulta essere quello di Figura 4.4.

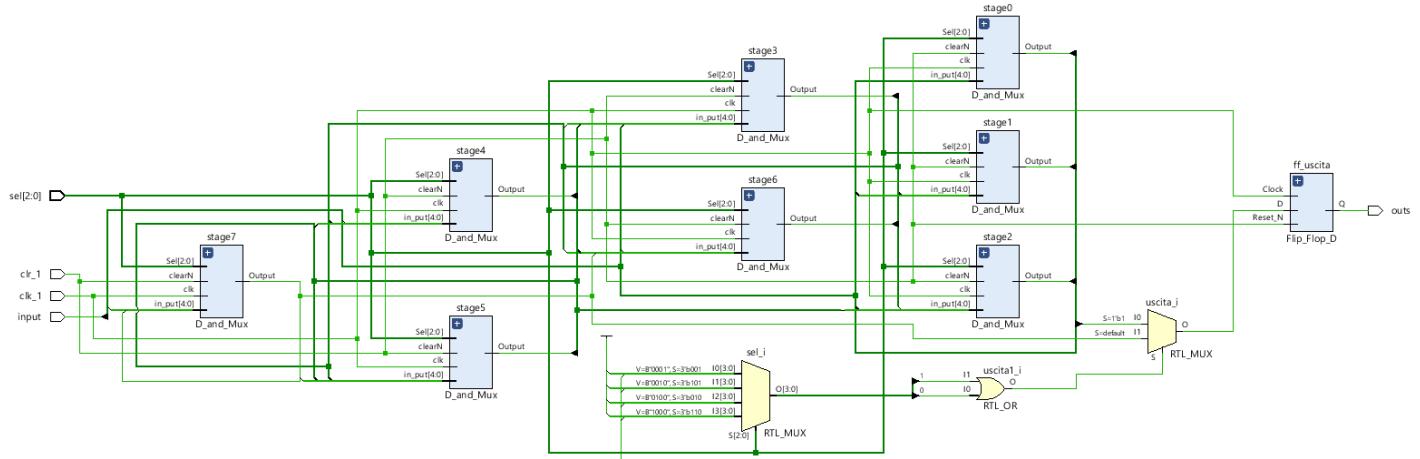


FIG. 4.4

Come si può notare, sono stati usati 8 componenti D_and_Mux opportunamente collegati per poter realizzare lo Shift Register. Per poter realizzare i collegamenti sono stati usati 3 vector signal, ovvero:

- store, per non effettuare nessun tipo di operazione
- sh_left, per gli shift a sinistra
- sh_right per gli shift a destra

Inoltre, è stato necessario aggiungere un ulteriore flipflopD per poter temporaneamente memorizzare il bit uscente, questo perché, se non fosse stato aggiunto, si avrebbe avuto il bit uscente sbagliato, questo perché l'uscita sarebbe stata direttamente collegata all'ultimo flipflopD; quindi, il bit in uscita dallo shift register non sarebbe stato quello in uscita dall'ultimo flipflop ma quello entrante.

4.3.2 TestBench

Shift_register_TB.vhd

```

ENTITY Shift_Register_TB IS
    -- Port ( );
END Shift_Register_TB;

ARCHITECTURE Behavioral OF Shift_Register_TB IS

    COMPONENT Shift_Register IS
        PORT (
            input : IN STD_LOGIC;
            outs : OUT STD_LOGIC;
            clk_1 : IN STD_LOGIC;
            sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
            clr_1 : IN STD_LOGIC
        );

```



```
        );
END COMPONENT;

--input
SIGNAL input : STD_LOGIC := '0';
SIGNAL clk_1 : STD_LOGIC := '0';
SIGNAL sel : STD_LOGIC_VECTOR(2 DOWNTO 0) := (OTHERS => '0');
SIGNAL clr_1 : STD_LOGIC := '0';

--output
SIGNAL output : STD_LOGIC;

--clock
CONSTANT CLK_period : TIME := 10 ns;

BEGIN

    uut : ENTITY work.Shift_Register(Structural) PORT MAP(
        input => input,
        outs => output,
        clk_1 => clk_1,
        sel => sel,
        clr_1 => clr_1
    );
    -- Clock process definitions
    CLK_process : PROCESS
    BEGIN
        CLK_1 <= '0';
        WAIT FOR CLK_period/2;
        CLK_1 <= '1';
        WAIT FOR CLK_period/2;
    END PROCESS;

    -- Stimulus process
    stim_proc : PROCESS
    BEGIN
        -- hold reset state for 100 ns.
        WAIT FOR 100 ns;

        -- start da 200 (100 + CLK_period*10)
        WAIT FOR CLK_period * 10;

        -- insert stimulus here

        --azzero tutti i registri
        clr_1 <= '0';
        WAIT FOR 30 ns;
        clr_1 <= '1';
        --test singolo shift a destra
        sel <= "001";

        input <= '1';
        WAIT FOR 10 ns;
        input <= '0';
        WAIT FOR 10 ns;
        input <= '0';
```

```
WAIT FOR 10 ns;
input <= '1';
WAIT FOR 10 ns;
input <= '0';

WAIT FOR 30 ns;

clr_1 <= '0';
WAIT FOR 30 ns;
clr_1 <= '1';

sel <= "010";

input <= '1';
WAIT FOR 10 ns;
input <= '0';
WAIT FOR 10 ns;
input <= '0';
WAIT FOR 10 ns;
input <= '1';
WAIT FOR 10 ns;
input <= '0';

WAIT FOR 30 ns;

clr_1 <= '0';
WAIT FOR 30 ns;
clr_1 <= '1';

sel <= "101";

input <= '1';
WAIT FOR 10 ns;
input <= '0';
WAIT FOR 10 ns;
input <= '0';
WAIT FOR 10 ns;
input <= '1';
WAIT FOR 10 ns;
input <= '0';

WAIT FOR 30 ns;

clr_1 <= '0';
WAIT FOR 30 ns;
clr_1 <= '1';

sel <= "110";

input <= '1';
WAIT FOR 10 ns;
input <= '0';
WAIT FOR 10 ns;
input <= '0';
WAIT FOR 10 ns;
input <= '1';
```



```

    WAIT FOR 10 ns;
    input <= '0';
    WAIT;
END PROCESS;

END Behavioral;

```

Il TestBench è stato fatto prima provando il singolo shift in entrambe le modalità, cioè prima con shift a sinistra e poi a destra modificando il bit di sel (Figura 4.5), successivamente è stato provato il doppio shift sempre in entrambe le modalità (Figura 4.6).

4.3.3 La simulazione

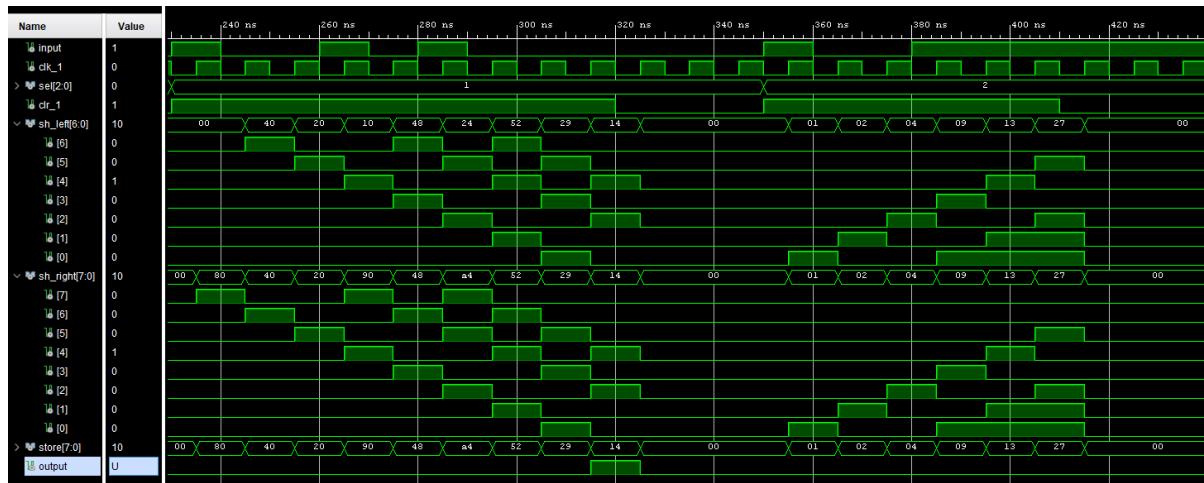


FIG. 4.5 Simulazione per singolo shift bidirezionale

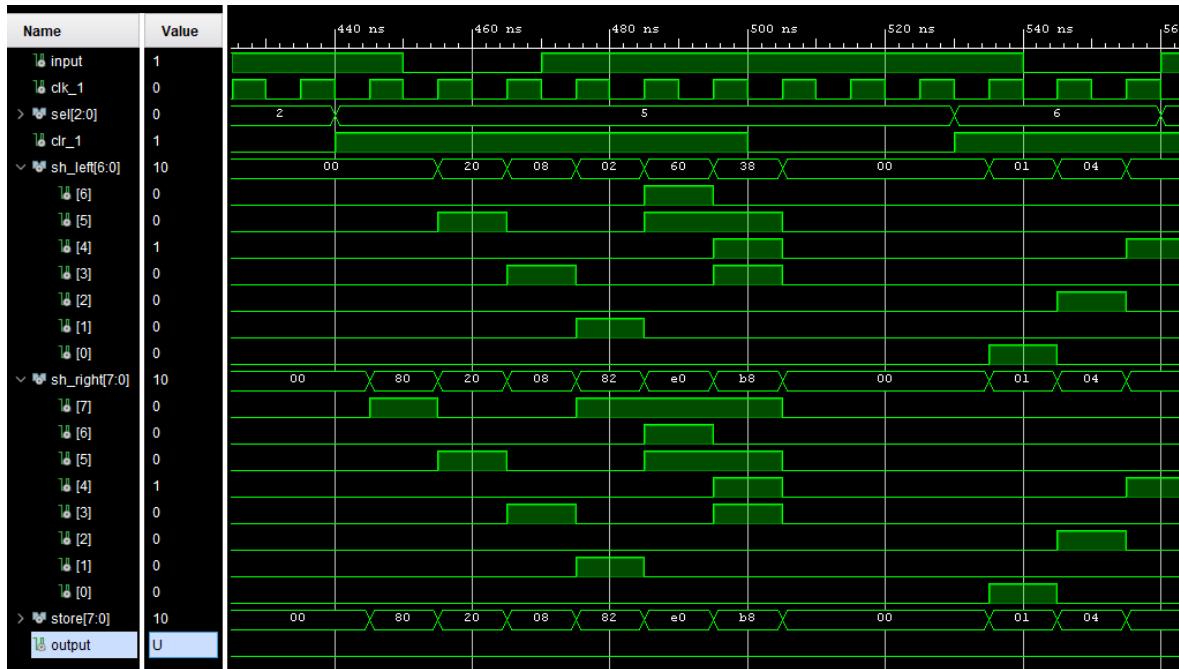


FIG. 4.5 Simulazione doppio shift bidirezionale

5 Cronometro

Si realizza un Cronometro seguendo le specifiche richieste dal cliente:

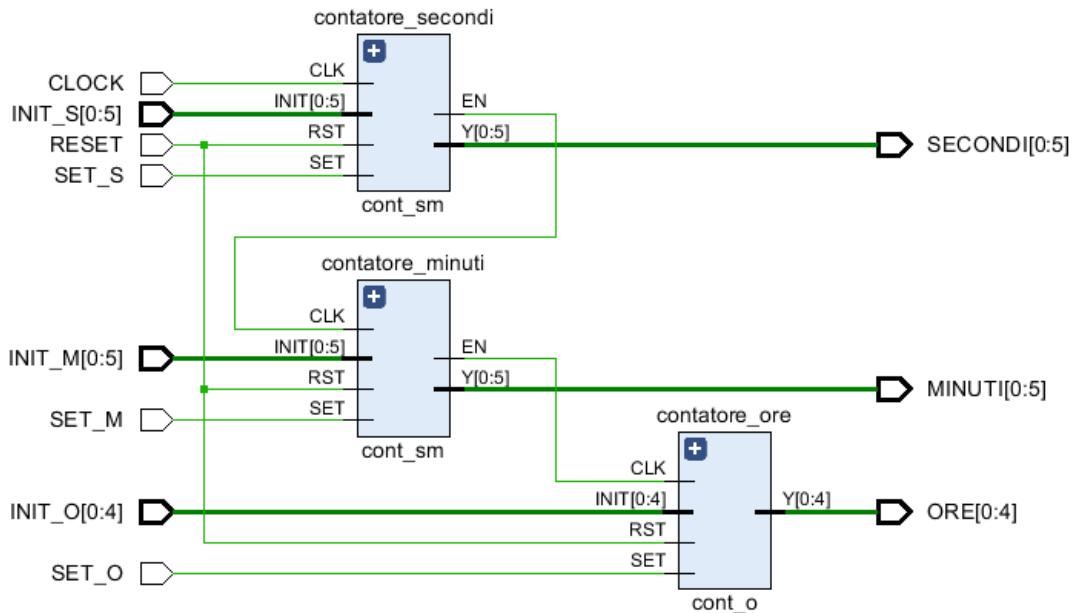
- **5.1** Progettare, implementare in VHDL e testare mediante simulazione un cronometro, in grado di scandire secondi, minuti e ore a partire da una base dei tempi prefissata (es. si consideri il clock a disposizione sulla board). Il progetto deve prevedere la possibilità di inizializzare il cronometro con un valore iniziale, sempre espresso in termini di ore, minuti e secondi, mediante un opportuno ingresso di *set*, e deve prevedere un ingresso di *reset* per azzerare il tempo. Il componente deve essere realizzato utilizzando un approccio strutturale, collegando opportunamente dei contatori secondo uno schema a scelta.
- **5.2** Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando i display a 7 segmenti per la visualizzazione dell'orario (o una combinazione di display e led nel caso in cui i display a disposizione siano in numero inferiore a quello necessario), gli switch per l'immissione dell'orario iniziale e due bottoni, uno per il *set* dell'orario e uno per il *reset*. Si utilizzi una codifica a scelta dello studente per la visualizzazione dell'orario sui display (esadecimale o decimale).
- **5.3** Estendere il componente sviluppato ai punti precedenti in modo che sia in grado di acquisire e memorizzare internamente fino ad un numero N di intertempi in corrispondenza di un ingresso di *stop*. Opzionalmente, il componente può prevedere una modalità di visualizzazione in cui, alla pressione di un bottone, vengano visualizzati sui display gli intertempi memorizzati (uno per ogni pressione).

5.1 Progettazione del Cronometro

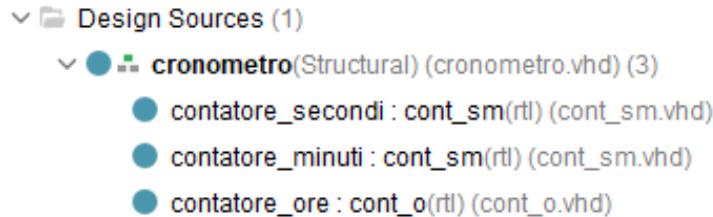
Come da richiesta del committente, il primo progetto del cronometro prevede il testing in simulazione andando opportunamente a settare l'ora, minuto e il secondo iniziale. Si è scelto, come da specifica, un approccio strutturale di cui i componenti sono i contatori. Il cuore del cronometro saranno dunque i 3 contatori posti in cascata dove il primo calcola i secondi (*contatore_secondi*) da 0 a 59, il secondo conta i minuti (*contatore_minuti*) da 0 a 59 e il terzo conta le ore (*contatore_ore*) da 0 a 23. Ciascun contatore è stato opportunamente realizzato con un approccio comportamentale incrementando il contatore fino a 59 per quanto riguarda il conteggio dei minuti e dei secondi, mentre conta fino a 24 per quanto riguarda le ore. Attenzione, per poter rappresentare 59 si è fatto uso di 6 bit ($\log_2 59$) mentre per le ore si è fatto uso di 5 bit ($\log_2 23$)

La base dei tempi di riferimento, come da specifica, è quella della board (100 MHz) cosicché occorre un divisore di frequenza con cui si possa passare a 1 Hz per avere un opportuno conteggio dei secondi ($1/T$ con $T = 1$ s). La stessa base dei tempi è posta in ingresso al contatore dei secondi in modo tale che inizia a contare i secondi e raggiunto 59 abilità un segnale di EN che attiva l'incremento del contatore dei minuti che a sua volta raggiungerà anche esso un conteggio di 59, abilità un segnale di EN che attiva l'incremento del contatore delle ore. In **FIG 5.1** si mostra lo schematico generato in fase di RTL Analysis.



**FIG 5.1** Interconnessione in cascata dei contatori 59-59-23

5.1.1 Codice VHDL del Cronometro

**FIG 5.2** Design sources dei file vhd

Di seguito si riporta il codice del componente strutturale “**cronometro.vhd**”:

Cronometro.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity cronometro is

    Port ( CLOCK: in std_logic;
            RESET: in std_logic;
            SET_S, SET_M, SET_O: in std_logic;
            INIT_S, INIT_M: in std_logic_vector(0 to 5);
            INIT_O: in std_logic_vector(0 to 4);
            SECONDI, MINUTI: out std_logic_vector(0 to 5);
    );
end entity;

```

```
      ORE: out std_logic_vector(0 to 4)
    );

end cronometro;

architecture Structural of cronometro is
component cont_sm is

  Port( CLK: in std_logic;
        RST: in std_logic;
        SET: in std_logic;
        INIT: in std_logic_vector(0 to 5);
        Y: out std_logic_vector(0 to 5);
        EN: out std_logic
  );
end component;

begin

  component cont_o is

    Port( CLK: in std_logic;
          RST: in std_logic;
          SET: in std_logic;
          INIT: in std_logic_vector(0 to 4);
          Y: out std_logic_vector(0 to 4)
    );
  end component;

  signal en1, en2: std_logic;

  begin

    contatore_secondi: cont_sm
      Port map ( CLK => CLOCK,
                  RST => RESET,
                  SET => SET_S,
                  INIT => INIT_S,
                  Y => SECONDI,
                  EN => en1
    );

    contatore_minuti: cont_sm
      Port map ( CLK => en1,
                  RST => RESET,
                  SET => SET_M,
                  INIT => INIT_M,
                  Y => MINUTI,
                  EN => en2
    );
  end;
```

```

);
contatore_ore: cont_o
Port map ( CLK => en2,
            RST => RESET,
            SET => SET_O,
            INIT => INIT_O,
            Y => ORE);

end Structural;

```

Si nota dal codice come si è riutilizzato il contatore dei secondi per i minuti poiché il conteggio è lo stesso. Lo si può notare nel seguente codice VHDL

cont_sm.vhd

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL; --overloading dell'operatore '+'

ENTITY cont_sm IS

PORT (
    CLK : IN STD_LOGIC;
    RST : IN STD_LOGIC;
    SET : IN STD_LOGIC;
    INIT : IN STD_LOGIC_VECTOR(0 TO 5);
    Y : OUT STD_LOGIC_VECTOR(0 TO 5);
    EN : OUT STD_LOGIC
);

END cont_sm;

ARCHITECTURE rtl OF cont_sm IS

SIGNAL TY : STD_LOGIC_VECTOR(0 TO 5);

BEGIN

-- Contatore modulo 60
count : PROCESS (CLK, RST, SET)
BEGIN
    IF (RST = '1') THEN
        TY <= "000000";
        EN <= '0';
    ELSIF (SET = '1') THEN
        TY <= INIT;
        EN <= '0';
    ELSIF (CLK'event AND CLK = '1') THEN
        IF (TY = "111011") THEN
            TY <= "000000";
            EN <= '1';
        ELSE
            TY <= TY + "000001";
        END IF;
    END IF;
END PROCESS;

```



```

        EN <= '0';
    END IF;
END IF;
END PROCESS;

-- Uscita del contatore
Y <= TY;

END rtl;

```

Dal codice del "cont_sm.vhd" si evince il segnale di EN che viene attivato quando *count* raggiunge il valore di 59. È possibile riusare questo codice sia per i secondi e sia per i minuti, mentre per quanto riguarda le ore è stato necessario solamente modificare il count (count = 23) e rimuovere l'**EN** visto che è la coda della cascata:

cont_sm.vhd

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL; --overloading dell'operatore '+'

ENTITY cont_o IS

PORT (
    CLK : IN STD_LOGIC;
    RST : IN STD_LOGIC;
    SET : IN STD_LOGIC;
    INIT : IN STD_LOGIC_VECTOR(0 TO 4);
    Y : OUT STD_LOGIC_VECTOR(0 TO 4)
);

END cont_o;

ARCHITECTURE rtl OF cont_o IS

SIGNAL TY : STD_LOGIC_VECTOR(0 TO 4);

BEGIN

-- Contatore modulo 24
count : PROCESS (CLK, RST, SET)
BEGIN
    IF (RST = '1') THEN
        TY <= "00000";
    ELSIF (SET = '1') THEN
        TY <= INIT;
    ELSIF (CLK'event AND CLK = '1') THEN
        IF (TY = "10111") THEN
            TY <= "00000";
        ELSE
            TY <= TY + "00001";
        END IF;
    END IF;
END PROCESS;

```



```
-- Uscita del contatore
Y <= TY;
```

```
END rtl;
```

Si poteva trovare una soluzione differente?

Si, un ulteriore soluzione può essere quella di utilizzare un approccio strutturale dove ciascun contatore potrebbe essere realizzato secondo il modello con *adder*.

5.1.2 La simulazione del Cronometro



Dalla simulazione si può notare che dopo lo scoccare del secondo 59 i secondi passano a 00 (e si incrementano di 1 i minuti), mentre i minuti passano anche essi da 59 a 00, ed infine, le ore passano da 10 a 11.

cronometroTB.vhd

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

entity cronometroTB is
end cronometroTB;

architecture behavioral of cronometroTB is
component cronometro is

    Port ( CLOCK: in std_logic;
           RESET: in std_logic;
           SET_S, SET_M, SET_O: in std_logic;
           INIT_S, INIT_M: in std_logic_vector(7 downto 0);
           INIT_O: in std_logic_vector(7 downto 0);
           CIFRA1_SECONDI, CIFRA2_SECONDI: out std_logic_vector(3
downto 0);
           CIFRA1_MINUTI, CIFRA2_MINUTI: out std_logic_vector(3
downto 0);
```

```

        CIFRA1_ORE, CIFRA2_ORE: out std_logic_vector(3 downto 0)
    );

end component;

signal c, r: std_logic;
signal sets, setm, seto: std_logic;
signal in_s, in_m: std_logic_vector(7 downto 0);
signal in_o: std_logic_vector(7 downto 0);
signal c1_sec, c2_sec, c1_min, c2_min: std_logic_vector(3 downto 0);
signal c1_ore, c2_ore: std_logic_vector (3 downto 0);

begin
    uut: cronometro
        port map ( CLOCK => c,
                    RESET => r,
                    SET_S => sets,
                    SET_M => setm,
                    SET_O => seto,
                    INIT_S => in_s,
                    INIT_M => in_m,
                    INIT_O => in_o,
                    CIFRA1_SECONDI => c1_sec,
                    CIFRA2_SECONDI => c2_sec,
                    CIFRA1_MINUTI => c1_min,
                    CIFRA2_MINUTI => c2_min,
                    CIFRA1_ORE => c1_ore,
                    CIFRA2_ORE => c2_ore
    );
prc: process
begin
    wait for 10 ns;
    -- resetto il cronometro a 00:00:00
    r <= '1';
    wait for 10 ns;
    r <= '0';
    -- setto l'ora iniziale a 10:59:40
    sets <= '1';
    in_s <= "00101000";
    wait for 10 ns;
    sets <= '0';
    setm <= '1';
    in_m <= "00111011";
    wait for 10 ns;

    setm <= '0';
    seto <= '1';
    in_o <= "00001010";

```

```

wait for 10 ns;

seto <= '0';

for i in 0 to 40 loop
    wait for 10 ns;
    C<='1';
    wait for 10 ns;
    C<='0';
end loop;
wait;
end process;

end behavioral;

```

5.2 Progettazione del Cronometro on Board

Per implementare sulla board il componente sviluppato al punto precedente, si fa uso del display a 7 segmenti per la visualizzazione dell'orario. Per questa versione del cronometro si farà uso degli switch per l'immissione dell'orario iniziale e due bottoni, uno per il set dell'orario e uno per il reset. Si è poi pensata ad una soluzione per la conversione in decimale opportuna dei valori da passare al display. Si è dunque esteso il progetto precedente aggiungendo opportunamente i nuovi componenti relativi ai bottoni, agli switch e al display (**FIG. 5.3**).

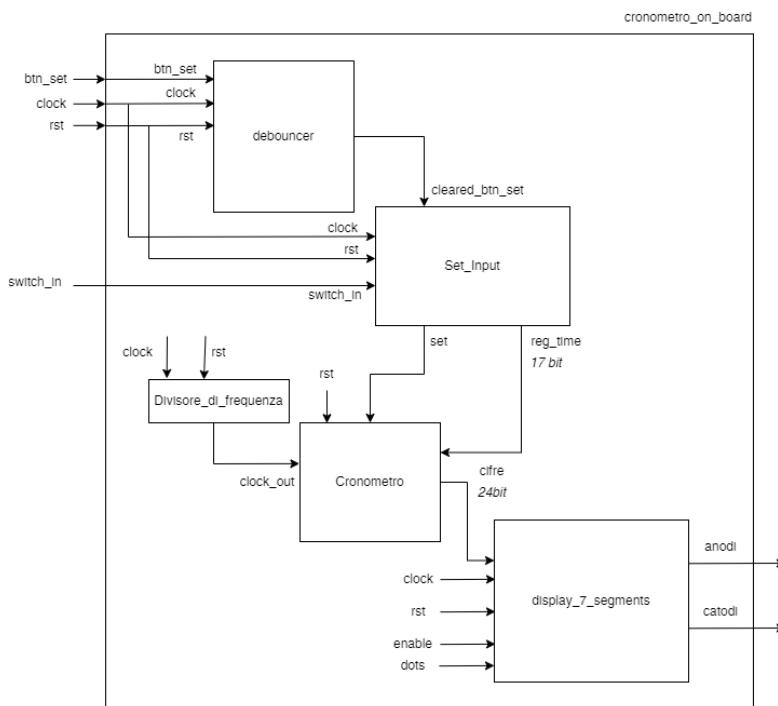


FIG 5.3 Schematico semblice Debouncer – Cronometro – Display

I componenti più importanti di questa estensione del progetto sono il Debouncer, analizzato già nei paragrafi precedenti, utilizzato per rimuovere le oscillazioni indesiderate dovute dalla pressione del

button sulla board. Perché è necessario? È necessario perché ad ogni pressione fatto sul button evolve l'automa della macchina sequenziale “*set_input*” e dunque oscillazioni indesiderate possono compromettere il funzionamento del cronometro e del relativo inserimento dei valori iniziali. Inoltre, si fa uso dello stesso Display_Seven_Segment precedentemente analizzato e che gentilmente è stato offerto durante il corso dai docenti.

5.2.1 Codice VHDL del Cronometro on Board

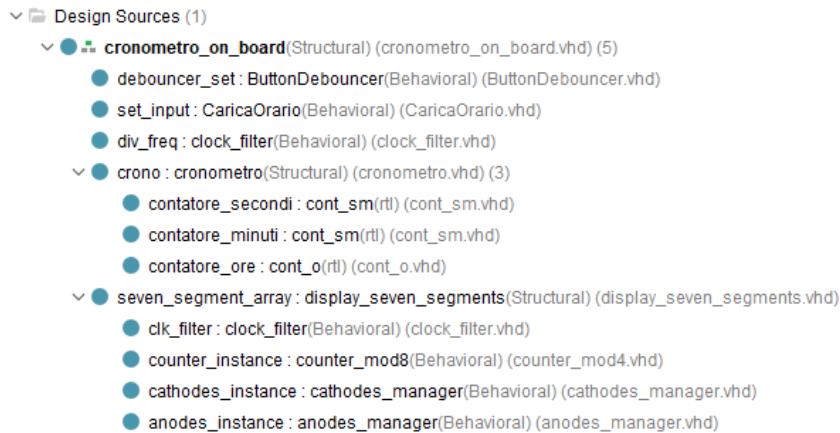


FIG 5.4 Design Sources cronometro_on_board

Sono stati **riutilizzati** i componenti: *debouncer*, *div_freq*, *cronometro*, *display_seven_segments*. Si riporta di seguito il top module per determinare le nuove interconnessioni.

chronometro_on_board.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity cronometro_on_board is
  Port(
    clock_in : in STD_LOGIC;
    reset_in : in STD_LOGIC;
    --bottone per settare l'orario iniziale
    set_init : in STD_LOGIC;
    --switch per inizializzazione orologio
    switch_in : in STD_LOGIC_VECTOR(5 downto 0);
    --visore
    anodes_out : out STD_LOGIC_VECTOR (7 downto 0); --anodi e catodi
    delle cifre, sono un output del topmodule
    cathodes_out : out STD_LOGIC_VECTOR (7 downto 0)
  );
end cronometro_on_board;

architecture Structural of cronometro_on_board is
  -- // COMPONENTI

```

```
COMPONENT ButtonDebouncer
    GENERIC (
        CLK_period: integer := 10; -- periodo del clock della board in
        nanosecondi
        btn_noise_time: integer := 6500000 --intervallo di tempo in cui
        si ha l'oscillazione del bottone
                                            --assumo che duri
        6.5ms=6500microsec=6500000ns
    );
    PORT ( RST : in STD_LOGIC;
            CLK : in STD_LOGIC;
            BTN : in STD_LOGIC;
            CLEARED_BTN : out STD_LOGIC);
END COMPONENT;

COMPONENT CaricaOrario is
    port(
        x : in std_logic_vector(5 downto 0);
        set : in std_logic;
        clk : in std_logic;
        reset : in std_logic;
        reg_time : out std_logic_vector(16 downto 0);
        load : out std_logic
    );
END COMPONENT;

COMPONENT cronometro
    PORT (
        CLOCK: in std_logic;
        RESET: in std_logic;
        SET : in std_logic;
        INIT_S, INIT_M: in std_logic_vector(5 downto 0);
        INIT_O: in std_logic_vector(4 downto 0);
        CIFRA1_SECONDI, CIFRA2_SECONDI: out std_logic_vector(3 downto 0);
        CIFRA1_MINUTI, CIFRA2_MINUTI: out std_logic_vector(3 downto 0);
        CIFRA1_ORE, CIFRA2_ORE: out std_logic_vector(3 downto 0)
    );
END COMPONENT;

COMPONENT clock_filter
    generic(
        CLKIN_freq : integer := 100000000; --clock board 100MHz
        CLKOUT_freq : integer := 500           --frequenza desiderata 500Hz
    );
    Port (
        clock_in : in STD_LOGIC;
        reset : in STD_LOGIC;
```

```

    clock_out : out STD_LOGIC -- attenzione: non è un vero clock ma
un impulso che sarà usato come enable
);
END COMPONENT;

```

```

COMPONENT display_seven_segments
PORT (
    CLK : IN std_logic;
    RST : IN std_logic;
    VALUE : IN std_logic_vector(31 downto 0);--valori da
mostrare sul display
    ENABLE : IN std_logic_vector(7 downto 0);--abilitazione di
ciascuna cifra (accensione)
    DOTS : IN std_logic_vector(7 downto 0); --abilitazione
punti (accensione)
    ANODES : OUT std_logic_vector(7 downto 0);
    CATHODES : OUT std_logic_vector(7 downto 0)
);
END COMPONENT;

```

-- i segnali che posso prelevare (reset, read_strobe_set(lettura dal bottone), value_temp (valore da leggere)

```

signal reset_n : std_logic;
signal read_strobe_set : std_logic;
signal clock_crono : std_logic;
signal value_temp : std_logic_vector(31 downto 0) := (others => '0');
signal set_in : std_logic;
signal set_time : std_logic_vector(16 downto 0);

```

begin

--IN CASO DI SCELTA DEL BUTTON CPU_RESET E' NECESSARIA QUESTA OPERAZIONE
reset_n <= not reset_in; --visto che utilizzo il bottone CPU_reset della board, che è attivo-basso,
--devo invertire il segnale di reset

-- MAPPATURA DEI SEGNALI

```

debouncer_set: ButtonDebouncer
GENERIC MAP(
    CLK_period => 10, -- periodo del clock della board pari a 10ns
    btn_noise_time => 500000000 --intervallo di tempo in cui si ha
l'oscillazione del bottone
    --assumo che duri 0.5s=500000000ns
)
PORT MAP ( RST => reset_n,

```



```
CLK => clock_in,
BTN => set_init,
CLEARED_BTN => read_strobe_set
);

set_input : CaricaOrario
Port Map(
    x => switch_in,
    set => read_strobe_set,
    clk => clock_in,
    reset => reset_n,
    reg_time => set_time,
    load => set_in
);

div_freq: clock_filter
GENERIC MAP(
    CLKIN_freq => 100000000, --clock board 100MHz
    CLKOUT_freq => 1           --frequenza desiderata 1Hz
)
PORT MAP(
    clock_in => clock_in,
    reset => reset_n,
    clock_out => clock_crono
);

crono: cronometro
PORT MAP (
    CLOCK => clock_crono,
    RESET => reset_n,
    SET => set_in,
    INIT_S => set_time(5 downto 0),
    INIT_M => set_time(11 downto 6),
    INIT_O => set_time(16 downto 12),
    CIFRA1_SECONDI => value_temp(7 downto 4),
    CIFRA2_SECONDI => value_temp(3 downto 0),
    CIFRA1_MINUTI => value_temp(15 downto 12),
    CIFRA2_MINUTI => value_temp(11 downto 8),
    CIFRA1_ORE => value_temp(23 downto 20),
    CIFRA2_ORE => value_temp(19 downto 16)
);

seven_segment_array: display_seven_segments
PORT MAP(
    CLK => clock_in,
    RST => reset_n,
    value => value_temp,
    enable => "00111111", --accendo solo le prime 6 cifre a partire da
destra
```

```

dots => "00010100", --accendo i punti per separare le ore dai minuti
e i minuti dai secondi
anodes => anodes_out,
cathodes => cathodes_out
);
end Structural;

```

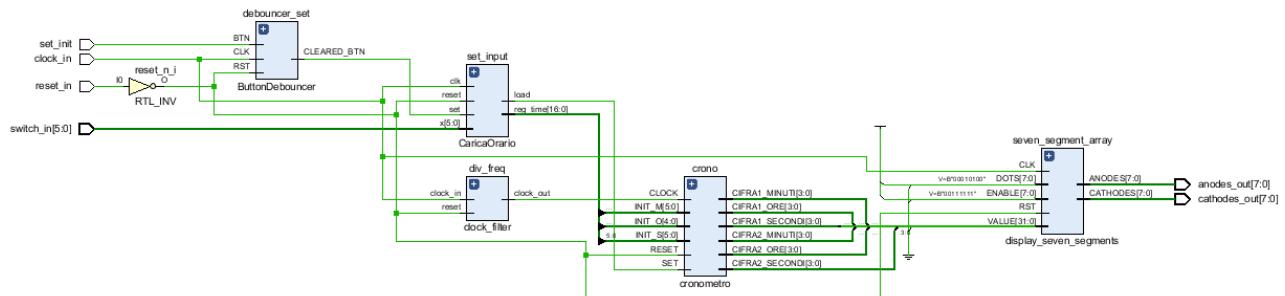


FIG 5.5 Schematico generato da Vivado

Per poter permettere il corretto inserimento dei valori sul cronometro a valle della pressione del button, si è scelto di implementare un automa. Quest'ultimo descritto nel componente **CaricoOrario.vhd**.

Dall'automa si nota come è possibile settare il cronometro utilizzando un solo button. L'idea è quella di settare inizialmente l'ora, poi i minuti e poi i secondi. Non viceversa, perché c'è il rischio di perdere qualche tempo e non avere il tempo desiderato (i.e se setto come secondo 58, come minuto 59 e poi l'ora 18, avrò perso qualche secondo di tempo per settare). (**FIG. 5.6**)

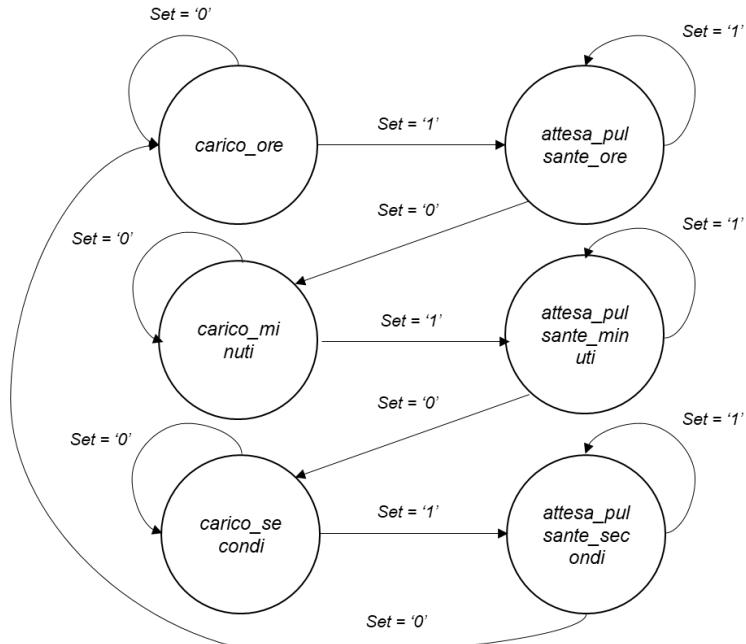


FIG 5.6 Automa caricamento ore-minuti-secondi

CaricoOrario.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CaricaOrario is
    port(
        x : in std_logic_vector(5 downto 0);
        set : in std_logic; --button per settare gli orari h, m, s
        clk : in std_logic;
        reset : in std_logic;
        reg_time : out std_logic_vector(16 downto 0); --effettivi h, m, s
        load : out std_logic --abilitazione degli orari
    );
end CaricaOrario;

architecture Behavioral of CaricaOrario is

type state is (carico_ore, attesa_pulsante_ore, carico_minuti,
attesa_pulsante_minuti, carico_secondi, attesa_pulsante_secondi);
signal current_state : state := carico_ore;

--serve ad abilitare il set del cronometro
signal load_temp : std_logic := '0';
--segnali temporanei in cui viene memorizzato il valore da settare sul
cronometro
signal reg_s_temp : std_logic_vector(5 downto 0) := (others => '0');
signal reg_m_temp : std_logic_vector(5 downto 0) := (others => '0');
signal reg_h_temp : std_logic_vector(4 downto 0) := (others => '0');

begin

load <= load_temp;
reg_time <= reg_h_temp & reg_m_temp & reg_s_temp; --l'uscita di questo
modulo è la concatenazione dei tre valori impostati sugli switch

p : process(clk, reset)
begin
if(reset = '1') then
    current_state <= carico_ore;
    load_temp <= '0';

elsif (clk'event AND clk = '1') then

    case current_state is

        -- CARICO LE ORE
        when carico_ore =>
            if(set = '0') then
                current_state <= carico_ore;
                load_temp <= '0';
            end if;
        when attesa_pulsante_ore =>
            if(set = '1') then
                current_state <= attesa_pulsante_ore;
                load_temp <= '1';
            end if;
        when others =>
            current_state <= current_state;
            load_temp <= load_temp;
    end case;
end if;
end process;

```



```

        else
            reg_h_temp <= x(4 downto 0); --switch
            current_state <= attesa_pulsante_ore;
            load_temp <= '0';
        end if;

-- ATTENDERE PULSANTE
when attesa_pulsante_ore =>
    if(set = '0') then
        current_state <= carico_minuti;
        load_temp <= '0';

    else
        current_state <= attesa_pulsante_ore;
    end if;

-- CARICO I MINUTI
when carico_minuti =>
    if(set = '0') then
        current_state <= carico_minuti;
        load_temp <= '0';

    else
        reg_m_temp <= x;
        current_state <= attesa_pulsante_minuti;
        load_temp <= '0';
    end if;

-- ATTENDERE PULSANTE
when attesa_pulsante_minuti =>
    if(set = '0') then
        current_state <= carico_secondi;
        load_temp <= '0';

    else
        current_state <= attesa_pulsante_minuti;
    end if;

-- CARICO I SECONDI E INVIO I DATI AL CRONOMETRO
when carico_secondi =>
    if(set = '0') then
        current_state <= carico_secondi;
        load_temp <= '0';

    else
        reg_s_temp <= x;
        current_state <= attesa_pulsante_secondi;
        load_temp <= '1';
    end if;

-- ATTENDERE PULSANTE
when attesa_pulsante_secondi =>
    if(set = '0') then
        current_state <= carico_ore;

```



```

        load_temp <= '0';

    else
        current_state <= attesa_pulsante_secondi;
        load_temp <= '0';
    end if;

    -- ALTRI CASI
    when others =>
        current_state <= carico_ore;
        load_temp <= '0';

end case;

end if;

end process;
end Behavioral;

```

Per poter gestire nel migliore dei modi le cifre e visualizzarle sul display in decimale, si è pensato ad una soluzione particolare:

- Avere un *count* in base decimale;
- Prendersi il **primo** valore del risultato attraverso la divisione di 10 (i.e $24/10 = 2$) e convertirlo in binario;
- Prendersi il **secondo** valore del risultato attraverso il modulo di 10 (i.e $24 \bmod 10 = 4$) e convertirlo in binario;

In modo tale che in uscita dal modulo contatore avrà 8 bit, di cui 4 bit codificheranno la cifra 2 e 4 bit codificheranno la cifra 1, tale da poter opportunamente mappato con il Display7Segments.

Cont_sm.vhd

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY cont_sm IS

    PORT (
        CLK : IN STD_LOGIC;
        RST : IN STD_LOGIC;
        SET : IN STD_LOGIC;
        INIT : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
        Y1 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";
        Y2 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";
        EN : OUT STD_LOGIC
    );
END cont_sm;

ARCHITECTURE rtl OF cont_sm IS

```

```

SIGNAL TY1, TY2 : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";
SIGNAL count : INTEGER;
SIGNAL temp : INTEGER;

BEGIN

    -- Contatore modulo 60
    count_sec_min : PROCESS (CLK, RST)
    BEGIN
        IF (RST = '1') THEN
            count <= 0;
            EN <= '0';
        ELSIF (SET = '1') THEN
            temp <= to_integer(unsigned(INIT));
            IF (temp >= 60) THEN
                count <= 0;
            ELSE
                count <= temp;
            END IF;
            EN <= '0';
        ELSIF (CLK'event AND CLK = '1') THEN
            IF (count = 59) THEN
                count <= 0;
                EN <= '1';
            ELSE
                count <= count + 1;
                EN <= '0';
            END IF;
        END IF;
    END PROCESS;

    -- Uscita del contatore
    -- si suddividono due uscite, una per la cifra 1(TY1) e cifra 2
    (TY2) opportunamente con
    -- la divisione di 10 e il modulo di 10 in maniera da mostrare
    l'orario in formato decimale
    TY1 <= STD_LOGIC_VECTOR(to_unsigned(count/10, 4));
    TY2 <= STD_LOGIC_VECTOR(to_unsigned(count MOD 10, 4));
    Y1 <= TY1;
    Y2 <= TY2;

END rtl;

```

Ci poteva essere un'altra soluzione?

L'ulteriore soluzione per evitare ridondanze nel codice era utilizzare una visualizzazione in esadecimale dell'orologio, ma non avrebbe avuto senso in una possibile applicazione reale.



5.3 Progettazione del Cronometro on Board con Intertempi

Si è esteso il componente sviluppato ai punti precedenti in modo che si è in grado di acquisire e memorizzare internamente fino ad un numero N di intertempi in corrispondenza di un ingresso di *stop*. Si è scelto opportunamente $N = 2$. Inoltre, si è previsto anche la modalità di visualizzazione in cui, alla pressione di un bottone, vengano visualizzati sui display gli intertempi memorizzati (uno per ogni pressione). Si è dunque pensato di aggiungere un MUX per dividere le modalità di utilizzo del cronometro, ovvero, se alzo lo switch dedicato all'abilitazione degli intertempi verrà prelevato, alla pressione del button, i valori salvati nella memoria opportunamente progettata nel componente *intertempi*. Dallo schematico nella **FIG 5.7** è possibile notare l'estensione in area del circuito.

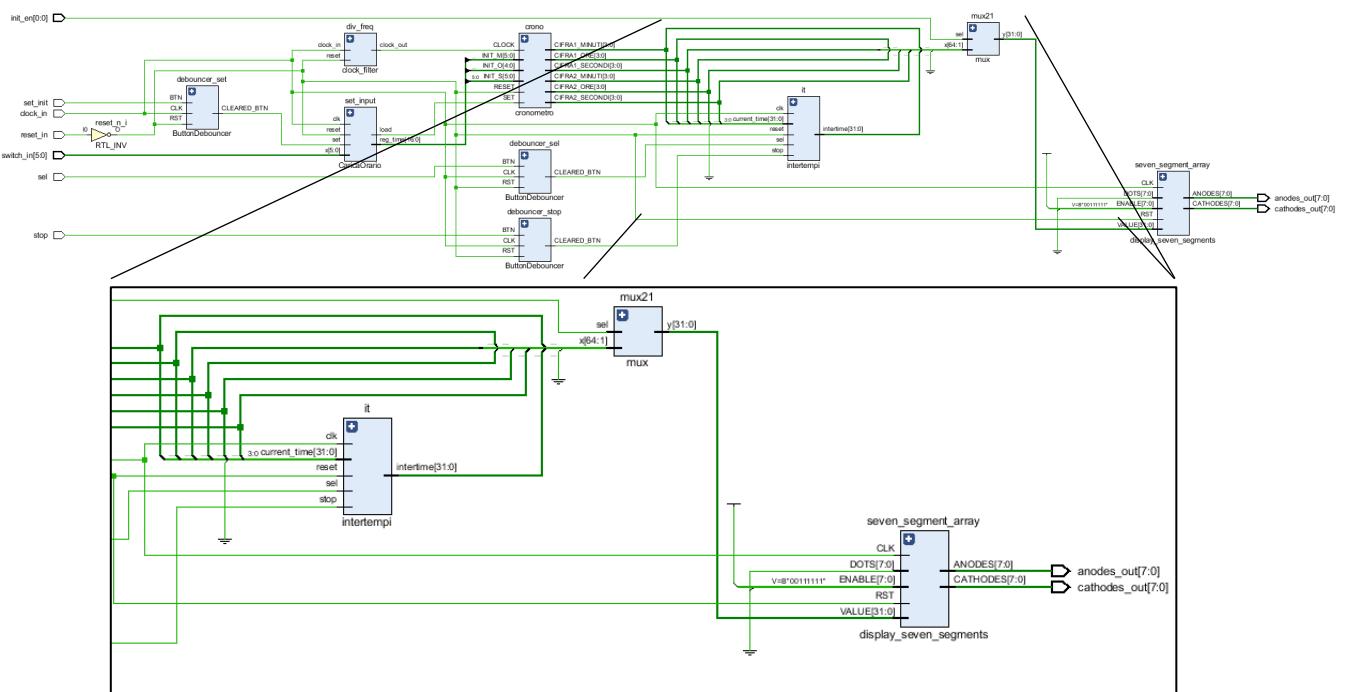


FIG 5.7 Zoom schematico interconnessioni intertempi e mux 2:1

In particolare, il componente *intertempi* è composto da:

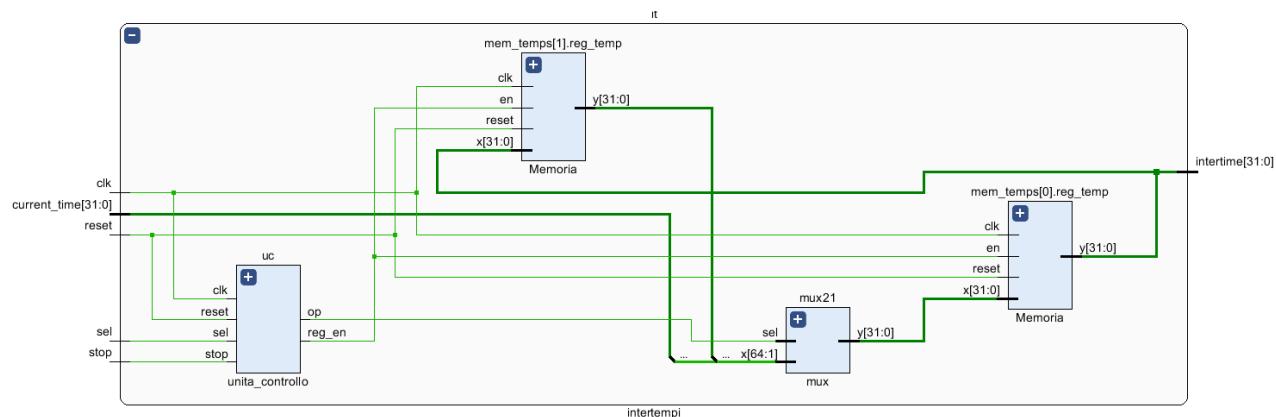


FIG 5.8 Componentistica IT

Si deve dunque creare una unità di controllo che permetterà opportunamente di andare ad abilitare le memorie e la relativa memorizzazione dell'intertempo. Si è scelto di progettare l'unità di controllo in logica cablata, in particolare si è progettato un automa (**FIG 5.10**).

5.3.1 Codice VHDL del Cronometro on Board

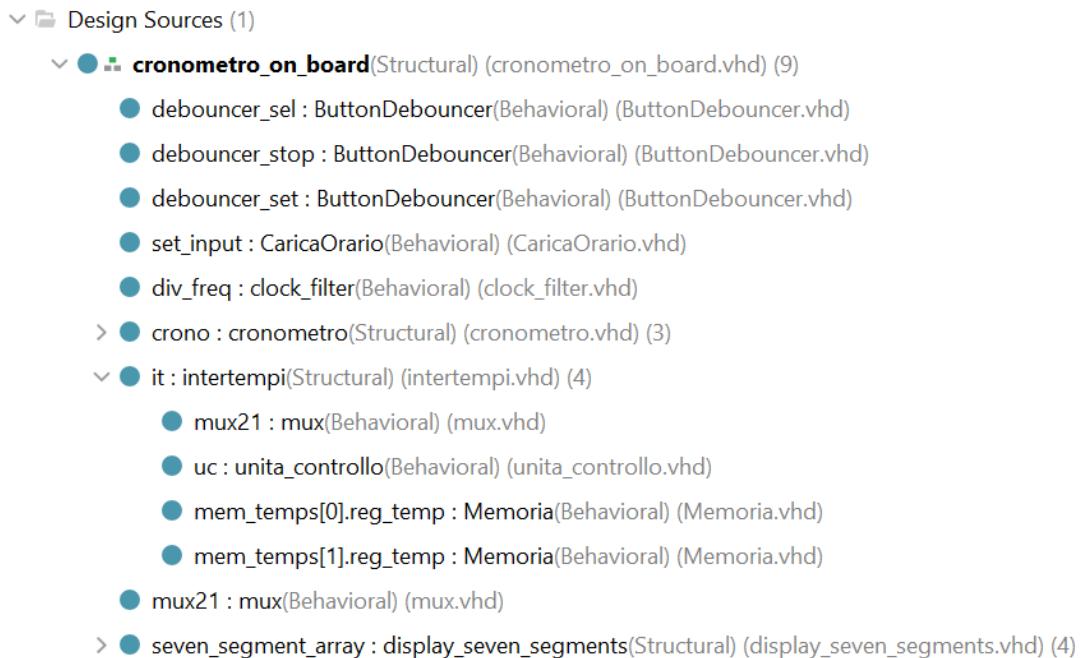


FIG 5.9 Design sources del Cronometro on Board con intertempi

Si riporta qui il codice del componente:

intertempi.vhd

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

entity intertempi is
  generic (
    n_register : positive := 2
  );
  port (
    stop : in std_logic;
    reset : in std_logic;
    clk : in std_logic;
    sel : in std_logic;
    current_time : in std_logic_vector(31 downto 0);
    intertime : out std_logic_vector(31 downto 0)
  );
end intertempi;
```

```
architecture Structural of intertempi is

-- // COMPONENTI

component mux is
    generic (
        n      : positive := 1
    );

    port (
        x      : in std_logic_vector(32*(2**n) downto 1);
        sel   : in std_logic_vector(n-1 downto 0);
        y      : out std_logic_vector(31 downto 0)
    );
end component;

component unita_controllo is
    port (
        stop : in std_logic;
        reset : in std_logic;
        clk : in std_logic;
        sel : in std_logic;
        reg_en : out std_logic;
        op : out std_logic -- op=0 shift circolare, op=1 inserisci nuovo
dato
    );
end component;

component Memoria
    port (
        x          : in std_logic_vector(31 downto 0);
        en         : in std_logic;
        clk        : in std_logic;
        reset      : in std_logic;
        y          : out std_logic_vector(31 downto 0)
    );
end component;

-- // SEGNALI UTILI

signal s_sel_mux : std_logic := '0';
signal s_reg_en : std_logic := '0';
signal s_temp : std_logic_vector(1 to 32*(n_register+1));
signal var : std_logic_vector(1 to 64);

begin

-- // MAPPING
```

```
intertime <= s_temp(33 to 64);
var <= s_temp(32*n_register + 1 to 32*(n_register+1)) &
current_time;

mux21 : mux
port map (
    x => var,
    sel(0) => s_sel_mux,
    y => s_temp(1 to 32)
);

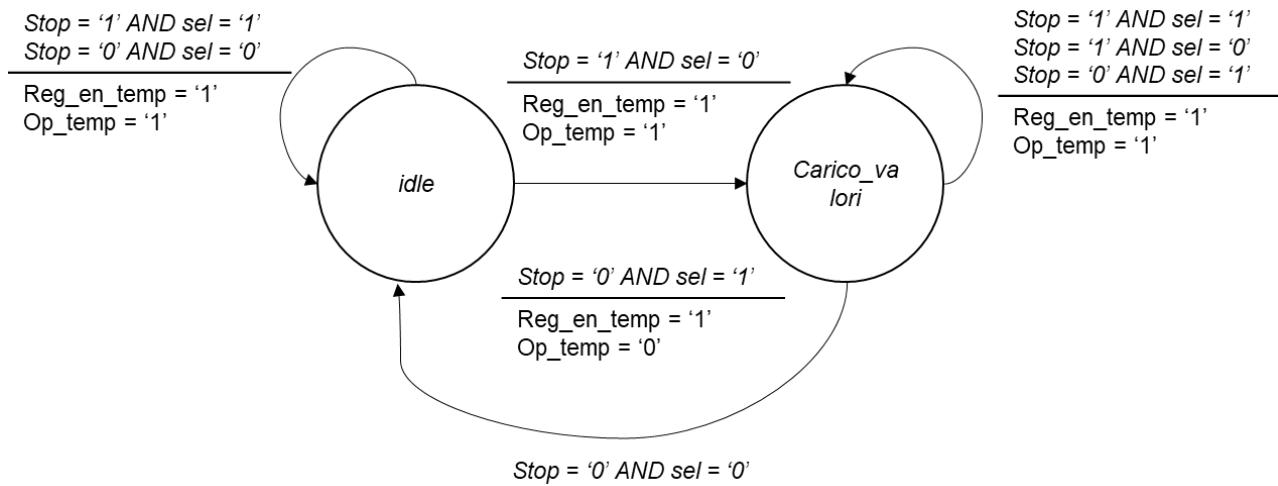
uc : unita_controllo
port map(
    stop => stop,
    reset => reset,
    clk => clk,
    sel => sel,
    reg_en => s_reg_en,
    op => s_sel_mux
);

mem_temps : for i in 0 to n_register - 1 generate
    reg_temp : Memoria
        port map(
            x => s_temp(32*i + 1 to 32*(i+1)),
            en => s_reg_en,
            clk => clk,
            reset => reset,
            y => s_temp(32*(i+1) + 1 to 32*(i+2))
        );
    end generate;

end Structural;
```

Come detto sopra, per la realizzazione dell'unità di controllo si è usato un approccio cablato. Dunque, una macchina sequenziale come quella riportata nella figura seguente:



**FIG 5.10** Automa memorizzazione

Se si preme il pulsante *stop* si carica il valore in memoria. Quando si preme invece il pulsante *sel* si va visualizzare gli ultimi due intertempi salvati in memoria. Di seguito il codice VHDL:

unita_controllo.vhd

```

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

entity unita_controllo is
    port (
        stop : in std_logic;
        reset : in std_logic;
        clk : in std_logic;
        sel : in std_logic;
        reg_en : out std_logic;
        op : out std_logic
    );
end unita_controllo;

architecture Behavioral of unita_controllo is

type state is (idle, carico_valori);
signal current_state : state := idle;
signal reg_en_temp : std_logic := '0';
signal op_temp : std_logic := '0';

begin
    reg_en <= reg_en_temp;
    op <= op_temp;

```

```
p : process(clk, reset)
begin

    if(reset = '1') then
        current_state <= idle;
        reg_en_temp <= '0';
        op_temp <= '0';

    elsif(clk'event AND clk = '1') then

        case current_state is

            when idle =>

                if (stop = '1' AND sel = '0') then
                    current_state <= carico_valori;
                    reg_en_temp <= '1';
                    op_temp <= '1';

                elsif (stop = '0' AND sel = '1') then
                    current_state <= carico_valori;
                    reg_en_temp <= '1';
                    op_temp <= '0';

                else
                    current_state <= idle;
                    reg_en_temp <= '0';
                    op_temp <= '0';

                end if;

            when carico_valori =>

                if(stop = '0' and sel = '0') then
                    current_state <= idle;

                else
                    current_state <= carico_valori;

                end if;

                reg_en_temp <= '0';
                op_temp <= '0';

            when others =>

                current_state <= idle;
                reg_en_temp <= '0';
```

```
        op_temp <= '0';

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

La memoria presenta in ingresso un segnale di *enable* utile per abilitare la scrittura su di essa.

Memoria.vhd

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

entity Memoria is
    port (
        x           : in std_logic_vector(31 downto 0);
        en          : in std_logic;
        clk         : in std_logic;
        reset       : in std_logic;
        y           : out std_logic_vector(31 downto 0)
    );
end Memoria;

architecture Behavioral of Memoria is

signal y_temp : std_logic_vector(31 downto 0) := (others => '0');

begin

    y <= y_temp;
    p : process(clk, reset)
    begin

        if(reset = '1') then
            y_temp <= (others => '0');

        elsif(clk'event and clk = '0' and en = '1') then
            y_temp <= x;
        end if;
    end process;
end Behavioral;
```

Il Multiplexer 2:1 sarà utile per gestire i modi del cronometro. Se lo switch dedicato è posto a 0 si ha la modalità *cronometro* altrimenti si è in modalità *intertempi view*.



mux.vhd

```

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

entity mux is
    generic (
        n      : positive := 2
    );
    port (
        x      : in std_logic_vector(32*(2**n) downto 1);
        sel    : in std_logic_vector(n-1 downto 0);
        y      : out std_logic_vector(31 downto 0)
    );
end mux;
architecture Behavioral of mux is

signal y_temp : std_logic_vector(31 downto 0) := (others => '0');

begin
    y <= y_temp;
    p : process(sel, x)
        variable temp : integer;

        begin
            temp := to_integer(unsigned(sel));
            y_temp <= x((32 * (2**n - temp)) downto (32 * (2**n - temp - 1)
+ 1));
        end process;
end Behavioral;

```

La soluzione adottata in questo paragrafo non ha soddisfatto del tutto l'idea di utilizzo del cronometro in quanto presenta diversi limiti. Il primo limite è sull'idea di uso del cronometro. Si deve per forza modificare prima l'ora poi i minuti e poi i secondi, ma se si volesse modificare soltanto i minuti avrei la necessità di modificare prima l'ora e poi i minuti. Dunque, si è pensata ad un ulteriore soluzione.

5.4 Progettazione del Cronometro on Board (Modifica dell'orologio contemporanea)

Sulla base di quanto fatto in precedenza, si è attuata l'idea di poter modificare a piacimento l'ora, i minuti e i secondi senza dover ricorrere alla ripetizione di inserimento che si ha nell'implementazione 5.2. Si fa uso degli switch per “*fermare il tempo*” e tre bottoni differenti per l'immissione dell'orario iniziale. Si aggiungeranno dunque dei *button debouncer* in più (set_h, set_m, set_s) e un collegamento *ore_en* collegato al cronometro e sarà il segnale di ingresso che mi determinerà quando fermare il tempo. In questo modo, lo schematico sarà dunque:



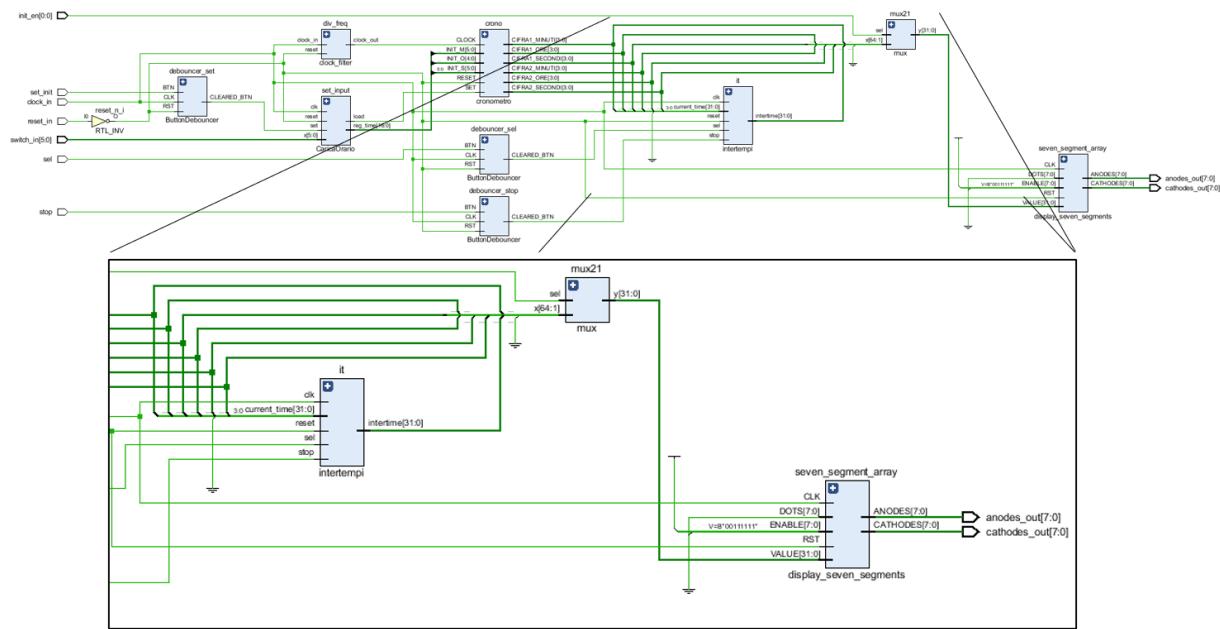


FIG 5.11 Zoom delle connessioni Debouncer set_h, set_m, set_s

È stato dunque necessario fare delle modifiche nel cronometro andando opportunamente a definire quando il tempo si deve fermare per far sì che possano essere fatte le modifiche senza perdere nessun tempo. Rispetto alla versione precedente si è evitato di inserire una macchina sequenziale.

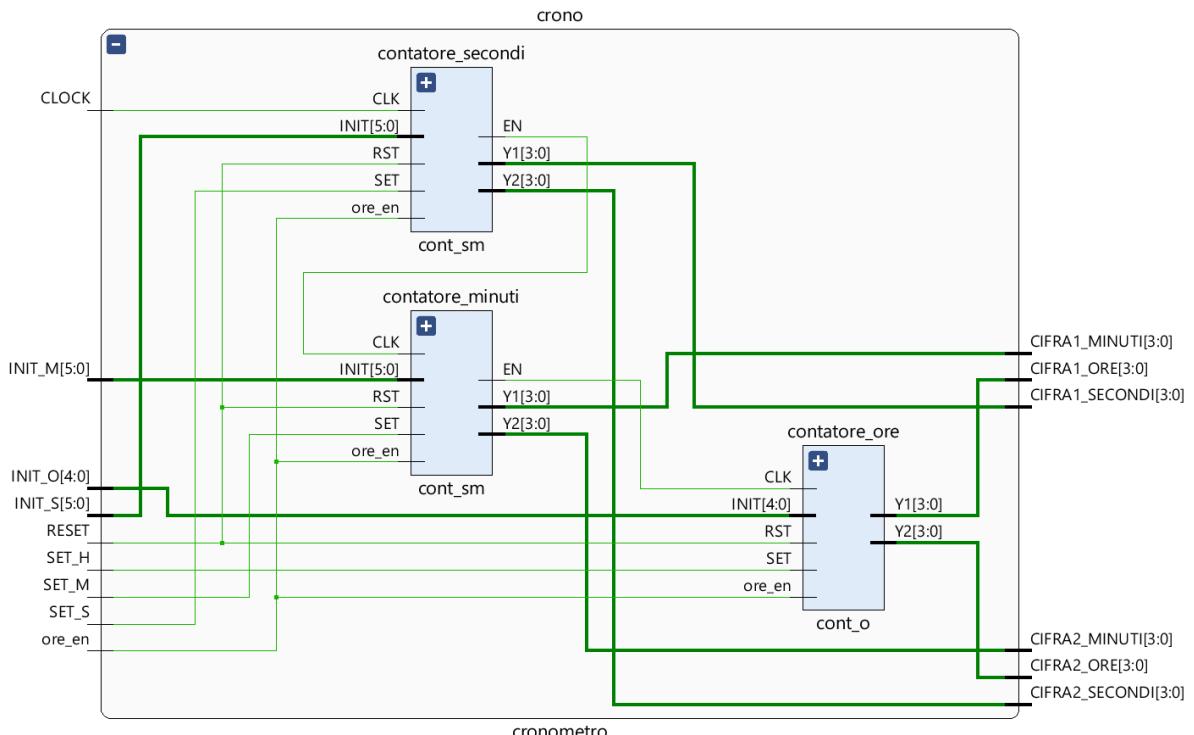


FIG 5.11 Schematico del cronometro con aggiunta del segnale `ore_en`

5.4.1 Codice VHDL del Cronometro on Board (Modifica dell'orologio contemporanea)

```

    ▾ Design Sources (1)
        ▾ cronometro_on_board(Structural) (cronometro_on_board.vhd) (10)
            debouncer_sel : ButtonDebouncer(Behavioral) (ButtonDebouncer.vhd)
            debouncer_stop : ButtonDebouncer(Behavioral) (ButtonDebouncer.vhd)
            debouncer_set_h : ButtonDebouncer(Behavioral) (ButtonDebouncer.vhd)
            debouncer_set_m : ButtonDebouncer(Behavioral) (ButtonDebouncer.vhd)
            debouncer_set_s : ButtonDebouncer(Behavioral) (ButtonDebouncer.vhd)
            div_freq : clock_filter(Behavioral) (clock_filter.vhd)
        ▾ crono : cronometro(Structural) (cronometro.vhd) (3)
            contatore_secondi : cont_sm(rtl) (cont_sm.vhd)
            contatore_minuti : cont_sm(rtl) (cont_sm.vhd)
            contatore_ore : cont_o(rtl) (cont_o.vhd)
        > it : intertempi(Structural) (intertempi.vhd) (4)
            mux21 : mux(Behavioral) (mux.vhd)
        > seven_segment_array : display_seven_segments(Structural) (display_seven_segments.vhd) (4)

```

FIG 5.12 Design Sources

Si riporta solamente il contatore **cont_sm.vhd** in quanto la modifica (indicato in rosso e in grassetto) è la stessa negli altri contatori e tutto il resto del codice non è stato toccato.

cont_sm.vhd

```

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

entity cont_sm is

    Port( CLK: in  std_logic;
          RST: in  std_logic;
          SET: in std_logic;
          ore_en: in std_logic;
          INIT: in  std_logic_vector(5 downto 0);
          Y1: out std_logic_vector(3 downto 0) := "0000";
          Y2: out std_logic_vector(3 downto 0) := "0000";
          EN: out std_logic
    );
end cont_sm;

architecture rtl of cont_sm is

```

```
signal TY1, TY2: std_logic_vector(3 downto 0) := "0000";
signal count: integer;
signal temp: integer;

begin

    -- Contatore modulo 60
    count_sec_min: process( CLK, RST )
    begin
        if( RST = '1' ) then
            count <= 0;
            EN <= '0';
        elsif( SET = '1' and ore_en = '1' ) then
            temp <= to_integer(unsigned(INIT));
            if (temp >= 60) then
                count <= 0;
            else
                count <= temp;
            end if;
            EN <= '0';
        elsif( CLK'event and CLK = '1' ) then
            if( ore_en ='1') then
                count <= count;
            elsif( count = 59 ) then
                count <= 0;
                EN <= '1';
            else
                count <= count + 1;
                EN <= '0';
            end if;
        end if;
    end process;

    -- Uscita del contatore
    -- si suddividono due uscite, una per la cifra 1(TY1) e cifra 2 (TY2)
    opportunamente con
    -- la divisione di 10 e il modulo di 10 in maniera da mostrare
    l'orario in formato decimale
    TY1 <= std_logic_vector(to_unsigned(count/10, 4));
    TY2 <= std_logic_vector(to_unsigned(count mod 10, 4));
    Y1 <= TY1;
    Y2 <= TY2;
end rtl;
```

In questo modo è possibile andare a bloccare il tempo poiché il *count* viene aggiornato con il valore di sé stesso.

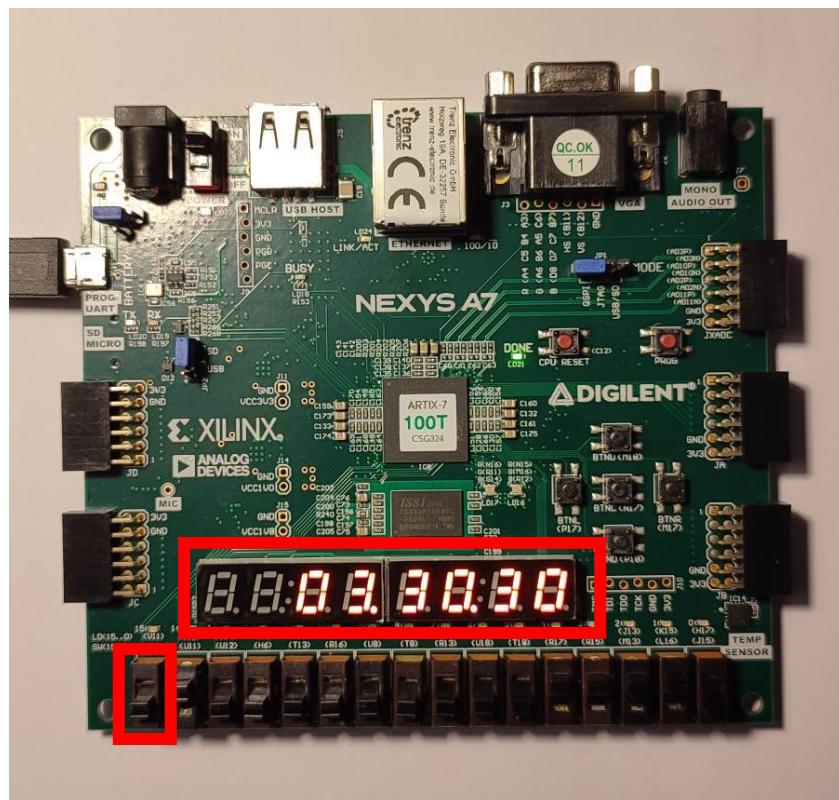


FIG 5.12 Modalità orologio, segnalazione 3:30:30

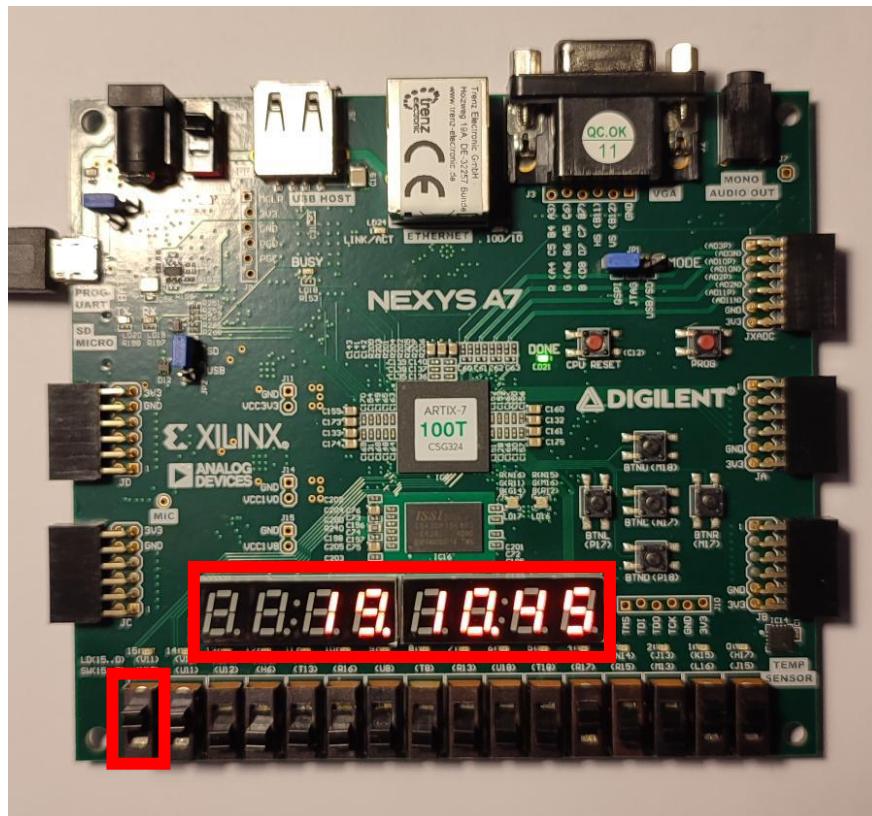


FIG 5.13 Modalità intertempo, salvataggio 19:10:45

5.4.3 Considerazioni finali e possibile scopo futuro

Analizzando il progetto, si può notare che estendendo in termini funzionali il *cronometro* si è dovuto ricorrere ad un incremento di porte e circuiti. Si può notare questo negli schematici nelle **FIG 5.7** e **FIG 5.11**. Questo perché, provando a “parallelizzare” l’inserimento dell’orario si è dovuto ricorrere a più bottoni (più hardware).

È possibile notare questa differenza anche in termini temporali analizzando opportunamente la timing analysis. Infatti, dall’analisi dello **WNS** al fine di verificare lo *slack* nel caso peggiore. Se il WNS è positivo allora vuol dire che tutti i vincoli sono stati rispettati altrimenti vorrebbe dire che il longest path ha un delay maggiore rispetto al periodo del clock.

Design Timing Summary							
WNS(ns)	TNS(ns)	TNS Failing Endpoints	TNS Total Endpoints	WHS(ns)	THS(ns)	THS Failing Endpoints	THS Total Endpoints
2.637	0.000	0	458	0.115	0.000	0	458

All user specified timing constraints are met.

Clock Summary			
Clock	Waveform(ns)	Period(ns)	Frequency (MHz)
sys_clk_pin	{0.000 5.000}	10.000	100.000

FIG 5.13 WNS del cronometro_on_board versione del paragrafo 5.3

Design Timing Summary							
WNS(ns)	TNS(ns)	TNS Failing Endpoints	TNS Total Endpoints	WHS(ns)	THS(ns)	THS Failing Endpoints	THS Total Endpoints
2.751	0.000	0	629	0.127	0.000	0	629

All user specified timing constraints are met.

Clock Summary			
Clock	Waveform(ns)	Period(ns)	Frequency (MHz)
sys_clk_pin	{0.000 5.000}	10.000	100.000

FIG 5.14 WNS del cronometro_on_board versione del paragrafo 5.4

Nello stesso file di Timing Summary c’è una sezione di analisi dello *Slack*, (la differenza tra clock desiderato ed effettivo) se è positivo, il sistema riesce ad eseguire e a terminare le sue operazioni nel periodo prestabilito altrimenti sarà necessario modificare il periodo aumentandolo per permettere la corretta conclusione di tutte le operazioni. È possibile analizzare lo *slack* anche nella sezione del *Max Delay Path*, ovvero nel caso del percorso più lungo all’interno del sistema, tra le varie porte che sono state mappate.



CRONOMETRO

```

-----+
| Timing Details
| -----
-----+
From Clock: sys_clk_pin
To Clock: sys_clk_pin

Setup :      0 Failing Endpoints, Worst Slack      2.637ns, Total Violation      0.000ns
Hold :       0 Failing Endpoints, Worst Slack      0.115ns, Total Violation      0.000ns
PW :        0 Failing Endpoints, Worst Slack      4.500ns, Total Violation      0.000ns
-----+

```

Max Delay Paths

```

Slack (MET) :      2.637ns (required time - arrival time)
Source:          it/uc/mem_temp_reg/C
                  (rising edge-triggered cell FDCE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Destination:     it/mem_temps[0].reg_temp/y_temp_reg[20]/D
                  (falling edge-triggered cell FDCE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Path Group:      sys_clk_pin
Path Type:       Setup (Max at Slow Process Corner)
Requirement:    5.000ns (sys_clk_pin fall@5.000ns - sys_clk_pin rise@0.000ns)
Data Path Delay: 2.315ns (logic 0.642ns (27.732%) route 1.673ns (72.268%))
Logic Levels:   1 (LUT5=1)
Clock Path Skew: -0.045ns (DCD - SCD + CPR)
Destination Clock Delay (DCD): 4.947ns = ( 9.947 - 5.000 )
Source Clock Delay (SCD): 5.248ns
Clock Pessimism Removal (CPR): 0.256ns
Clock Uncertainty: 0.035ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
Total System Jitter (TSJ): 0.071ns
Total Input Jitter (TIJ): 0.000ns
Discrete Jitter (DJ): 0.000ns
Phase Error (PE): 0.000ns
-----+

```

FIG 5.15 Slack del cronometro_on_board versione del paragrafo 5.3

```

-----+
| Timing Details
| -----
-----+
From Clock: sys_clk_pin
To Clock: sys_clk_pin

Setup :      0 Failing Endpoints, Worst Slack      2.751ns, Total Violation      0.000ns
Hold :       0 Failing Endpoints, Worst Slack      0.127ns, Total Violation      0.000ns
PW :        0 Failing Endpoints, Worst Slack      4.500ns, Total Violation      0.000ns
-----+

```

Max Delay Paths

```

Slack (MET) :      2.751ns (required time - arrival time)
Source:          it/uc/op_temp_reg/C
                  (rising edge-triggered cell FDCE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Destination:     it/mem_temps[0].reg_temp/y_temp_reg[12]/D
                  (falling edge-triggered cell FDCE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Path Group:      sys_clk_pin
Path Type:       Setup (Max at Slow Process Corner)
Requirement:    5.000ns (sys_clk_pin fall@5.000ns - sys_clk_pin rise@0.000ns)
Data Path Delay: 2.223ns (logic 0.642ns (28.880%) route 1.581ns (71.120%))
Logic Levels:   1 (LUT5=1)
Clock Path Skew: -0.025ns (DCD - SCD + CPR)
Destination Clock Delay (DCD): 5.030ns = ( 10.030 - 5.000 )
Source Clock Delay (SCD): 5.232ns
Clock Pessimism Removal (CPR): 0.177ns
Clock Uncertainty: 0.035ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
Total System Jitter (TSJ): 0.071ns
Total Input Jitter (TIJ): 0.000ns
Discrete Jitter (DJ): 0.000ns
Phase Error (PE): 0.000ns
-----+

```

FIG 5.16 Slack del cronometro_on_board versione del paragrafo 5.4



Da questa analisi si può notare come si sia incrementato anche di poco lo *slack time* dovuto proprio dall'aumento del path più lungo tra le porte mappate.

Attenzione.

Osservando in secondo momento la progettazione, salta all'occhio altre possibili soluzioni per la gestione degli intertempi. Nel caso d'esame, come scritto nella documentazione, la soluzione prevede un numero di memorie proporzionale ad n . Nel nostro caso, in cui si è pensato di usare un $N = 2$ è stato possibile scegliere questa soluzione ma nel momento in cui le richieste del committente siano un N alquanto grande le cose si complicano. Si sarebbe potuto ricorrere ad una memoria di dimensione N che veniva aggiornata ad ogni pressione del button stop e memorizzare gli intertempi ed in secondo momento scandire con un contatore per la visualizzazione, ma anche questa soluzione può destare problematiche in quanto serve aggiungere un altro componente di indirizzamento.

6 Sistema di testing

Si realizza un Sistema di Testing seguendo le specifiche richieste dal cliente:

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

6.1 Progettazione del Sistema di Testing

SYSTEM TESTING

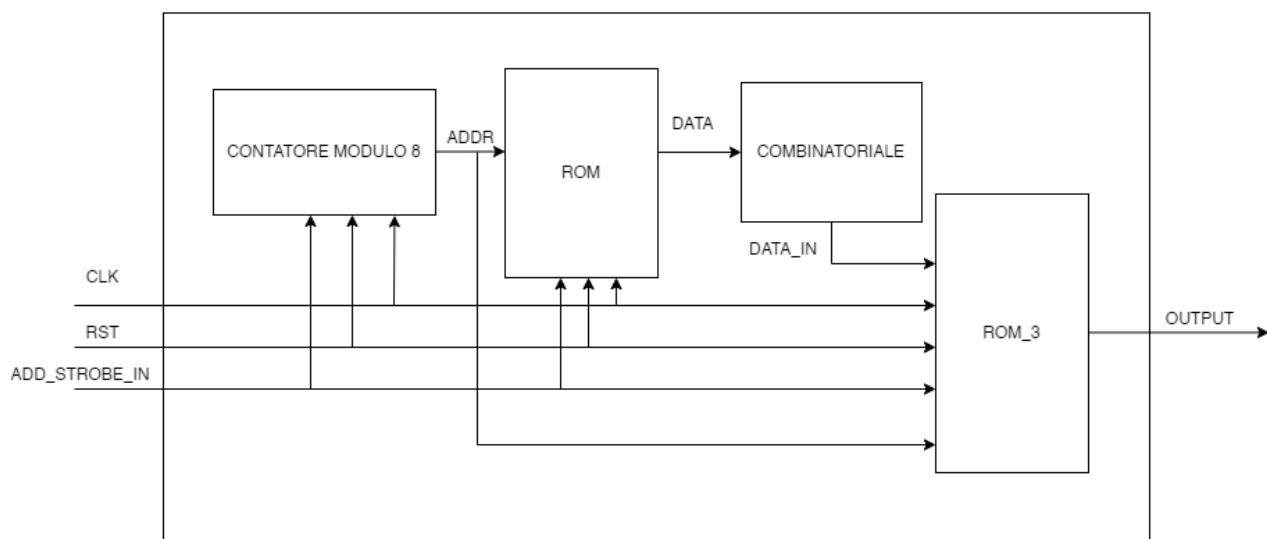


FIG 6.1 Schematico generale

6.1.1 Funzionamento

Il sistema di testing ha come compito di calcolare il numero di bit alzati dei valori all'interno della ROM, il sistema è composto da 4 elementi:

- contatore modulo 8
- una ROM
- blocco combinatorio
- una memoria interna, chiamata impropriamente ROM_3

Il **contatore modulo 8** ha in ingresso il clock, il reset e l'*addr_strobe_in* che corrisponde all'*enable*, in uscita invece produce il valore del conteggio che va in ingresso alla ROM.

La **ROM** ha in ingresso il clock, il reset, l'addr_strobe_in e il conteggio; quest'ultimo segnale viene convertito in intero e si riferisce all'indirizzo della ROM al quale bisogna prelevare il dato. Nella ROM sono stati inseriti 8 segnali di 4 bit che andranno in input al blocco combinatoriale.

Il **blocco combinatoriale** ha in input il valore che si trova all'indirizzo ADDR della ROM che è un valore di 4 bit e produce in uscita un segnale di 3 bit in base all'input in ingresso.

Infine, abbiamo la **memoria interna (ROM_3)** che ha in ingresso il clock, il reset, l'addr_strobe_in, address e DATA-IN (il dato uscente dalla macchina combinatoriale); il suo compito è di salvare nella memoria interna, all'indirizzo fornito in ingresso, il valore prodotto dal blocco combinatorio e di dare in uscita lo stesso valore.

6.1.2 Codice VHDL

Andiamo a vedere singolarmente tutti i componenti del sistema, partendo dalla memoria ROM.

ROM.vhd

```
ENTITY ROM IS
    PORT (
        CLK : IN STD_LOGIC; -- clock della board
        RST : IN STD_LOGIC;
        READ : IN STD_LOGIC; -- segnale che abilita la lettura,
inserito tramite un bottone
        ADDR : IN STD_LOGIC_VECTOR(2 DOWNTO 0); --3 bit di indirizzo
per accedere agli elementi della ROM,
        DATA : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) -- dato su 4 bit letto
dalla ROM
    );
END ROM;
```

ARCHITECTURE Behavioral OF ROM IS

```
TYPE rom_type IS ARRAY (7 DOWNTO 0) OF STD_LOGIC_VECTOR(3 DOWNTO
0);
SIGNAL ROM : rom_type := (
    "0000",
    "0001",
    "0010",
    "0011",
    "0100",
    "0101",
    "0110",
    "0111"
);
```

Come si può vedere, il blocco di memoria ROM è inizializzato a dei valori di 4 bit arbitrari. Ad ogni colpo di clock, il process di cui è composta non fa altro che controllare se il segnale di reset è alto, se sì, mette in uscita un vettore di tutti 0, altrimenti mette il valore presente all'interno dell'indirizzo specificato nel segnale ADDR in ingresso.

Il secondo componente è la macchina combinatoriale, che, come detto, non fa altro che dare in uscita il numero di bit alti presenti nella stringa in ingresso.

Combinatorial.vhd

```
ENTITY Combinatorial IS
    PORT (
        input : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        output : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
    );
END Combinatorial;

ARCHITECTURE Behavioral OF Combinatorial IS

BEGIN

    output <= "000" WHEN input = "0000" ELSE
        "001" WHEN input = "0001" ELSE
        "001" WHEN input = "0010" ELSE
        "010" WHEN input = "0011" ELSE
        "001" WHEN input = "0100" ELSE
        "010" WHEN input = "0101" ELSE
        "010" WHEN input = "0110" ELSE
        "011" WHEN input = "0111" ELSE
        "001" WHEN input = "1000" ELSE
        "010" WHEN input = "1001" ELSE
        "010" WHEN input = "1010" ELSE
        "011" WHEN input = "1011" ELSE
        "010" WHEN input = "1100" ELSE
        "011" WHEN input = "1101" ELSE
        "011" WHEN input = "1110" ELSE
        "100" WHEN input = "1111" ELSE
        "---" when input ="---";

END Behavioral;
```

Successivamente abbiamo il contatore.

Counter_mod8.vhd

```
ENTITY counter_mod8 IS
    PORT (
        clock : IN STD_LOGIC;
        reset : IN STD_LOGIC;
        enable : IN STD_LOGIC;
        counter : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END counter_mod8;

ARCHITECTURE Behavioral OF counter_mod8 IS

    SIGNAL c : STD_LOGIC_VECTOR (2 DOWNTO 0) := (OTHERS => '0');
BEGIN
    counter <= c;

    counter_process : PROCESS (clock)
BEGIN
```

```

IF (rising_edge(clock)) THEN
    IF reset = '1' THEN
        c <= (OTHERS => '0');
    ELSIF enable = '1' THEN
        c <= STD_LOGIC_VECTOR(unsigned(c) + 1);
    END IF;
END IF;
END PROCESS;

END Behavioral;
ENTITY counter_mod8 IS
    PORT (
        clock : IN STD_LOGIC;
        reset : IN STD_LOGIC;
        enable : IN STD_LOGIC;
        counter : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END counter_mod8;

ARCHITECTURE Behavioral OF counter_mod8 IS

    SIGNAL c : STD_LOGIC_VECTOR (2 DOWNTO 0) := (OTHERS => '0');

BEGIN
    counter <= c;

    counter_process : PROCESS (clock)
    BEGIN

        IF (rising_edge(clock)) THEN
            IF reset = '1' THEN
                c <= (OTHERS => '0');
            ELSIF enable = '1' THEN
                c <= STD_LOGIC_VECTOR(unsigned(c) + 1);
            END IF;
        END IF;
    END PROCESS;

END Behavioral;

```

Poi c'è il blocco di memoria interna, chiamato impropriamente ROM_3.

ROM_3.vhd

```

ENTITY ROM_3 IS
    PORT (
        CLK : IN STD_LOGIC; -- clock della board
        RST : IN STD_LOGIC;
        READ : IN STD_LOGIC; -- segnale che abilita la lettura,
        inserito tramite un bottone
        ADDR : IN STD_LOGIC_VECTOR(2 DOWNTO 0); --3 bit di indirizzo
        per accedere agli elementi della ROM,
        DATA_IN : IN STD_LOGIC_VECTOR(2 DOWNTO 0); --dato in ingresso
        da memorizzare

```



```

        DATA_OUT : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
    );
END ROM_3;

ARCHITECTURE Behavioral OF ROM_3 IS

    TYPE rom_type IS ARRAY (7 DOWNTO 0) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ROM : rom_type;

    ATTRIBUTE rom_style : STRING;
    ATTRIBUTE rom_style OF ROM : SIGNAL IS "block";-- block dice al
tool di sintesi di inferire blocchi di RAMB,
-- distributed di usare le LUT

BEGIN

PROCESS (CLK)
BEGIN
    IF rising_edge(CLK) THEN
        IF (RST = '1') THEN
            ROM(0) <= "000";
            ROM(1) <= "000";
            ROM(2) <= "000";
            ROM(3) <= "000";
            ROM(4) <= "000";
            ROM(5) <= "000";
            ROM(6) <= "000";
            ROM(7) <= "000";
            DATA_OUT <= ROM(conv_integer("000"));

        ELSIF (READ = '1') THEN
            ROM(conv_integer(ADDR - 1)) <= DATA_IN;
            DATA_OUT <= ROM(conv_integer(ADDR - 2));
        END IF;
    END IF;
END PROCESS;
END Behavioral;

```

Questo blocco non fa altro che memorizzare il dato in ingresso. Si noti che la memoria interna riceve in ingresso come indirizzo il valore di conteggio del contatore, ma non memorizza il dato esattamente a quell'indirizzo, perché, volendo fare in modo che il numero di bit alti della i-esima stringa presente nella ROM fosse memorizzato esattamente allo stesso indirizzo i-esimo della memoria interna, si è dovuto tenere conto dei ritardi del Sistema. Più nello specifico, come si può vedere dallo schematico, il valore di conteggio (l'indirizzo) viene dato simultaneamente sia alla ROM che alla memoria interna, ma per fare in modo che la memorizzazione del risultato i-esimo avvenisse allo stesso indirizzo i della stringa i-esima, la macchina combinatoria avrebbe dovuto lavorare in tempo 0, cosa ovviamente impossibile. Un'altra soluzione possibile sarebbe stata quella di prevedere un secondo contatore che andasse a scandire gli indirizzi in cui avrebbe dovuto memorizzare la memoria interna.

Infine, c'è il top module System Testing. Si sottolinea il fatto che il sistema effettua una intera operazione (ovvero prelievo del dato dalla ROM, calcolo del valore dei bit alti e memorizzazione) solo quando viene dato in ingresso il segnale di abilitazione addr_strobe_in.

TestingSystem.vhd

```

ENTITY TestingSystem IS
    PORT (
        clock_in : STD_LOGIC;
        reset_in : IN STD_LOGIC;
        addr_strobe_in : IN STD_LOGIC; -- bottone che abilita la
selezione della cella della ROM da visualizzare
        output : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
    );
END TestingSystem;

ARCHITECTURE Structural OF TestingSystem IS

    COMPONENT ROM IS
        PORT (
            CLK : IN STD_LOGIC; -- clock della board
            RST : IN STD_LOGIC;
            READ : IN STD_LOGIC; -- segnale che abilita la lettura,
inserito tramite un bottone
            ADDR : IN STD_LOGIC_VECTOR(2 DOWNTO 0); -- 3 bit di
indirizzo per accedere agli elementi della ROM,
-- sono inseriti tramite gli switch
            DATA : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) -- dato su 4 bit
letto dalla ROM
        );
    END COMPONENT;

    COMPONENT Combinatorial IS
        PORT (
            input : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
            output : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT ROM_3 IS
        PORT (
            CLK : IN STD_LOGIC; -- clock della board
            RST : IN STD_LOGIC;
            READ : IN STD_LOGIC; -- segnale che abilita la lettura,
inserito tramite un bottone
            ADDR : IN STD_LOGIC_VECTOR(2 DOWNTO 0); -- 3 bit di
indirizzo per accedere agli elementi della ROM,
            DATA_IN : IN STD_LOGIC_VECTOR(2 DOWNTO 0); -- dato in
ingresso da memorizzare
            DATA_OUT : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT counter_mod8 IS
        PORT (
            clock : IN STD_LOGIC;
            reset : IN STD_LOGIC;
            enable : IN STD_LOGIC;
            counter : OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
        );

```

```
END COMPONENT;

SIGNAL reset_n, read_strobe : STD_LOGIC;
SIGNAL data_temp_1 : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL data_temp_2 : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL data_temp_out : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL addr : STD_LOGIC_VECTOR(2 DOWNTO 0);

BEGIN

    reset_n <= reset_in;
    mem_1 : ROM
    PORT MAP(
        clk => clock_in,
        rst => reset_n,
        read => addr_strobe_in,
        addr => addr,
        data => data_temp_1
    );

    mac_comb : Combinatorial
    PORT MAP(
        input => data_temp_1,
        output => data_temp_2
    );

    mem_2 : ROM_3
    PORT MAP(
        clk => clock_in,
        rst => reset_n,
        read => addr_strobe_in,
        addr => addr,
        data_in => data_temp_2,
        data_out => output
    );

    counter : counter_mod8
    PORT MAP(
        clock => clock_in,
        reset => reset_n,
        enable => addr_strobe_in,
        counter => addr
    );

```

END Structural;

6.1.3 Test Bench

TestingSystemTB.vhd

```

ENTITY TestingSystem_TB IS
    -- Port ( );
END TestingSystem_TB;

ARCHITECTURE Behavioral OF TestingSystem_TB IS

    COMPONENT TestingSystem IS
        PORT (
            clock_in : STD_LOGIC;
            reset_in : IN STD_LOGIC;
            addr_strobe_in : IN STD_LOGIC; -- bottone che abilita la
selezione della cella della ROM da visualizzare
            -- address_in : in STD_LOGIC_VECTOR(2 downto 0); --
indirizzo di selezione inserito tramite switch
            output : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
        );
    END COMPONENT;

    CONSTANT CLK_period : TIME := 10 ns;

    SIGNAL clk : STD_LOGIC;
    SIGNAL read : STD_LOGIC;
    SIGNAL reset : STD_LOGIC;
    SIGNAL data : STD_LOGIC_VECTOR(2 DOWNTO 0);

BEGIN

    uut : TestingSystem PORT MAP(
        clock_in => clk,
        reset_in => reset,
        addr_strobe_in => read,
        output => data);
    stimulus : PROCESS
    BEGIN

        WAIT FOR 100ns;
        read <= '1';
        WAIT FOR 300ns;
        read <= '0';
        -- reset <= '1';

    END PROCESS;
    CLK_process : PROCESS
    BEGIN
        CLK <= '0';
        WAIT FOR CLK_period/2;
        CLK <= '1';
        WAIT FOR CLK_period/2;
    END PROCESS;

END Behavioral;

```

6.1.4 La simulazione del Sistema di Testing

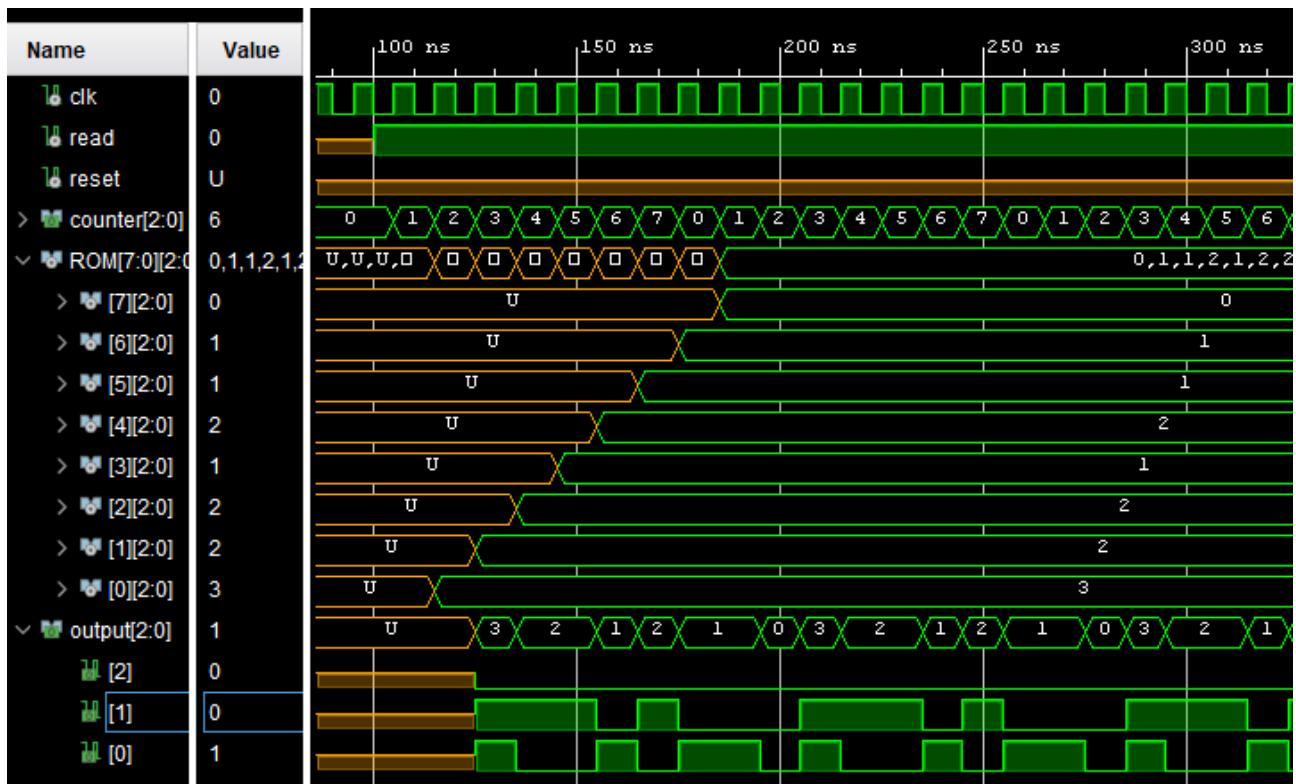


FIG 6.2 Simulazione

6.2 Implementazione sulla board

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

6.2.1 Modifiche

Abbiamo inserito 2 debouncer uno per il bottone di enable (*addr_strobe_in*) e uno per il bottone di reset. Abbiamo abilitato sul file *constraints* della board data in dotazione, il clock e i due bottoni. Infine, abbiamo implementato i 2 debouncer attraverso il generic nel top module, cioè nel system testing.

6.2.2 Codice VHDL

Del System Tesing riportiamo solamente i port map.

TestingSystem.vhd

```

SIGNAL reset_n : STD_LOGIC;
SIGNAL read_strobe : STD_LOGIC;
SIGNAL reset_strobe : STD_LOGIC;
SIGNAL data_temp_1 : STD_LOGIC_VECTOR(3 DOWNTO 0);

```

```
SIGNAL data_temp_2 : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL data_temp_out : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL addr : STD_LOGIC_VECTOR(2 DOWNTO 0);

BEGIN

reset_n <= reset_in; --visto che utilizzo il bottone CPU_reset della
board, che è attivo-basso,
--devo convertire il segnale di reset
b_deb_reset : ButtonDebouncer
GENERIC MAP(
    CLK_period => 10, -- periodo del clock della board pari a 10ns
    btn_noise_time => 500000000 --intervallo di tempo in cui si ha
l'oscillazione del bottone
    --assumo che duri 500ms=500000microsec=500000000ns
)
PORT MAP(
    rst => '0',
    clk => clock_in,
    btn => reset_n,
    cleared_btn => reset_strobe
);
b_deb_read : ButtonDebouncer
GENERIC MAP(
    CLK_period => 10, -- periodo del clock della board pari a 10ns
    btn_noise_time => 500000000 --intervallo di tempo in cui si ha
l'oscillazione del bottone
    --assumo che duri 500ms=500000microsec=500000000ns
)
PORT MAP(
    rst => reset_strobe,
    clk => clock_in,
    btn => addr_strobe_in,
    cleared_btn => read_strobe
);

mem_1 : ROM
PORT MAP(
    clk => clock_in,
    rst => reset_strobe,
    read => read_strobe,
    addr => addr,
    data => data_temp_1
);

mac_comb : Combinatorial
PORT MAP(
    input => data_temp_1,
    output => data_temp_2
);

mem_2 : ROM_3
PORT MAP(
    clk => clock_in,
    rst => reset_strobe,
    read => read_strobe,
```



```

    addr => addr,
    data_in => data_temp_2,
    data_out => output
);

counter : counter_mod8
PORT MAP(
    clock => clock_in,
    reset => reset_strobe,
    enable => read_strobe,
    counter => addr
);

END Behavioral;

```

ButtonDebouncer.vhd

```

ENTITY ButtonDebouncer IS
    GENERIC (
        CLK_period : INTEGER := 10; -- periodo del clock della board
10 nanosecondi
        btn_noise_time : INTEGER := 6500000 --intervallo di tempo in
cui si ha l'oscillazione del bottone
        --assumo che duri 6.5ms=6500microsec=6500000ns
    );
    PORT (
        RST : IN STD_LOGIC;
        CLK : IN STD_LOGIC;
        BTN : IN STD_LOGIC;
        CLEARED_BTN : OUT STD_LOGIC);
END ButtonDebouncer;

```

ARCHITECTURE Behavioral **OF** ButtonDebouncer **IS**

```

    -- questo componente prende in input il segnale proveniente dal
    bottone e genera un
    -- segnale "ripulito" che presenta un impulso della durata di un
    colpo di clock per
    -- segnalare l'avvenuta pressione del bottone.
    -- ATTENZIONE: per questo progetto non sarebbe necessario, in
    quanto il bottone viene
    -- usato solo per dare un enable all'indirizzo di selezione della
    ROM (eventuali oscillazioni
    -- che risultino in più pressioni successive non modificherebbero
    la logica del sistema)
    -- Il debouncer implementa un semplice automa di 2 stati
    -- si parte da NOT_PRESSED e, appena si rileva BTN=1, si va in
    PRESSSED dove
    -- si aspettano 6.5 millisecondi in modo da "superare"
    l'oscillazione

```

```

TYPE stato IS (NOT_PRESSED, PRESSED);
SIGNAL BTN_state : stato := NOT_PRESSED;

```

```

CONSTANT max_count : INTEGER := btn_noise_time/CLK_period; --
6500000/10= conto 650000 colpi di clock

```



BEGIN

```

deb : PROCESS (CLK)
  VARIABLE count : INTEGER := 0;

BEGIN
  IF rising_edge(CLK) THEN

    IF (RST = '1') THEN
      BTN_state <= NOT_PRESSED;
      CLEARED_BTN <= '0';
    ELSE
      CASE BTN_state IS
        WHEN NOT_PRESSED =>
          CLEARED_BTN <= '0';
          IF (BTN = '1') THEN
            BTN_state <= PRESSED;
          ELSE
            BTN_state <= NOT_PRESSED;
          END IF;
        WHEN PRESSED =>
          IF (count = max_count - 1) THEN
            count := 0;
            CLEARED_BTN <= '1';
            BTN_state <= NOT_PRESSED;
          ELSE
            count := count + 1;
            BTN_state <= PRESSED;
          END IF;
        WHEN OTHERS =>
          BTN_state <= NOT_PRESSED;
      END CASE;
    END IF;
  END PROCESS;
END Behavioral;

```

Constraints

```

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

##Buttons
set_property -dict { PACKAGE_PIN M18      IOSTANDARD LVCMOS33 } [get_ports {
reset_in }];
#IO_L4N_T0_D05_14 Sch=btnu
set_property -dict { PACKAGE_PIN P18      IOSTANDARD LVCMOS33 } [get_ports {
addr_strobe_in }];
#IO_L9N_T1_DQS_D13_14 Sch=btnd

```



6.2.3 Test sulla Board

ADDR = 0 , OUTPUT = 3



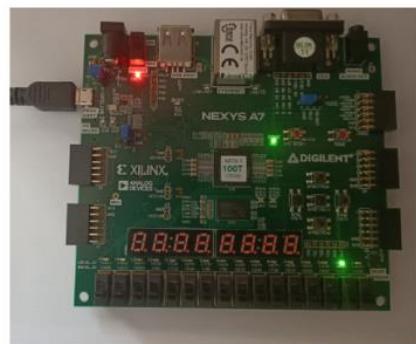
ADDR = 1 , OUTPUT = 2



ADDR = 2 , OUTPUT = 2



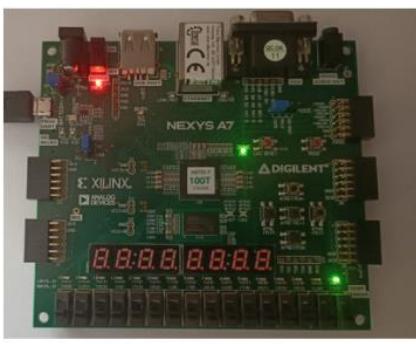
ADDR = 3 , OUTPUT = 1



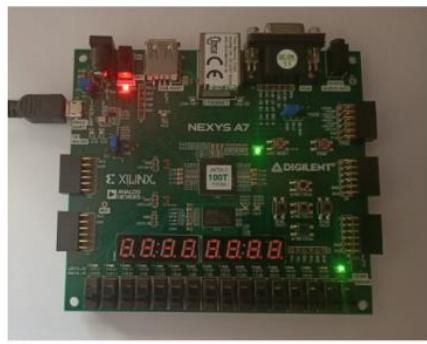
ADDR = 4 , OUTPUT = 2



ADDR = 5 , OUTPUT = 1



ADDR = 6, OUTPUT = 1



ADDR = 7 , OUTPUT = 0

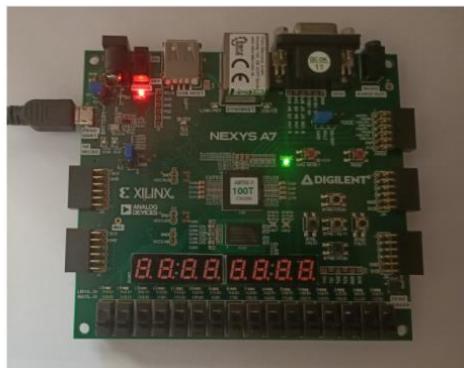


FIG. 6.3

Nel caso in cui volessimo si volesse generare una sequenza di input senza premere il bottone per n volte è possibile assegnare il segnale di abilitazione ad uno switch, così da generare una sequenza di input quando lo switch è alto.

##Switches

```
set_property -dict { PACKAGE_PIN J15    IOSTANDARD LVCMOS33 } [get_ports {  
addr_strobe_in }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
```

7 Handshaking

Si realizza un protocollo di Handshaking seguendo le specifiche richieste dal cliente:

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

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

7.1 Progettazione Handshaking tra due entità

Occorre affrontare come due entità possano interagire tra loro attraverso un protocollo (accordo) tra le due parti tale da poter permetterne l'interazione. Esistono diversi approcci per poter risolvere tale problematica tra le due entità, il nostro è stato quello di applicarlo in un caso generale e reale. Ovvvero l'interazione tra due entità che abbiano due *clock* differenti. Pare normale che dispositivi differenti abbiano *clock* differenti in quanto ogni dispositivo ha caratteristiche costruttive dissimili. Pertanto, si è lavorato su un protocollo *asincrono*, ovvero le entità comunicano a valle della ricezione di eventi che finché non si verificano non trasmettono. Ad esempio, siano due entità A e B (che si prenderanno in considerazione nell'esecuzione dell'esercizio), se A vuole inviare un'informazione a B c'è la necessità di accordarsi, in modo che, A manda il dato e dà il via alla comunicazione, B risponde di aver ricevuto il dato (**Handshaking**) **FIG 7.1**.

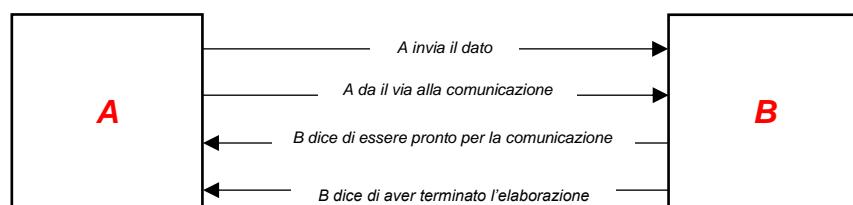


FIG 7.1 Protocollo Handshaking interlacciato

Il progetto, dunque, prevede due entità (*Entity_A*) ed (*Entity_B*) “separate” (per dare un senso di diversificazione). Entrambe prevedono una memoria ROM (Read Only Memory) “precaricata” con $N = 8$ stringhe da $M = 8$ bit nel quale verranno scandite da un *contatore* opportunamente pilotato da un’*unità di controllo*. Quest’ultimo sarà utile per disciplinare la trasmissione/ricezione delle stringhe

e per terminare la comunicazione. Inoltre, il committente richiede che l'entità B abbia la capacità di sommare la stringa i-esima in ingresso con la stringa i-esima presente nella propria ROM e salvarla in una ulteriore memoria locale che in questo caso funzionerà da memoria di lettura e scrittura. Per attuare la somma, l'entità B, oltre all'unità di controllo, prevede un'*unità operativa* utile proprio per l'elaborazione della somma.

Per gestire il protocollo di *Handshaking* si è scelto di realizzare un componente “*interface*” che funzionerà da arbitro per evitare possibili errori durante la comunicazione tra le due entità. Si è scelta questa soluzione poiché è possibile estenderla nel momento in cui i due sistemi non abbiano lo stesso parallelismo nel trasferimento dei dati (ad esempio A comunica tramite un bus di 8 bit e B legge da un bus di 16 bit). Nel caso in esame, A comunica tramite un bus di 8 bit e B legge da un bus di 8 bit.

Si riporta qui uno schematico generale:

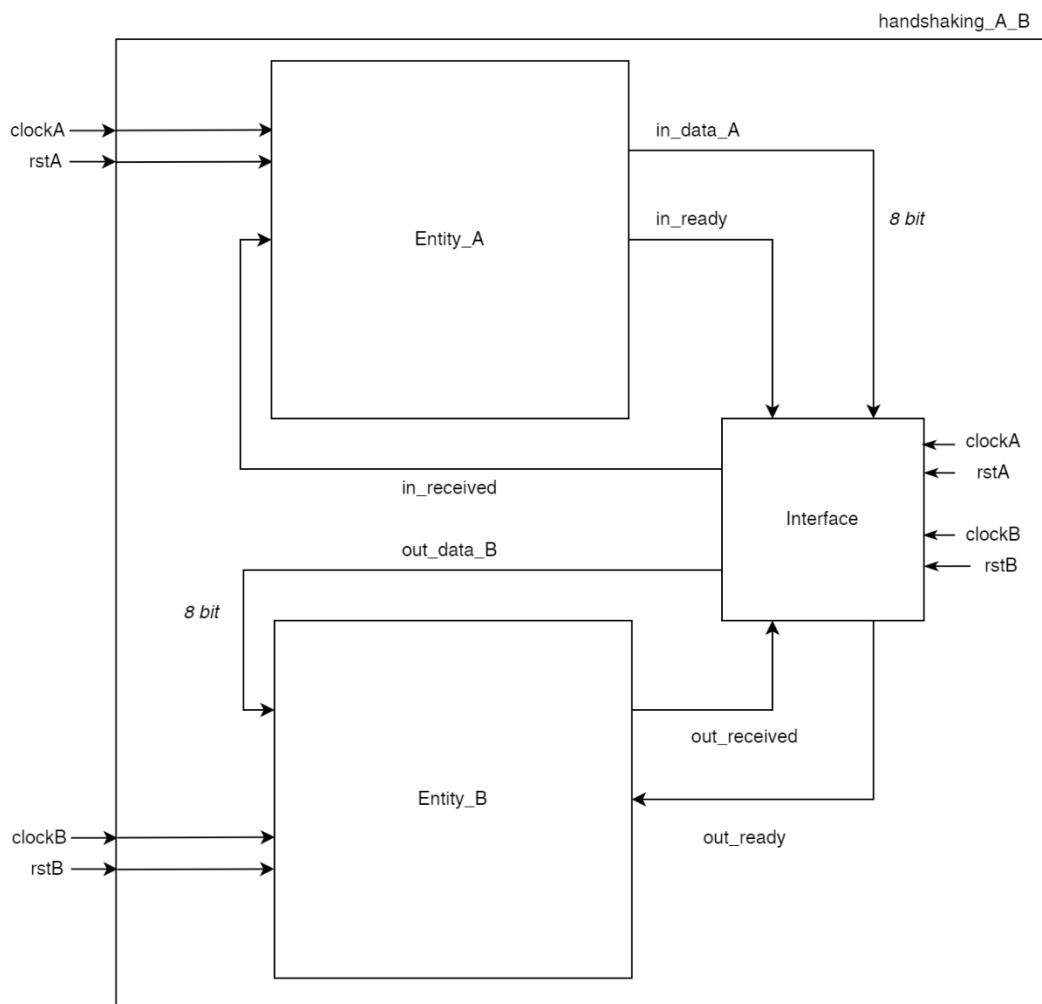


FIG 7.2 Schematico generale di interazione Entity_A, Entity_B e interface

7.1.1 Codice VHDL dell'Handshaking

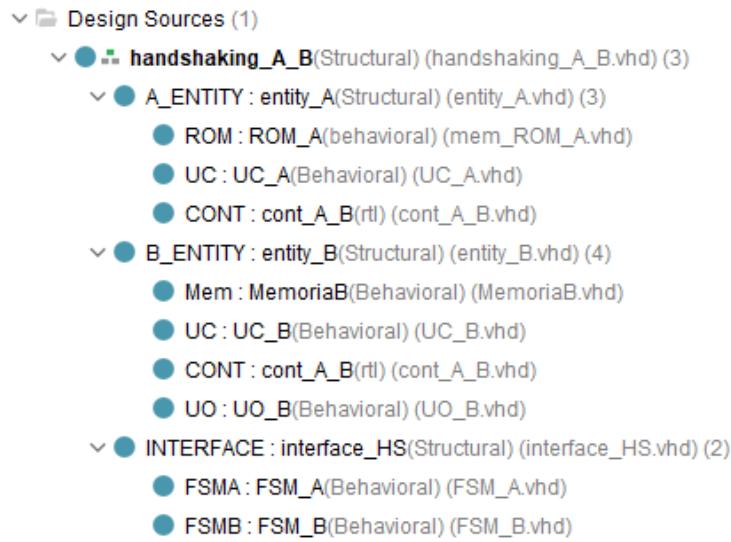


FIG 7.3 Design Sources Handshaking_A_B

Dalla **FIG 7.2** è possibile dunque descrivere a livello strutturale il top-module del progetto in esame. Si riporta il codice dell'architettura.

TOP-MODULE *handshaking_A_B.vhd*

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity handshaking_A_B is
  generic (
    N : integer := 8;
    M : integer := 8
  );
  Port (
    clk_A : in std_logic;
    clk_B : in std_logic;
    rst_A : in std_logic;
    rst_B : in std_logic
  );
end handshaking_A_B;

architecture Structural of handshaking_A_B is

-- // COMPONENTI

component entity_A
  generic (
    N : integer := 8;
    M : integer := 8
  );
  
```

```

);
Port (
    clk_A : in STD_LOGIC;
    rst_A : in STD_LOGIC;
    --- dall'interfaccia
    in_received : in STD_LOGIC;
    in_data : out std_logic_vector(M-1 downto 0);
    in_ready : out std_logic
);
end component;

component entity_B
generic (
    N : integer := 8;
    M : integer := 8
);
Port (
    clk_B : in STD_LOGIC;
    rst_B : in STD_LOGIC;
    --- dall'interfaccia
    out_received : out STD_LOGIC;
    out_data : in std_logic_vector(M-1 downto 0);
    out_ready : in std_logic
);
end component;

component interface_HS
generic (
    N : integer := 8;
    M : integer := 8
);
Port (
    CLK_A : in std_logic;
    RST_A : in std_logic;
    CLK_B : in std_logic;
    RST_B : in std_logic;
    in_received : out STD_LOGIC;
    in_data : in std_logic_vector(M-1 downto 0);
    in_ready : in std_logic;
    out_received : in STD_LOGIC;
    out_data : out std_logic_vector(M-1 downto 0);
    out_ready : out std_logic
);
end component;

--segnali interni
signal indata : std_logic_vector(M-1 downto 0);
signal outdata : std_logic_vector(M-1 downto 0);
signal inready, inreceived, outready, outreceived : std_logic;

```



```
-- // MAPPING
begin

    A_ENTITY : entity_A
    generic map (
        M => 8,
        N => 8
    )
    port map (
        clk_A => clk_A,
        rst_A => rst_A,
        in_received => inreceived,
        in_data => indata,
        in_ready => inready
    );
    
    B_ENTITY : entity_B
    generic map (
        M => 8,
        N => 8
    )
    port map (
        clk_B => clk_B,
        rst_B => rst_B,
        out_received => outreceived,
        out_data => outdata,
        out_ready => outready
    );
    
INTERFACE : interface_HS
generic map(
    N => 8,
    M => 8
)
Port map(
    CLK_A => clk_A,
    RST_A => rst_A,
    CLK_B => clk_B,
    RST_B => rst_B,
    in_received => inreceived,
    in_data => indata,
    in_ready => inready,
    out_received => outreceived,
    out_data => outdata,
    out_ready => outready
);

```

```
end Structural;
```

Si descrivono strutturalmente le due entità:

Entity_A.vhd

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

entity entity_A is
    generic (
        N : integer := 8;
        M : integer := 8
    );
    Port (
        clk_A : in STD_LOGIC;
        rst_A : in STD_LOGIC;
        --dall'interfaccia
        in_received : in STD_LOGIC := '0';
        --verso l'interfaccia
        in_data : out std_logic_vector(M-1 downto 0);
        in_ready : out std_logic := '0'
    );
end entity_A;

architecture Structural of entity_A is

-- // COMPONENTI

component ROM_A
generic (
    M : integer := 8
);
PORT (
    CLK : in std_logic;
    RST: in std_logic;
    READ : in std_logic := '0'; -- segnale da uc
    ADDR : in std_logic_vector(2 downto 0); --3 bit di indirizzo per
    accedere agli elementi della ROM
    DATA : out std_logic_vector(M-1 downto 0) -- dato su 8 bit letto
    dalla ROM
);
END component;

component UC_A
PORT (
    CLK_UC_A : in std_logic;
```

```

RST_UC_A: in std_logic;
---segnali di controllo
in_received_UC_A : in STD_LOGIC;
in_ready_UC_A : out STD_LOGIC;
---gestione del contatore
count_in_UC_A : out STD_LOGIC;
count_end_UC_A : in STD_LOGIC; --termine comunicazione stringhe
esaurite (arrivato a 7)
---leggo dalla ROM
read_mem : out STD_LOGIC
);
END component;

component cont_A_B
GENERIC (
    N : integer := 8
);
port(
    CLK : in std_logic;
    RST : in std_logic;
    cont : out std_logic_vector(2 downto 0);
    ---dall'unita' di controllo
    cont_in : in std_logic;
    cont_end : out std_logic
);
END component;

-- // segnali interni

signal address_in: std_logic_vector(2 downto 0):= "000";
signal cont_in: std_logic := '0';
signal cont_end: std_logic := '0';
signal read: std_logic := '0';

begin

-- // MAPPING

    ROM : ROM_A
    generic map (
        M => 8
    )
    port map (
        CLK => clk_A,
        RST => rst_A,
        ADDR => address_in,
        DATA => in_data,
        READ => read
    );

```

```

UC : UC_A
port map (
    CLK_UC_A => clk_A,
    RST_UC_A => rst_A,
    in_received_UC_A => in_received,
    in_ready_UC_A => in_ready,
    count_in_UC_A => cont_in,
    count_end_UC_A => cont_end,
    read_mem => read
);

CONT : cont_A_B
generic map (
    N => 8
)
port map (
    CLK => clk_A,
    RST => rst_A,
    cont_in => cont_in,
    cont_end => cont_end,
    cont => address_in
);

end Structural;

```

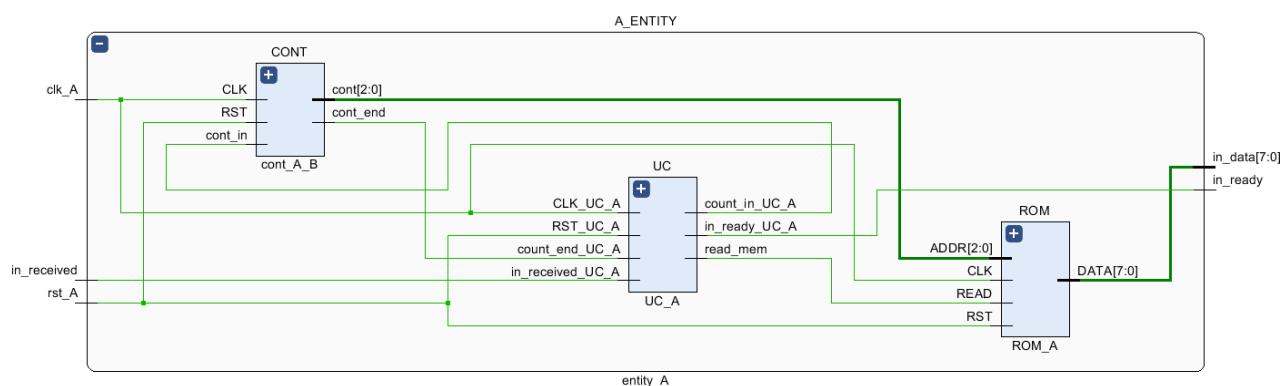


FIG 7.4 Schematici dell'Entity_A generato da VIVADO

Si riporta la ROM di dimensione 8 x 8 bit (64 bit).

ROM_A.vhd (Si riporta solo questa ROM perché per la realizzazione della ROM_B si fa uso della stessa implementazione)

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

```



```

entity ROM_A is
generic(
    N : integer := 8;
    M : integer := 8
);
port(
    CLK : in std_logic;
    RST : in std_logic;
    READ : in std_logic;
    ADDR : in std_logic_vector(2 downto 0);
    DATA : out std_logic_vector(M-1 downto 0)
);
end ROM_A;

-- creo una ROM di 8 elementi da 8 bit ciascuno

architecture behavioral of ROM_A is
type rom_type is array (0 to N-1) of std_logic_vector(M-1 downto 0);
signal ROM : rom_type := (
    "00010011",
    "00010111",
    "00011011",
    "00110011",
    "01010011",
    "11111111",
    "01110011",
    "10010011");

attribute rom_style : bit;

begin

process(CLK)
begin
    if (rising_edge(CLK)) then
        if (RST = '1') then
            DATA <= ROM(conv_integer("000"));
        elsif (READ = '1') then -- uc che gli dice invia
            DATA <= ROM(conv_integer(ADDR));
        end if;
    end if;
end process;

end behavioral;

```

cont_A_B.vhd (Si è usato lo stesso contatore sia per l'entità A e sia per l'entità B)

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

entity cont_A_B is
    generic(
        N: integer := 8
    );
    port(
        CLK : in std_logic;
        RST : in std_logic;
        cont : out std_logic_vector(2 downto 0) := "000";
        --dall'unita' di controllo
        cont_in : in std_logic;
        cont_end : out std_logic := '0'
    );
end cont_A_B;

architecture rtl of cont_A_B is

signal TY: std_logic_vector(2 downto 0);

begin
    -- Contatore modulo 8
    count: process(CLK,RST)
    begin
        if(RST = '1') then
            TY <= "000";
        elsif(rising_edge(CLK)) then
            if (cont_in = '1') then
                if( TY = "111" ) then
                    TY <= "000";
                    cont_end <= '1';
                else
                    TY <= TY + "001";
                end if;
            end if;
        end if;
    end process;
    -- Conteggio in uscita
    cont <= TY;

end rtl;
```



L'unità di controllo dell'entità A è stata progettata in logica cablata; pertanto, si è realizzato un automa che determina la sequenza di eventi che determinano l'evoluzione dell'entità A. Quest'ultimo si rende pronto ($in_ready_UC_A = 1$) a trasmettere quando $in_received_UC_A$ e/o $count_in_UC_A$ sono uguali a 0. Rimango nello stato di *send_req* fino a quando riceve ingresso (proveniente dall'interfaccia) $in_received_UC_A$ uguale ad 1 che abilita il contatore ($count_in_UC_A = 1$) a contare e scandire la prossima stringa. Si entra poi nella fase di *wait_ack* che ritransita in fase di *idle* quando $in_received_UC_A$ ritorna uguale a 0 che setta *read_mem* è uguale 1 ed abilita la lettura dalla memoria (**FIG 7.5**). Si riporta la descrizione in VHDL successivamente.

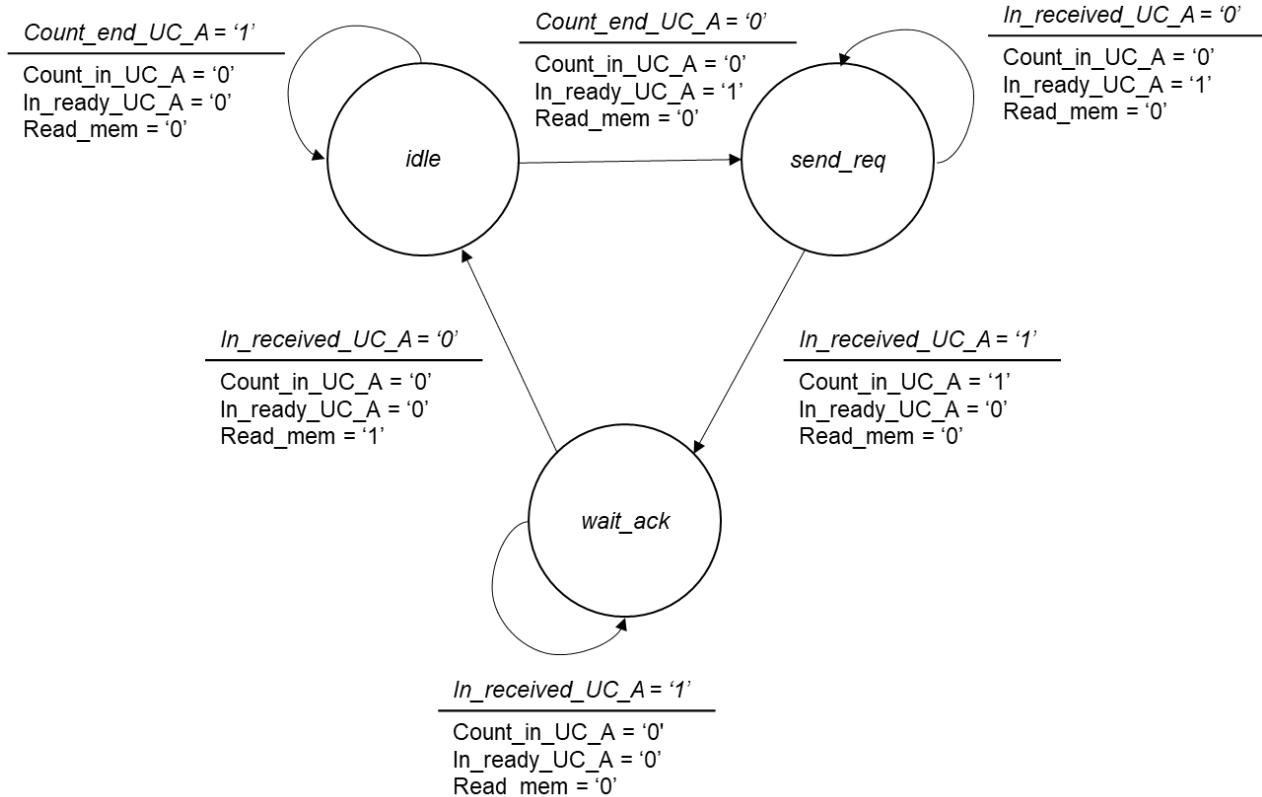


FIG 7.5 Automa UC_A

UC_A.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity UC_A is
    Port (
        CLK_UC_A : in std_logic;
        RST_UC_A: in std_logic;
        -- segnali di controllo
        in_received_UC_A : in STD_LOGIC;
        in_ready_UC_A : out STD_LOGIC := '0';
        -- gestione del contatore
        count_in_UC_A : out STD_LOGIC := '0';

```

```

count_end_UC_A : in STD_LOGIC;
---leggo dalla ROM
read_mem : out STD_LOGIC := '1'
);
end UC_A;

architecture Behavioral of UC_A is

type state is (idle, send_req, wait_ack);
signal current_state : state := idle;

begin

p : process (CLK_UC_A, RST_UC_A)
begin
-- Automa UC_A
if(RST_UC_A = '1') then
    current_state <= idle;

elsif(CLK_UC_A'event AND CLK_UC_A = '1') then
    case current_state is

when idle =>

        if (count_end_UC_A = '0') then
            current_state <= send_req;
            count_in_UC_A <= '0';
            in_ready_UC_A <= '1';
            read_mem <= '0';

        else
            current_state <= idle;
            count_in_UC_A <= '0';
            in_ready_UC_A <= '0';
            read_mem <= '0';

        end if;

when send_req =>

        if(in_received_UC_A = '1') then
            current_state <= wait_ack;
            count_in_UC_A <= '1';
            in_ready_UC_A <= '0';
            read_mem <= '0';

        else
            current_state <= send_req;
            count_in_UC_A <= '0';

```

```
        in_ready_UC_A <= '1';
        read_mem <= '0';

    end if;

    when wait_ack =>

        if (in_received_UC_A = '0') then
            current_state <= idle;
            count_in_UC_A <= '0';
            in_ready_UC_A <= '0';
            read_mem <= '1';

        else
            current_state <= wait_ack;
            count_in_UC_A <= '0';
            in_ready_UC_A <= '0';
            read_mem <= '0';

        end if;

    when others =>
        current_state <= idle;

    end case;
end if;
end process;

end Behavioral;
```

Si passa ora ad analizzare l'entità B:

Entity_B.vhd

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

entity entity_B is
    generic (
        N : integer := 8;
        M : integer := 8
    );
    Port (
        clk_B : in STD_LOGIC;
        rst_B : in STD_LOGIC;
        --dall'interfaccia
        out_received : out STD_LOGIC;
        --verso l'interfaccia
        out_data : in std_logic_vector(M-1 downto 0);
```



```

        out_ready : in std_logic
    );
end entity_B;

architecture Structural of entity_B is

-- // COMPONENTI

component MemoriaB is
    generic (
        N : integer := 8;
        M : integer := 8
    );
    port(
        ADDR: in std_logic_vector(2 downto 0); -- Indirizzo a cui leggere o
        scrivere dalla memoria
        MEM_SOMMA_IN: in std_logic_vector(M-1 downto 0); -- Dato da scrivere
        nella memoria
        MEM_WE: in std_logic; -- Write enable della memoria
        MEM_CLOCK: in std_logic;
        MEM_DATA_OUT: out std_logic_vector(M-1 downto 0) -- Dato in uscita
        dalla memoria
    );
end component;

component UC_B
PORT (
    CLK_UC_B : in std_logic;
    RST_UC_B: in std_logic;
    ---segnali di controllo
    out_received_UC_B : out STD_LOGIC;
    out_ready_UC_B : in STD_LOGIC;
    ---gestione del contatore
    count_in_UC_B : out STD_LOGIC;
    count_end_UC_B : in STD_LOGIC; --termine comunicazione stringhe
    esaurite (arrivato a 7)
    write_enable : out STD_LOGIC;
    enable_ALU : out STD_LOGIC;
    end_somma_in : in std_logic;
    bit_overflow : in STD_LOGIC
);
END component;

component cont_A_B
GENERIC (
    N : integer := 8
);
port(
    CLK: in std_logic;

```

```

        RST: in std_logic;
        cont:    out std_logic_vector(2 downto 0);
        ---dall'unità di controllo
        cont_in : in std_logic;
        cont_end : out std_logic
    );
END component;

component UO_B
generic (
    N : integer := 8;
    M : integer := 8
);
PORT (
    CLK_UO_B : in std_logic;
    RST_UO_B: in std_logic;
    out_data_UO_B: in STD_LOGIC_VECTOR(M-1 downto 0);
    out_data_MEM_B: in STD_LOGIC_VECTOR(M-1 downto 0);
    bit_overflow: out STD_LOGIC;
    enable_ALU : in STD_LOGIC;
    end_somma_out : out std_logic;
    sum_data: out STD_LOGIC_VECTOR(M-1 downto 0) --bit di overflow della
somma
);
END component;

```

-- // SEGNALI INTERNI

```

signal address_in: std_logic_vector(2 downto 0):= (others => '0');
signal cont_in: std_logic := '0';
signal cont_end: std_logic := '0';
signal somma: std_logic_vector(M-1 downto 0);
signal end_somma : std_logic := '0';
signal out_data_MEMb: std_logic_vector(M-1 downto 0);
signal we: std_logic := '0';
signal overflow: std_logic := '0';
signal enable: std_logic := '0';

```

-- // MAPPING

```

begin
    Mem : MemoriaB
    generic map (
        N => 8,
        M => 8
    )
    port map(
        ADDR => address_in,

```



```

    MEM_SOMMA_IN => somma,
    MEM_WE => we,
    MEM_CLOCK => clk_b,
    MEM_DATA_OUT => out_data_MEMb
);

UC : UC_B
port map (
    CLK_UC_B => clk_B,
    RST_UC_B => rst_B,
    out_received_UC_B => out_received,
    out_ready_UC_B => out_ready,
    count_in_UC_B => cont_in,
    count_end_UC_B => cont_end,
    enable_ALU => enable,
    bit_overflow => overflow,
    end_somma_in => end_somma,
    write_enable => we
);

CONT : cont_A_B
generic map (
    N => 8
)
port map (
    CLK => clk_B,
    RST => rst_B,
    cont_in => cont_in,
    cont_end => cont_end,
    cont => address_in
);

UO : UO_B
port map (
    CLK_UO_B => clk_B,
    RST_UO_B => rst_B,
    out_data_UO_B => out_data,
    out_data_MEM_B => out_data_MEMb,
    sum_data => somma,
    bit_overflow => overflow,
    end_somma_out => end_somma,
    enable_ALU => enable
);

end Structural;

```

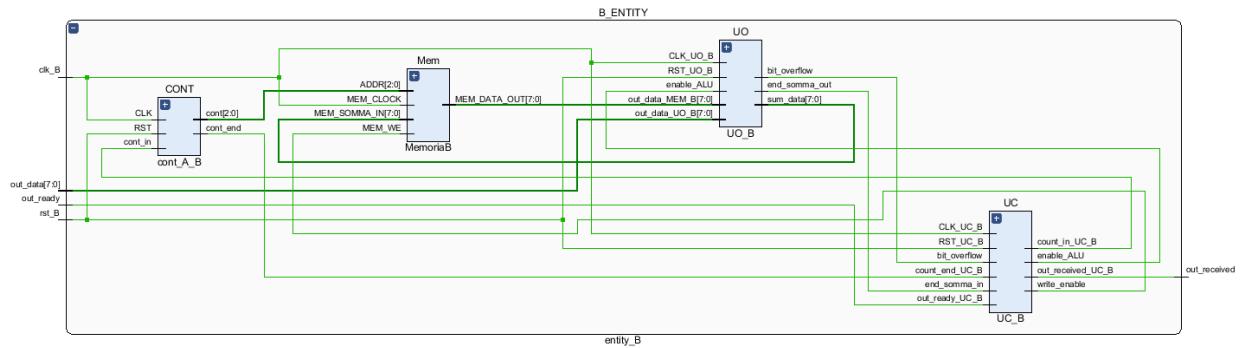


FIG 7.6 Schematico dell'Entity_B

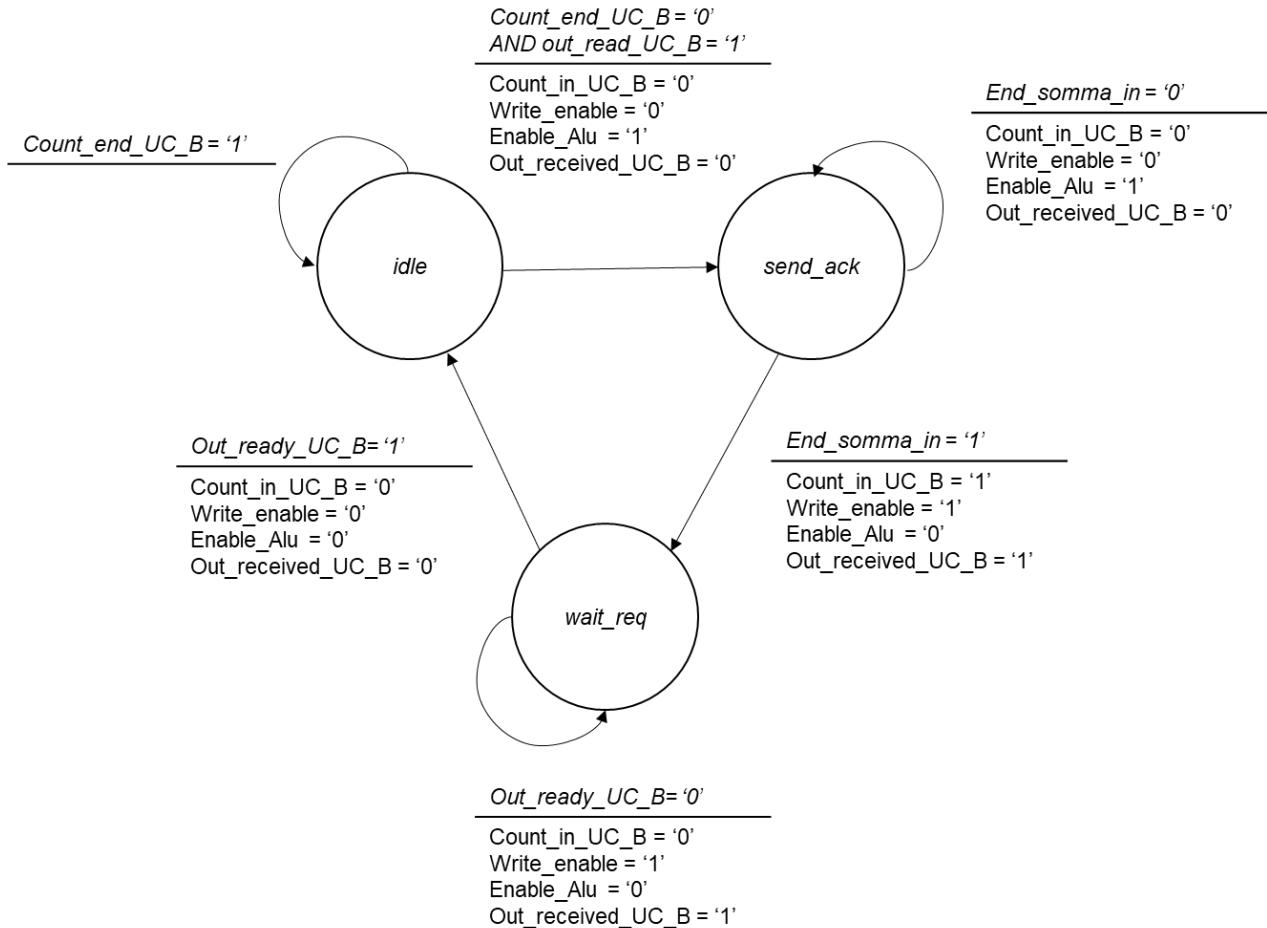
Come già detto in precedenza, la memoria e il Contatore sono gli stessi utilizzati nell'entità A, l'unica cosa che differisce sono i dati inseriti nella memoria stessa; infatti, in questo caso la soluzione prevede che ci siano delle locazioni libere per essere occupate dai valori che calcolerà l'Unità Operativa. (Si riportano i valori della memoriaB):

```

TYPE MEM_ARRAY IS ARRAY (0 TO 2 * N - 1) OF STD_LOGIC_VECTOR (M - 1
DOWNT0 0);
-- valori iniziali nella memoria
SIGNAL MEM : MEM_ARRAY := (
    "01010011",
    "01010111",
    "00111011",
    "00110011",
    "01010111",
    "00000001",
    "00001011",
    "00010011",
    "00000000",
    "00000000",
    "00000000",
    "00000000",
    "00000000",
    "00000000",
    "00000000",
    "00000000");
  
```

Pertanto, si passa allo studio dell'unità di controllo dell'entità B e della relativa unità operativa utile per fare la somma.

Anche l'unità di controllo dell'Entità B è in logica cablata e l'automa descritto determina la conferma di ricezione da parte dell'entità B all'entità A con il segnale *out_ready_UC_B* se esso è 1. Lo stato iniziale è idle che transita nello stato di *send_ack* quando il *count_end_UC_B* = 0 e *out_read_UC_B* è 1 abilitando l'ALU (*Enable_ALU* = 1). Permane in *send_ack* fino a quando il segnale di *end_somma_in* non è alto e transita nello stato di *wait_req*. Si ritorna nello stato di *idle* quando il segnale di *out_ready_UC_B* sia uguale ad 1.

**FIG 7.7 Automa UC_B****UC_B.vhd**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity UC_B is
    Port (
        CLK_UC_B : in STD_logic;
        RST_UC_B: in STD_logic;
        ---gestione del contatore
        out_received_UC_B : out STD_LOGIC := '0';
        out_ready_UC_B : in STD_LOGIC;
        count_in_UC_B : out STD_LOGIC := '0';
        count_end_UC_B : in STD_LOGIC;
        write_enable : out STD_LOGIC := '0';
        enable_ALU : out STD_LOGIC := '0';
        end_somma_in : in std_logic;
        bit_overflow : in STD_LOGIC
    );
end UC_B;

```

architecture Behavioral of UC_B is

```
type state is (idle, send_ack, wait_req);
signal current_state : state := idle;

begin

    p : process (CLK_UC_B, RST_UC_B)
    begin
        --automa UC_B
        if(RST_UC_B = '1') then
            current_state <= idle;

        elsif(CLK_UC_B'event AND CLK_UC_B = '1') then
            case current_state is

                when idle =>
                    if (count_end_UC_B = '0' AND out_ready_UC_B =
                    '1') then
                        current_state <= send_ack;
                        write_enable <= '0';
                        enable_ALU <= '1';
                        out_received_UC_B <= '0';
                        count_in_UC_B <= '0';

                    else
                        current_state <= idle;

                    end if;

                when send_ack =>
                    if(end_somma_in = '1') then
                        current_state <= wait_req;
                        write_enable <= '1';
                        enable_ALU <= '0';
                        out_received_UC_B <= '1';
                        count_in_UC_B <= '1';

                    else
                        current_state <= send_ack;
                        write_enable <= '0';
                        enable_ALU <= '1';
                        out_received_UC_B <= '0';
                        count_in_UC_B <= '0';

                    end if;

            end case;
        end if;
    end process;
end;
```



```

when wait_req =>

    if(out_ready_UC_B = '1') then
        current_state <= idle;
        write_enable <= '0';
        enable_ALU <= '0';
        out_received_UC_B <= '0';
        count_in_UC_B <= '0';

    else
        current_state <= wait_req;
        write_enable <= '1';
        enable_ALU <= '0';
        out_received_UC_B <= '1';
        count_in_UC_B <= '0';

    end if;

when others =>
    current_state <= idle;

end case;

end if;
end process;

end Behavioral;

```

Nell'Unità Operativa di B vengono sommati i segnali di ingresso e quelli presenti nella memoria_B e il risultato sarà trasmesso in uscita. Viene gestito anche il segnale di overflow durante la somma.

UO_B.vhd

```

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

entity UO_B is
generic (
    N : integer := 8;
    M : integer := 8
);
PORT (
    CLK_UO_B : in std_logic;
    RST_UO_B: in std_logic;
    out_data_UO_B: in STD_LOGIC_VECTOR(M-1 downto 0);
    out_data_MEM_B: in STD_LOGIC_VECTOR(M-1 downto 0);

```



```
bit_overflow: out STD_LOGIC := '0';
enable_ALU : in STD_LOGIC;
end_somma_out : out std_logic := '0';
sum_data: out STD_LOGIC_VECTOR(M-1 downto 0)
);
end UO_B;

architecture Behavioral of UO_B is

-- segnali temporanei per la gestione dell'overflow
signal sum_temp : STD_LOGIC_VECTOR(M downto 0);
signal out_data_UO_temp : STD_LOGIC_VECTOR(M downto 0);
signal out_data_MEM_temp : STD_LOGIC_VECTOR(M downto 0);

begin
    -- faccio la concatenazione di uno zero in testa per avere il bit di resto
    out_data_UO_temp <= "0" & out_data_UO_B;
    out_data_MEM_temp <= "0" & out_data_MEM_B;

ALU : process (CLK_UO_B, RST_UO_B)
begin
    if rising_edge(CLK_UO_B) then
        if(RST_UO_B = '1') then
            sum_temp <= "000000000";
        elsif(enable_ALU = '1') then
            sum_temp <= out_data_UO_temp + out_data_MEM_temp;
            end_somma_out <= '1';
            if (sum_temp(M) = '1') then
                sum_data <= "00000000"; -- se c'è overflow allora stampo 0
                bit_overflow <= '1';
            else
                sum_data <= sum_temp(M-1 downto 0); -- se non c'è overflow allora stampo il risultato
                bit_overflow <= '0';
            end if;
        else
            end_somma_out <= '0';
        end if;
    end if;
end process;
end Behavioral;
```

Per l'attuazione dell'**interfaccia**, si riporta il modulo di esso.



Interface_HS.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity interface_HS is
    generic (
        N : integer := 8;
        M : integer := 8
    );
    Port (
        CLK_A : in std_logic;
        RST_A : in std_logic;
        CLK_B : in std_logic;
        RST_B : in std_logic;
        in_received : out STD_LOGIC;
        in_data : in std_logic_vector(M-1 downto 0);
        in_ready : in std_logic;
        out_received : in STD_LOGIC;
        out_data : out std_logic_vector(M-1 downto 0);
        out_ready : out std_logic
    );
end interface_HS;

architecture Structural of interface_HS is

component FSM_A
PORT (
    CLK : in std_logic;
    RST : in std_logic;
    in_data_fsm_A: in std_logic_vector(M-1 downto 0);
    word_buffer: out std_logic_vector(M-1 downto 0);
    in_ready : in std_logic;
    in_received : out std_logic;
    buffer_picked : in std_logic;
    buffer_full : out std_logic
);
END component;

component FSM_B
PORT (
    CLK : in std_logic;
    RST : in std_logic;
    out_data_fsm_B: out std_logic_vector(M-1 downto 0);
    word_buffer: in std_logic_vector(M-1 downto 0);
    out_ready : out std_logic;
    out_received : in std_logic;
    buffer_full : in std_logic;
);
END component;

```



```
    buffer_picked : out std_logic
);
END component;

--segnali interni
signal count: std_logic := '0';
signal buffer_full_signal: std_logic := '0';
signal buffer_picked_signal: std_logic := '0';
signal exchange_word_buffer: std_logic_vector(M-1 downto 0);

begin

  FSMA : FSM_A
  port map (
    CLK => CLK_A,
    RST => RST_A,
    in_data_fsm_A => in_data,
    word_buffer => exchange_word_buffer,
    in_ready => in_ready,
    in_received => in_received,
    buffer_picked => buffer_picked_signal,
    buffer_full => buffer_full_signal
  );

  FSMB : FSM_B
  port map (
    CLK => CLK_B,
    RST => RST_B,
    out_data_fsm_B => out_data,
    word_buffer => exchange_word_buffer,
    out_ready => out_ready,
    out_received => out_received,
    buffer_picked => buffer_picked_signal,
    buffer_full => buffer_full_signal
  );

END structural;
```

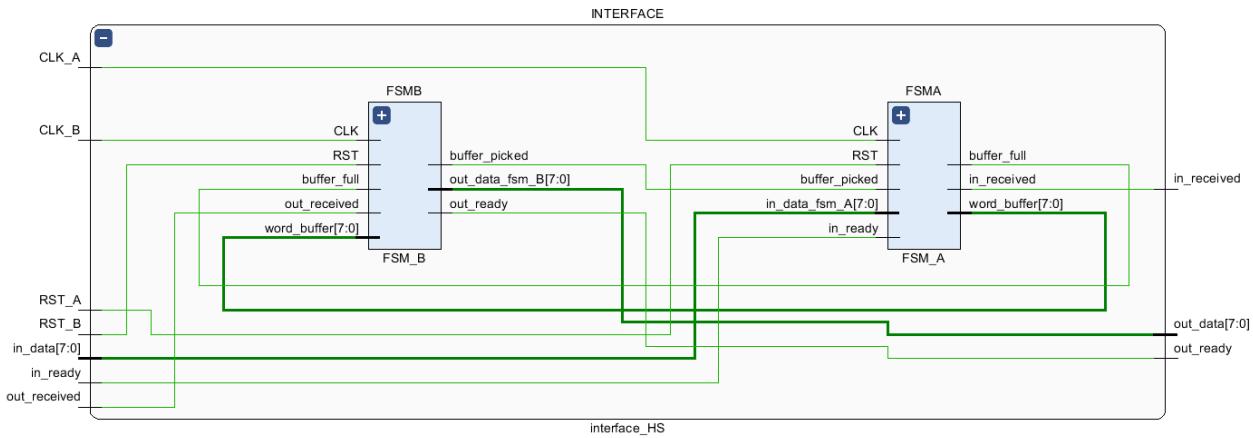


FIG 7.8 Schematico interface

Si passa all'analisi dei due automi che descrivono il protocollo: FSMA e FSMB.

Si nota dalla **FIG 7.9** come lo scambio dei segnali *in_ready* e *in_received* sia stato ripetuto per 8 volte (8 bit) cosicché il buffer si riempie e si alza il segnale di *buffer_full*. A questo punto l'interfaccia attende che *buffer_picked* si alza in modo tale da svuotare il buffer e inviare un segnale di *out_ready*. Si alzerà inoltre il segnale di *out_received*.

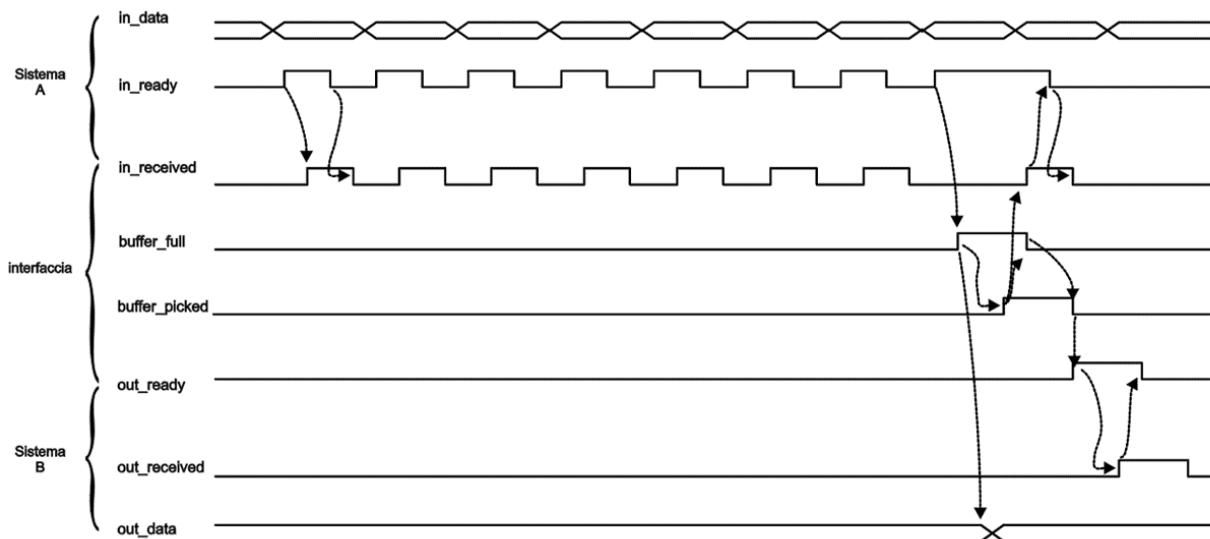
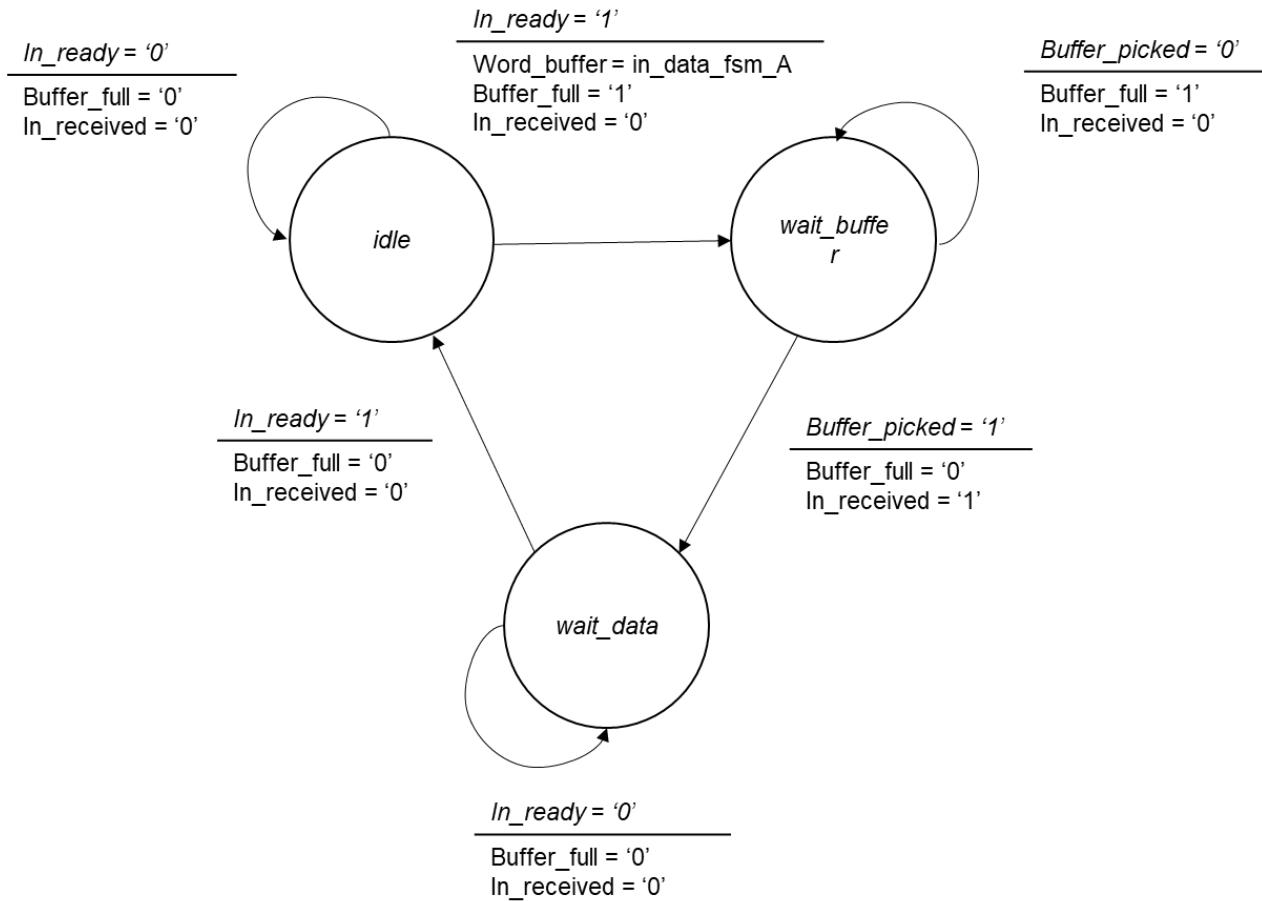


FIG 7.9 Waveform dell'interfaccia

L'automa seguente descrive il comportamento del componente FSM_A in quanto rimane in idle fino a quando *in_ready* non sia uguale ad 1. Quando lo diventa si riempie il *word_buffer* con *in_data_fsm_A* (il dato proveniente dall'entità A) e si pone il buffer come full (*buffer_full* = 1). Attende che l'FSM_B prenda il contenuto del *word_buffer* in modo tale che poi passa in *buffer_picked* = '1' e inviare un segnale di *in_received* all'entità A per indicare che l'entità B ha ricevuto il segnale.

**FIG 7.9** Automa FSM_A**FSM_A.vhd**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FSM_A is
    generic (
        N : integer := 8;
        M : integer := 8
    );
    Port (
        CLK : in std_logic;
        RST : in std_logic;
        in_data_fsm_A: in std_logic_vector(M-1 downto 0);
        word_buffer: out std_logic_vector(M-1 downto 0);
        in_ready : in std_logic;
        in_received : out std_logic := '0';
        buffer_picked : in std_logic;
        buffer_full : out std_logic := '0'
    );

```

```
end FSM_A;

architecture Behavioral of FSM_A is

type state is (idle, wait_buffer, wait_data);
signal current_state : state := idle;

begin

fsm_a: process(CLK, RST)
    variable count : integer range 0 to 1 := 0;
begin
    if (RST = '1') then
        current_state <= idle;

    elsif (CLK'event AND CLK = '1') then
        case current_state is
            when idle =>

                if(in_ready = '1') then
                    current_state <= wait_buffer;
                    word_buffer <= in_data_fsm_A;
                    buffer_full <= '1';
                    in_received <= '0';

                else
                    current_state <= idle;
                    buffer_full <= '0';
                    in_received <= '0';

                end if;

            when wait_buffer =>

                if(buffer_picked = '1') then
                    current_state <= wait_data;
                    buffer_full <= '0';
                    in_received <= '1';

                else
                    current_state <= wait_buffer;
                    buffer_full <= '1';
                    in_received <= '0';

                end if;

            when wait_data =>

                if(in_ready = '1') then
```

```

        current_state <= idle;
        buffer_full <= '0';
        in_received <= '0';

    else
        current_state <= wait_data;
        buffer_full <= '0';
        in_received <= '0';

    end if;

when others =>
    current_state <= idle;
end case;
end if;
end process;

end Behavioral;

```

L'automa FSM_B invece deve attendere che il *buffer_full* sia uguale ad 1 (buffer pieno) e dunque si deve settare *buffer_picked* = 1 in modo tale da segnalarlo all' FSM_A. Si passa nello stato di *wait_data* nel quale attende che l'*out_received* sia uguale ad 1 e passare nello stato di *wait_buffer* e tornare nello stato di idle se il buffer ritorna di nuovo pieno (*buffer_full* = 1).

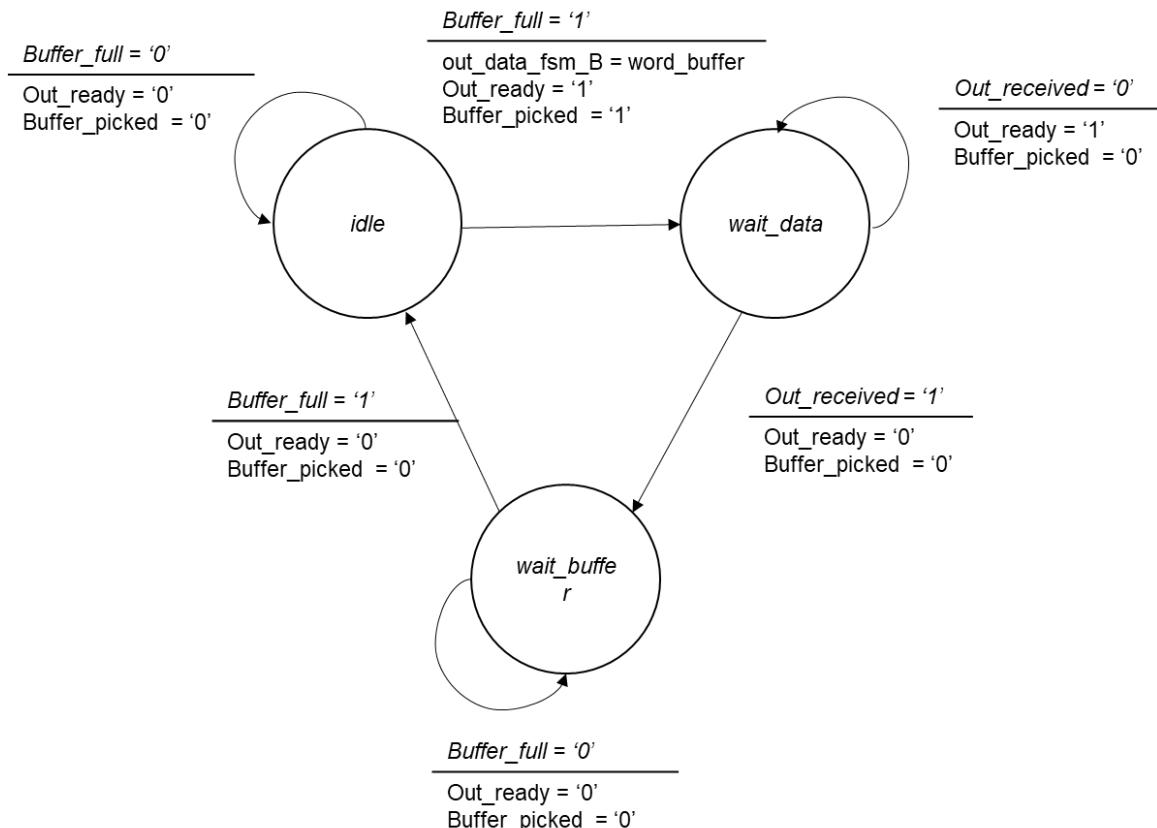


FIG 7.10 Automa FSM_B

FSM_B.vhd

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

entity FSM_B is
    generic (
        N : integer := 8;
        M : integer := 8
    );
    Port (
        CLK : in std_logic;
        RST : in std_logic;
        out_data_fsm_B: out std_logic_vector(M-1 downto 0);
        word_buffer: in std_logic_vector(M-1 downto 0);
        out_ready : out std_logic := '0';
        out_received : in std_logic;
        buffer_full : in std_logic;
        buffer_picked : out std_logic := '0'
    );
end FSM_B;
```

```
architecture Behavioral of FSM_B is
```

```
type state is (idle, wait_data, wait_buffer);
signal current_state : state := idle;
```

```
begin
```

```
fsm_b: process(CLK,RST)
begin
    if (RST = '1') then
        current_state <= idle;

    elsif (CLK'event AND CLK = '1') then
        case current_state is
            when idle =>

                if(buffer_full = '1') then
                    current_state <= wait_data;
                    out_data_fsm_B <= word_buffer;
                    out_ready <= '1';
                    buffer_picked <= '1';

                else
                    current_state <= idle;
                    buffer_picked <= '0';
                    out_ready <= '0';

                end if;

            when wait_data =>

                if(out_ready = '1') then
                    current_state <= idle;
                    buffer_picked <= '1';
                    out_data_fsm_B <= word_buffer;
                    word_buffer <= word_buffer;

                else
                    current_state <= wait_data;
                    buffer_picked <= '0';
                    out_ready <= '0';

                end if;

            when wait_buffer =>

                if(out_ready = '1') then
                    current_state <= idle;
                    buffer_picked <= '1';
                    out_data_fsm_B <= word_buffer;
                    word_buffer <= word_buffer;

                else
                    current_state <= wait_buffer;
                    buffer_picked <= '0';
                    out_ready <= '0';

                end if;

        end case;
    end if;
end process fsm_b;
```



```
        end if;

        when wait_data =>

            if(out_received = '1') then
                current_state <= wait_buffer;
                buffer_picked <= '0';
                out_ready <= '0';

            else
                current_state <= wait_data;
                buffer_picked <= '0';
                out_ready <= '1';

            end if;

        when wait_buffer =>

            if(buffer_full = '1') then
                current_state <= idle;
                buffer_picked <= '0';
                out_ready <= '0';

            else
                current_state <= wait_buffer;
                buffer_picked <= '0';
                out_ready <= '0';

            end if;

        when others =>
            current_state <= idle;

        end case;
    end if;
end process;

end Behavioral;
```

7.1.2 La simulazione della Rete

Come da progettazione, i clock dell'entità sono differenti. Nonostante questa differenza della base di clock le entità sono state in grado di scambiarsi le informazioni, o meglio, l'entità A è stato in grado di trasmettere ogni stringa di bit. Si può notare infatti che la memoria di B si sia riempita (zoom del contenuto della memoria).



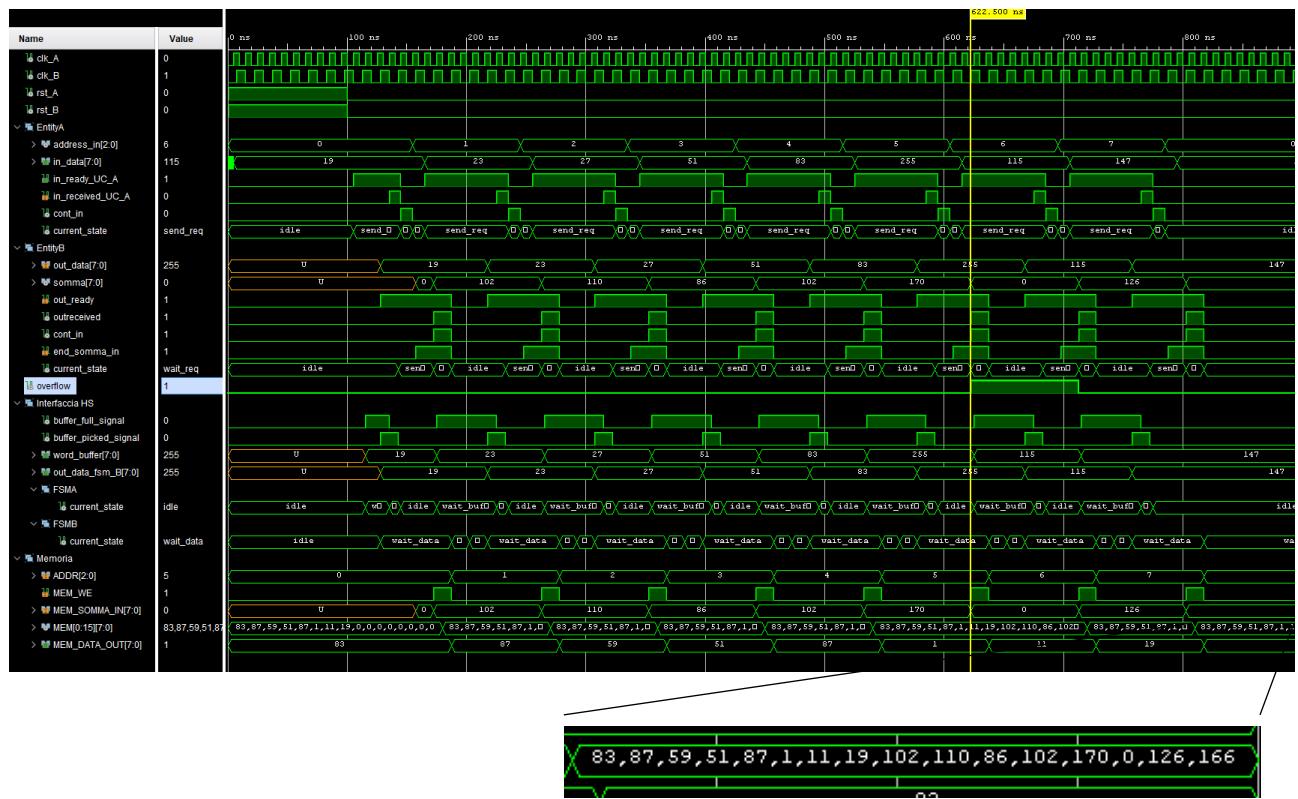


FIG 7.11 TestBench Handshaking

tb_handshaking_A_B.vhd

```

library ieee;
use ieee.std_logic_1164.all;

entity tb_handshaking_A_B is
end tb_handshaking_A_B;

architecture tb of tb_handshaking_A_B is

component handshaking_A_B
    port (clk_A : in std_logic;
          clk_B : in std_logic;
          rst_A : in std_logic;
          rst_B : in std_logic);
end component;

signal clk_A : std_logic;
signal clk_B : std_logic;
signal rst_A : std_logic;
signal rst_B : std_logic;

constant TbPeriodA : time := 10 ns; -- EDIT Put right period here
signal TbClockA : std_logic := '0';
signal TbSimEndedA : std_logic := '0';

```



```
constant TbPeriodB : time := 15 ns; -- EDIT Put right period here
signal TbClockB : std_logic := '0';
signal TbSimEndedB : std_logic := '0';

begin

    dut : handshaking_A_B
    port map (clk_A => clk_A,
              clk_B => clk_B,
              rst_A => rst_A,
              rst_B => rst_B);

    -- Clock generation
    TbClockA <= not TbClockA after TbPeriodA/2 when TbSimEndedA /= '1'
    else '0';
    TbClockB <= not TbClockB after TbPeriodB/2 when TbSimEndedB /= '1'
    else '0';

    -- EDIT: Check that clk_A is really your main clock signal
    clk_A <= TbClockA;
    clk_B <= TbClockB;

    stimuli : process
    begin
        -- EDIT Adapt initialization as needed
--        clk_B <= '0';
--        rst_B <= '0';

        -- Reset generation
        -- EDIT: Check that rst_A is really your reset signal
        rst_A <= '1';
        rst_B <= '1';
        wait for 100 ns;
        rst_A <= '0';
        rst_B <= '0';
        wait for 100 ns;

        -- EDIT Add stimuli here
        wait for 100 * TbPeriodA;
        wait for 100 * TbPeriodB;

        -- Stop the clock and hence terminate the simulation
        TbSimEndedA <= '1';
        TbSimEndedB <= '1';
        wait;
    end process;

end tb;
```



7.1.3 Considerazioni finali e possibile scopo futuro

Il protocollo di Handshaking progettato prevede anche la possibilità di non avere lo stesso parallelismo nel trasferimento dei dati. Ad esempio, A trasmette 4 bit, B può ricevere 16 bit. L'interfaccia funge da arbitro per evitare possibili errori di trasferimento. Per gestire, dunque, un parallelismo differente si deve aggiungere all'interno dell'interfaccia un contatore e nel relativo automa di FSM_A un ulteriore stato in cui si valuta il valore del contatore in maniera tale da riempire il word_buffer con le stringhe inviate da A, appena che il word_buffer si sarà riempito si passa nello stato wait_buffer che segnala a B il *buffer_full* = 1, così come avviene adesso.

Il progetto può essere riutilizzato nei progetti che richiedono il protocollo come la comunicazione seriale.

8 Processore MIC-1

Si analizza il Processore MIC-1 seguendo le specifiche richieste dal cliente:

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

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

8.1 Introduzione

Per l'esercizio proposto è stato deciso di approfondire le istruzioni BIPUSH ed ISUB, analizzando attentamente l'evoluzione delle rispettive micro-procedure. Si è poi scelto di modificare l'istruzione IOR, in modo da farle eseguire l'operazione di AND.

8.1.1 Architettura Processore Mic-1

Il processore Mic-1 implementa un sottoinsieme delle istruzioni della Java Virtual Machine. È un processore a registri generali per dati ed indirizzi, ma presenta una serie di registri interni ognuno con un compito ben preciso. Inoltre questo processore presenta una architettura a stack, ovvero le istruzioni aritmetiche non hanno operandi esplicativi, bensì impliciti e si presuppone che gli operandi su cui si vuole eseguire l'istruzione siano presenti nelle prime due locazioni dello stack. Per eseguire le operazioni è quindi necessario controllare gli accessi in memoria e la ALU.

L'implementazione processore Mic-1 utilizzata nell'esercizio 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 meglio il funzionamento del processore Mic-1 (sia della parte operativa che di quella di controllo), si riporta lo schema dello stesso in Figura 8.1.

Il datapath contiene registri a 32 bit accessibili solo a livello microarchitetturale e 2 bus, B e C, per lettura e scrittura. Alcuni registri sono cablati in modo da poter essere usati solo per uno scopo specifico:

- i registri dell'interfaccia con la memoria:
 - MAR - memory address register;
 - MDR - memory data register;
 - PC - program counter;
 - MBR - memory byte register;

MAR e MDR costituiscono l'interfaccia dato, cioè vengono utilizzati quando bisogna prelevare un dato dalla memoria; invece, PC e MBR costituiscono l'interfaccia istruzione, cioè sono utilizzati quando bisogna prelevare un'istruzione dalla memoria.

- il registro che mantiene il primo operando dell'ALU:
 - H - holding.



- Gli altri registri sono funzionalmente equivalenti, ed i loro nomi sono assegnati sulla base dell'uso che se ne fa nel microprogramma:
 - SP - stack pointer;
 - LV - local variables;
 - CPP - constant pool pointer;
 - TOS - top of stack;
 - OPC - scratch register.

C'è poi la ALU che esegue le operazioni su due operandi: uno contenuto appunto nel registro H e l'altro proveniente dal bus B. L'operazione effettuata dipende dal valore dei bit di controllo in ingresso.

La micro-ROM è implementata dal control store. Ogni istruzione contiene i seguenti campi:

- Addr: indirizzo di una potenziale prossima microistruzione, che verrà messo all'interno del registro MPC;
- JAM: tramite i bit di JAM possiamo andare a modificare l'indirizzo della prossima microistruzione contenuto all'interno di MPC, andando a modificare però solo il bit più significativo;
- ALU: bit di controllo dell'ALU e dello shifter;
- C: controlla quali registri vengono scritti dal bus C;
- Mem: controlla le operazioni di memoria;
- B: seleziona il registro connesso al bus B; questi bit dovrebbero essere 9 in quanto sono 9 i registri che possono scrivere sul bus B, ma siccome solo uno alla volta può scrivere; quindi, il bus B può essere usato solo in mutua esclusione, viene utilizzata una codifica su 4 bit; questo approccio però necessita poi dell'utilizzo di un decoder 4-16.

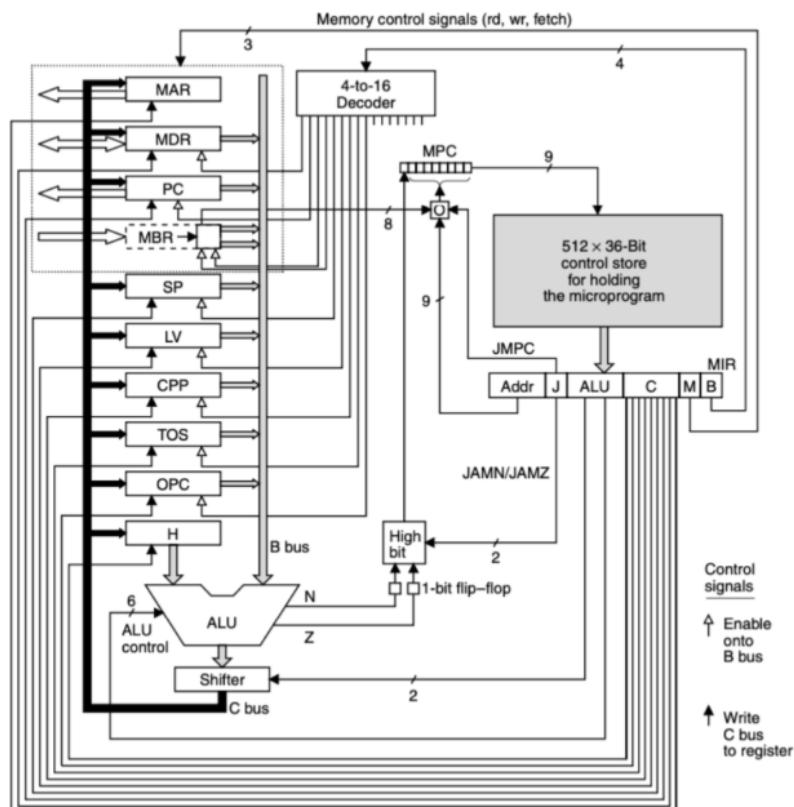


FIG 8.1: Architettura Processore Mic-1

8.2 Soluzione

8.2.1 Esercizio A

Come richiesto dalla traccia, si è deciso di effettuare l'analisi di due istruzioni: BIPUSH e ISUB.

BIPUSH. Per analizzare il funzionamento dell'istruzione BIPUSH si è sviluppato il programma in linguaggio IJVM riportato di seguito:

```
.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 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.

```
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.

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.

**FIG 8.2:** 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, come si può vedere da mem_inst_in.
 - 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; infatti, ora MBR contiene il codice 0x10.
 - b) Viene aggiornato il valore del micro-Program Counter (MPC), in modo da eseguire la microprocedura 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 JMPC pari ad 1 all'interno del registro MIR, registro contenente la microistruzione correntemente eseguita. Come deve essere,

- 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 (reg_to_b_control, c_to_reg_control e alu_control), 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 (pc_reg).
 - c) Viene aggiornato il valore del PC (mem_instr_in).
 - d) Viene incrementato di uno il valore del MPC, in modo da eseguire la microistruzione successiva (mpc_virtual_reg).
 - 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 (mbr_reg).
 - b) Viene aggiornato il valore dei registri MAR e SP (mar_reg, sp_reg). 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 (mpc_virtual_reg).
 - 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 (pc_reg).
 - 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.

ISUB. Per analizzare il funzionamento dell'istruzione ISUB si è sviluppato il programma in linguaggio IJVM riportato di seguito:

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

Si osserva inoltre che l'istruzione ISUB, 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 ISUB.

```
isub = 0x5C:
    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 per realizzare l'istruzione ISUB.

Si noti che l'analisi dell'istruzione 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.



FIG 8.3 Simulazione esecuzione istruzione ISUB

Dalla simulazione si evincono le seguenti operazioni principali, necessarie alla corretta esecuzione dell'istruzione:

1. (285 ns) BIPUSH
 - a) 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.
 - b) Vengono inoltre eseguite le ultime 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 0x5C, codice operativo dell'istruzione ISUB (mem_instr_in).
 - c) Vengono inoltre eseguite le ultime operazioni riguardanti l'istruzione BIPUSH.
3. (305 ns) MAIN:
 - a) Viene prelevata il codice operativo dell'istruzione ISUB e caricato nel registro MBR (mbr_reg).
 - b) Viene aggiornato il valore del MPC (col valore 05c), in modo da eseguire la microprocedura associata all'istruzione ISUB: 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 JMPC pari ad 1 all'interno del registro MIR.

- Come deve essere, 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 (reg_to_b_control, c_to_reg_control e alu_control), in modo da incrementare il PC.
4. (315 ns) Inizio ISUB:
- 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 0x36 (mem_instr_in).
 - c) Viene aggiornato il valore del PC (pc_reg), da 8 a 9.
 - d) Viene incrementato di uno il valore del MPC da 05c a 05d (mpc_virtual_reg), 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 0x36 e caricato nel registro MBR (mbr_reg).
 - b) Viene alzato il segnale rd_ff di richiesta di lettura dalla memoria.
 - c) Viene aggiornato il valore dei registri MAR (mar_reg) e SP (sp_reg). In questo modo viene specificato di leggere dalla memoria il byte contenuto alla locazione puntata dal nuovo SP (sp_reg), rappresentante l'operando B 0xa. Tuttavia, si osserva che il valore del registro TOS (tos_reg) non viene variato, per cui conterrà ancora il valore dell'operando A 0xe.
 - d) Viene incrementato di uno il valore del MPC, in modo da eseguire la microistruzione successiva, da 05d a 05e.
 - 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 (mdr_reg) con il valore dell'operando B (0xa), 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 (h_reg).
 - 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 differenza 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, al valore ffffffc. Solo in questo momento il risultato è pronto per essere scritto in memoria.
8. (355 ns) L'operazione di scrittura del risultato 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 ISUB. In particolare, si nota che lo stack pointer viene aggiornato all'istante 325 ns, mentre il registro TOS all'istante 345 ns.

8.2.2 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 IOR in modo da far eseguire l'istruzione IAND. Per poter effettuare le modifiche necessarie si è sviluppato il programma in linguaggio IJVM riportato di seguito, il quale esegue l'istruzione IOR dopo aver caricato due operandi nello stack.

```
.main
BIPUSH 0xA
BIPUSH 0xE
IOR
.endmethod
```

Di seguito è riportata la specifica in linguaggio MAL dell'istruzione IJVM IOR.

```
ior = 0xB6:
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR OR H; wr; goto main
```

A questo punto è possibile proseguire con l'analisi delle microistruzioni eseguite dal processore per poter realizzare IOR.

Si noti che l'analisi dell'istruzione 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.

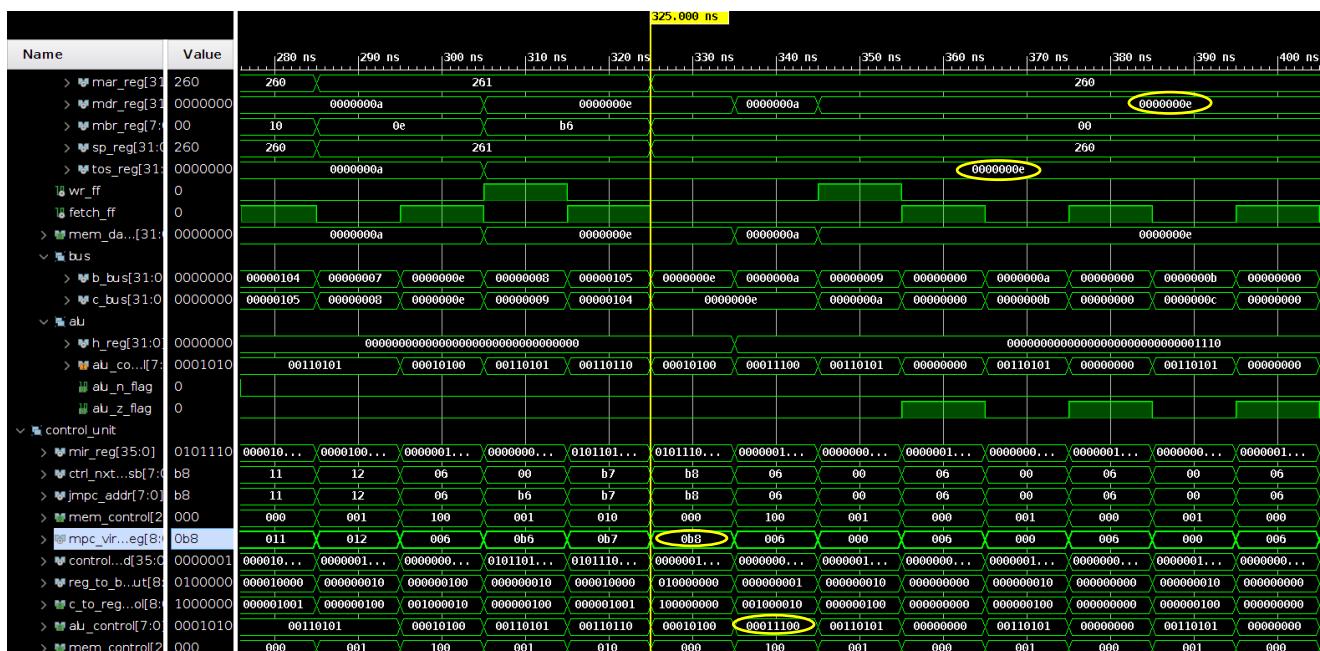


FIG 8.4: Simulazione esecuzione IOR

Dalla simulazione si osserva che le operazioni eseguite dal processore per realizzare l'istruzione IOR sono del tutto simili a quelle effettuate per realizzare l'istruzione ISUB. 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 differenza, in questo caso sono configurati per effettuare l'operazione di OR bit a bit.

L'operazione che viene effettuata è $0xa \text{ OR } 0xe = 0xe$.

Dalla simulazione si evince inoltre che il punto chiave dell'istruzione IOR risiede nell'impostazione dei bit del registro ALU_CONTROL nell'istante 335 ns (00011100). Tale configurazione di bit corrisponde infatti all'esecuzione da parte dell'ALU dell'operazione di OR bit a bit fra gli operandi in ingresso. Il risultato (0xe) dell'elaborazione è visibile nei registri MDR e TOS all'istante 345 ns.

Nella tabella sono riportati i valori del registro ALU_CONTROL per le istruzioni IOR e IAND.

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

Per modificare il comportamento della IOR 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 OR.

L'indirizzo di interesse è 0Xb8, corrispondente al valore 184 in decimale. Il valore contenuto nella micro-ROM a tale indirizzo è:

000000110000 '00011100' 0010000101000000

dove i bit delimitati dagli apici rappresentano i bit di controllo dell'ALU, contenuti in ALU_CONTROL. Seguendo quanto riportato in Tabella, per modificare l'istruzione di IOR in modo da farle eseguire la AND bit a bit è stato quindi sufficiente sostituire al valore precedente la stringa di bit:

000000110000 '00001100' 0010000101000000

Apportate tali modifiche al contenuto della micro-ROM, si è osservato il comportamento del processore nel caso di esecuzione dell'istruzione IOR. La simulazione dell'esecuzione è riportata in **Figura 8.5**

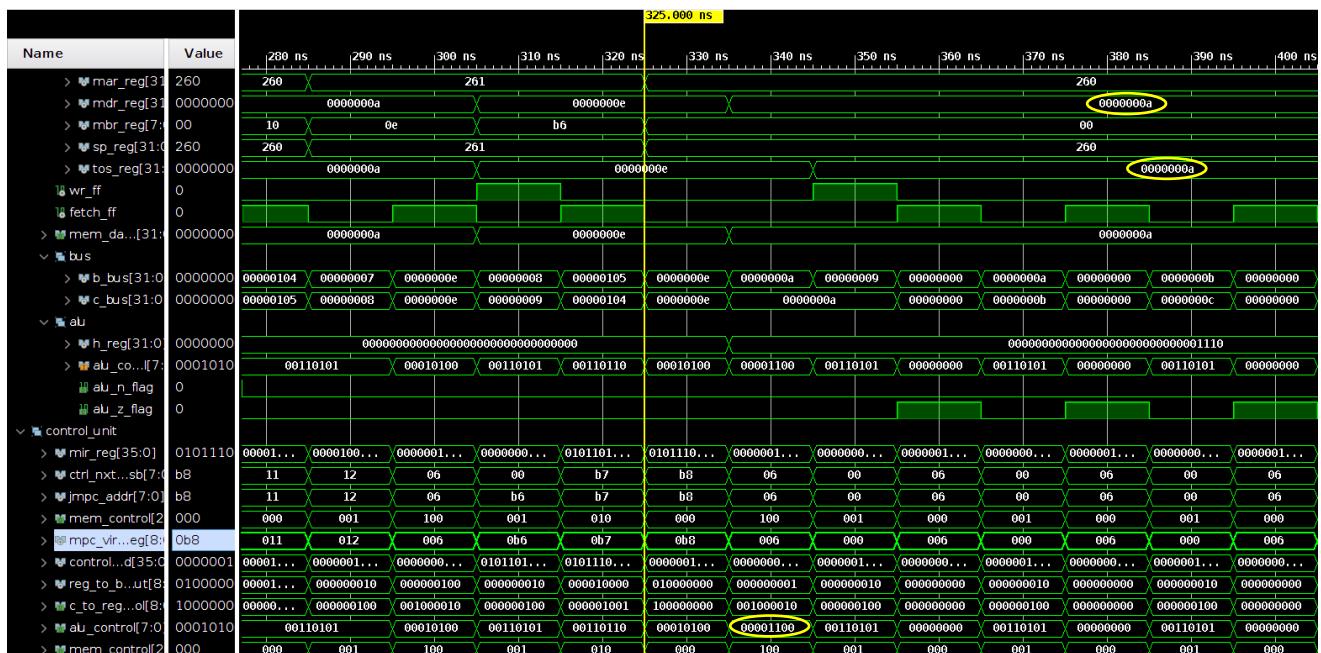


FIG 8.5: Simulazione esecuzione IOR modificata in IAND

Si può osservare come il valore del registro ALU_CONTROL sia stato modificato al valore 00001100 all'istante 335ns e quindi come il risultato dell'istruzione di IOR corrisponda effettivamente al risultato dell'IAND. Il risultato (0xa) dell'elaborazione è visibile nei registri MDR e TOS all'istante 345 ns.

8.2.3 Modo alternativo Esercizio B

Il procedimento appena proposto per la modifica di una istruzione richiede però la conoscenza della tabella della ALU per poter andare poi a modificare i bit corrispondenti ad alu_control nell'istruzione che si vuole modificare nel control store.

Un modo alternativo e più semplice per poter modificare un'istruzione consiste nell'andare a modificarla nel file MAL e lanciare successivamente i target make relativi.

Prendendo sempre l'esempio precedentemente proposto, vogliamo andare a modificare nel file MAL l'istruzione IOR in modo che esegua la IAND, nel seguente modo:

```
ior = 0xB6:  
    MAR = SP = SP - 1; rd  
    H = TOS  
    MDR = TOS = MDR AND H; wr; goto main
```

Dopo aver fatto questo, bisogna andare a lanciare i comandi make per andare a modificare il control store e la ram. Per creare il control store bisogna andare da linea di comando nella cartella src/main/mal/ajvm.mal e lanciare il comando

```
cmake -build build -target create_control_store
```

Per creare la ram, bisogna andare nella cartella src/main/ajvm/program.ajvm e lanciare il comando

```
cmake -build build -target create_ram
```

Dopo aver lanciato questi comandi, andando a controllare nel control store a riga 184 (quella corrispondente alla IOR), possiamo vedere che è stata effettuata la stessa modifica fatta manualmente nel procedimento precedente:

000000110000 '00011100' 0010000101000000 \Rightarrow 000000110000 '0001100' 0010000101000000



9 Interfaccia seriale

Si realizza un'Interfaccia Seriale seguendo le specifiche richieste dal cliente:

- **9.1:** Sfruttando l'implementazione fornita dalla Digilent di un dispositivo UART (componente RS232RefComp.vhd), progettare e implementare in VHDL un sistema costituito da 2 nodi A e B collegati tra loro mediante una interfaccia seriale. Il sistema A acquisisce una stringa di 8 bit dall'utente (mediante gli switch della board di sviluppo) e la invia mediante la seriale al sistema B, che la manda in output sui led della board di sviluppo.
- **9.2:** Implementare uno dei seguenti sistemi a scelta dello studente:
 - 2_UART_MEM:** come variante dell'esercizio 9.1, il sistema A invia al sistema B tramite l'interfaccia seriale N stringhe di 8 bit contenute all'interno di una memoria ROM. Le stringhe ricevute vengono memorizzate in una memoria locale a B. Il progetto deve prevedere che A utilizzi un componente contatore per scandire le N stringhe da inviare.
 - UART_PC:** il sistema realizza la comunicazione fra un nodo A rappresentato da un componente sintetizzato su un FPGA e un nodo B rappresentato da un terminale seriale in esecuzione su PC (es. Termite), previa connessione di PC e board tramite dispositivo fisico RS232 (uno degli endpoint di comunicazione è rappresentato dal PC). Il componente A acquisisce una stringa di 8 bit che rappresenta un carattere in codifica ASCII fornita dall'utente mediante gli switch della board di sviluppo, e la invia mediante il dispositivo UART al terminale B in esecuzione sul PC, in cui il carattere viene visualizzato. Allo stesso modo, il componente deve essere in grado di ricevere attraverso lo stesso dispositivo UART (oppure una seconda UART) un carattere trasmesso dal terminale e mostrarlo sui led.

9.1 Progettazione interfaccia seriale

Il progetto è composto da un nodo trasmettitore, un nodo ricevitore ed un'unità di controllo.

Il **trasmettitore** può essere descritto dall'automa nella figura 9.1.

Il trasmettitore parte da uno stato IDLE, con il contatore posto a 0 e il buffer vuoto. Quando il segnale di scrittura (WR) si alza passiamo nello stato TRANSFER. Lo stato TRANSFER abilita i load del registro a scorrimento, fino a quando i dati paralleli non vengono scritti nel registro. Dopodiché si passa nello stato di SHIFT, dovremo ovviamente shiftare il bit meno significativo sulla TXD e incrementare il valore del contatore di trasferimento.

Dopo aver inviati tutti i bit si passa allo stato di CHECKSTOP che termina la trasmissione, verifica eventuali errori di trasmissione e ritorna nello stato IDLE.

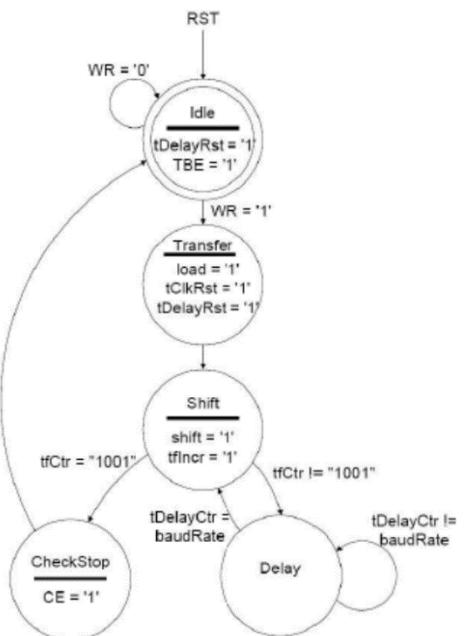


FIG. 9.1 Automa Trasmettitore

Il **ricevitore** può essere descritto dall' automa nella figura 9.2. Il ricevitore parte anch'esso dallo stato IDLE e rimane in questo stato fino a quando RXD è '1'. Quando RXD diventa uguale a '0' allora si passa allo stato EIGHT-DELAY, tale stato conterà fino a 8 e quando finirà il conteggio ci sposteremo negli stati WAITFOR0 e WAITFOR1.

In questi ultimi due stati andiamo a valutare il bit più significativo del contatore, rimaniamo su WAITFOR1 per i conteggi che hanno valore compreso tra 8 e 15, così di fatto aspetteremo 16 conteggi in modo tale da piazzarci al centro del bit successivo. Successivamente questa macchina a stati finiti ci porterà in uno stato GETDATA, che ha come compito di attivare lo shift register e incrementare il datacounter.

Un contatore verrà usato come divisore di frequenza e un altro come contatore di bit ricevuti; quindi, dovremmo avere un contatore modulo 10, dato che abbiamo 8 bit per il dato, 1 bit di parità ed 1 bit di stop.

Appena il dataCtr sarà uguale a "1001" ci spostiamo nello stato di CHECKSTOP, il cui compito è quello di verificare la presenza di eventuali errori di trasmissione, infine, torniamo nello stato di IDLE.

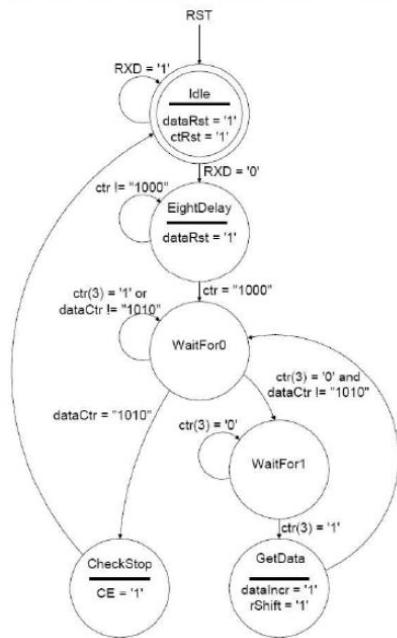


FIG. 9.2 Automa Ricevitore

Per il progetto dell'Interfaccia Seriale abbiamo, dunque, realizzato un'unità di controllo che permette il reset del trasmettitore e del ricevitore quando il segnale di reset è alto. Mentre, quando il segnale di Load è alto abilitiamo la scrittura e passiamo il dato desiderato, se invece è alto il segnale di TBE poniamo il segnale di lettura pari a '0'. Infine, quando il segnale di RDA è alto allora accendiamo i led e poniamo RD a '1'.

9.1.1 Schematico in RTL Analysis di Vivado

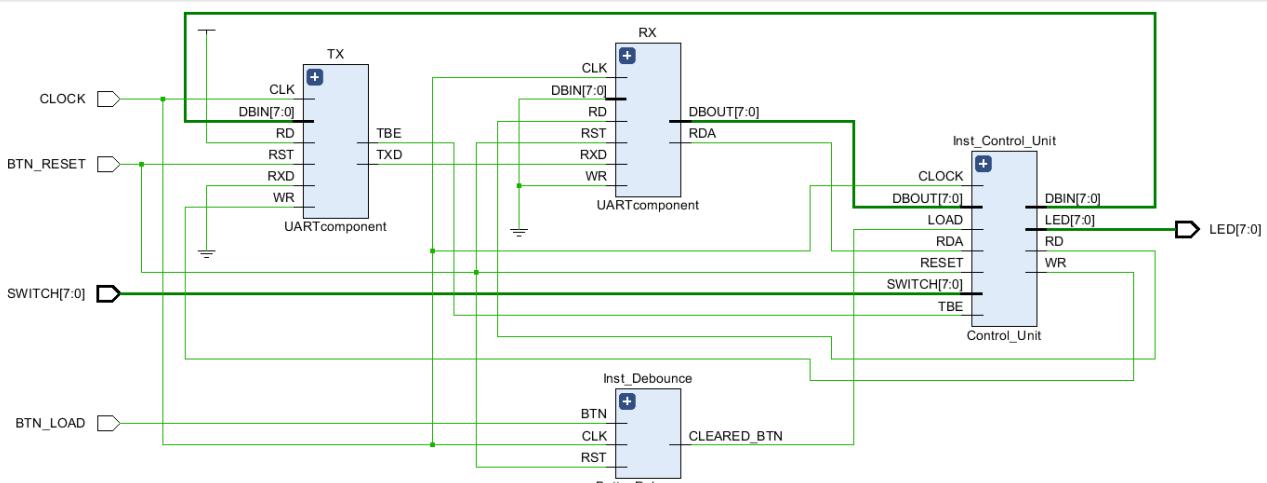


FIG. 9.3

9.1.2 Codice VHDL***Control_Unit.vhd***

```
begin

PROC : process (CLOCK)
    begin
        if(rising_edge(CLOCK)) then
            if(RESET = '1') then
                LED <= (others =>'0');
                WR <= '0';
                DBIN <= (others =>'0');
                RD <= '1';
            else
                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 if;
    end process PROC;

end Behavioral;
```

9.1.3 Test Bench***UART_On_Board_TB.vhd***

```
stim_proc : PROCESS
BEGIN
    -- hold reset state for 100 ns.
    WAIT FOR 10 ms;

    WAIT FOR CLOCK_period * 10;
    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;

```

9.1.4 Simulazione

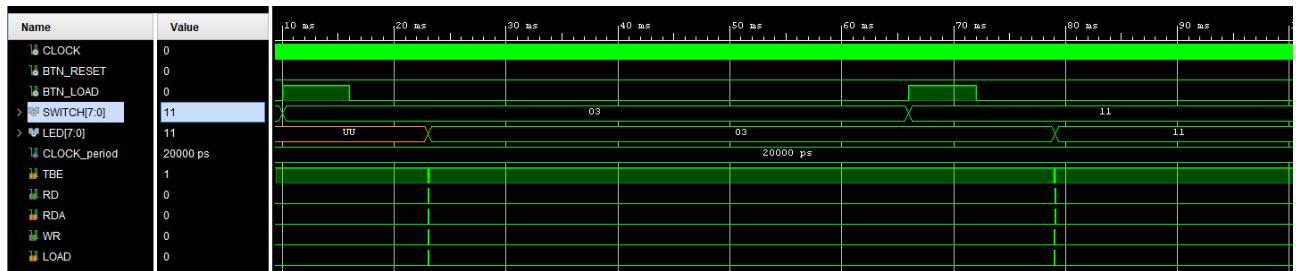


FIG. 9.4

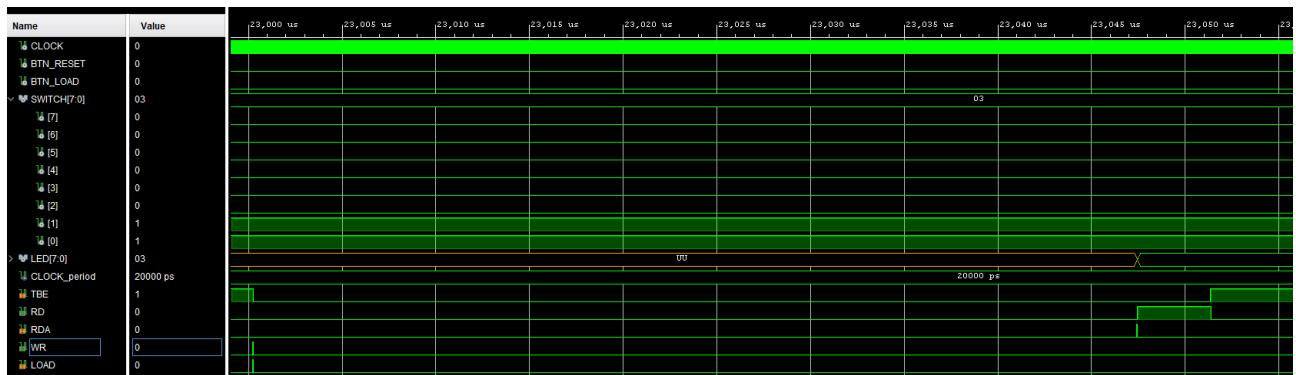


FIG 9.5 (Variazione TBE, RD, RDA, LOAD)

Istante 10ms:

Ricevitore e trasmettitore si trovano in IDLE alziamo il BTN_LOAD e impostiamo il valore dello SWITCH a "00000011".

Intorno all'istante 23ms:

- 1) Con TBE alzato, si alza il segnale di load e carichiamo il dato in DBIN ("00000011"), al colpo di clock successivo il load si abbassa e si attiva il segnale di write (WR) per un colpo di clock, dopodiché si abbassa il TBE e il WR.
- 2) Dopo 40 us si alza il segnale RDA per un colpo di clock, al colpo di clock successivo il dato viene messo in uscita sui LED ("00000011") ed il segnale RD si alza, infine, si ritorna nello stato di IDLE.

Istante 62ms:

Ricevitore e trasmettitore si trovano nuovamente in IDLE alziamo il BTN_LOAD e impostiamo il valore dello SWITCH a "00001011".

Intorno all'istante 79ms:

- 1) Con TBE alzato, si alza il segnale di load e carichiamo il dato in DBIN ("00001011"), al colpo di clock successivo il load si abbassa e si attiva il segnale di write (WR) per un colpo di clock, dopodiché si abbassa il TBE e il WR.
- 2) Dopo 40 us si alza il segnale RDA per un colpo di clock, al colpo di clock successivo il dato viene messo in uscita sui LED ("00001011") ed il segnale RD si alza, infine, si ritorna nello stato di IDLE.

9.1.5 Constraints

##Switches

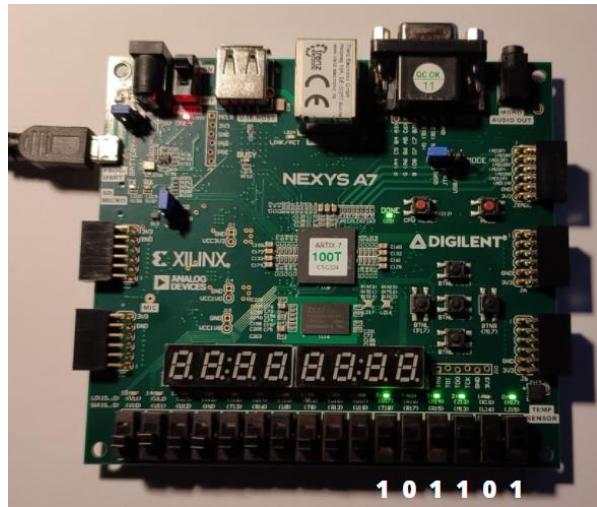
```
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { SWITCH[0] }];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { SWITCH[1] }];
#IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { SWITCH[2] }];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { SWITCH[3] }];
#IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { SWITCH[4] }];
#IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { SWITCH[5] }];
#IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { SWITCH[6] }];
#IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { SWITCH[7] }];
#IO_L5N_T0_D07_14 Sch=sw[7]
```

LEDs

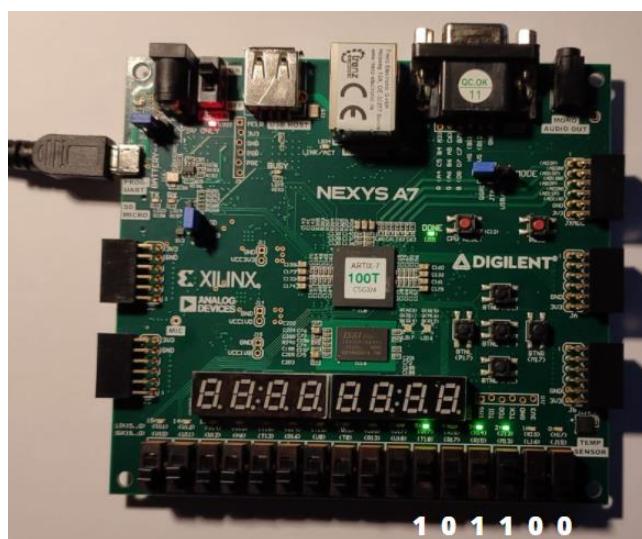
```
set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports { LED[0] }];
#IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [get_ports { LED[1] }];
#IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [get_ports { LED[2] }];
#IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [get_ports { LED[3] }];
#IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18 IOSTANDARD LVCMOS33 } [get_ports { LED[4] }];
#IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMOS33 } [get_ports { LED[5] }];
#IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMOS33 } [get_ports { LED[6] }];
#IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports { LED[7] }];
#IO_L18P_T2_A12_D28_14 Sch=led[7]
```

##Buttons

```
set_property -dict { PACKAGE_PIN P18 IOSTANDARD LVCMOS33 } [get_ports { BTN_RESET }];
#IO_L9N_T1_DQS_D13_14 Sch=btnd
set_property -dict { PACKAGE_PIN N17 IOSTANDARD LVCMOS33 } [get_ports { BTN_LOAD }];
#IO_L9P_T1_DQS_14 Sch=btnc
```

9.1.6 Implementazione sulla board

Impostiamo il trasmettitore col dato 45 ("00101101") sugli switch e dimostriamo il corretto funzionamento dell'interfaccia seriale, in quanto il dato ricevuto dal ricevitore è mostrato sui led e corrisponde esattamente al dato inviato ovvero 45. ("00101101")



Impostiamo il trasmettitore col dato 44 ("00101100") sugli switch e dimostriamo il corretto funzionamento dell'interfaccia seriale, in quanto il dato ricevuto dal ricevitore è mostrato sui led e corrisponde esattamente al dato inviato ovvero 44. ("00101100")

9.2 Esercizio “2_UART_MEM”

a) 2_UART_MEM: come variante dell'esercizio 9.1, il sistema A invia al sistema B tramite l'interfaccia seriale N stringhe di 8 bit contenute all'interno di una memoria ROM. Le stringhe ricevute vengono memorizzate in una memoria locale a B. Il progetto deve prevedere che A utilizzi un componente contatore per scandire le N stringhe da inviare.

9.2.1 Descrizione del funzionamento di “2_UART_MEM”

Partendo dall'esercizio nel paragrafo 9.1, utilizziamo l'interfaccia seriale UART per la comunicazione tra due entità, A e B, con lo scopo di sincronizzare l'invio da parte del trasmettitore, così da evitare errori di overrun.

I 3 componenti principali, come mostrati in Figura 9.6, del nostro progetto sono:

- UART, interfaccia seriale
- A, nodo trasmettitore
- B, nodo ricevitore

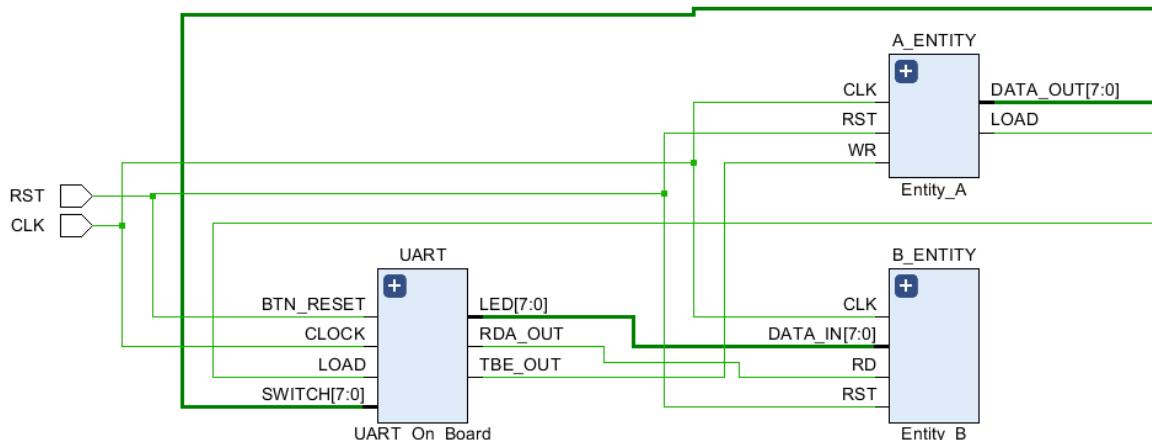


FIG. 9.6

Il componente UART è composto dai componenti visti nel paragrafo 9.1

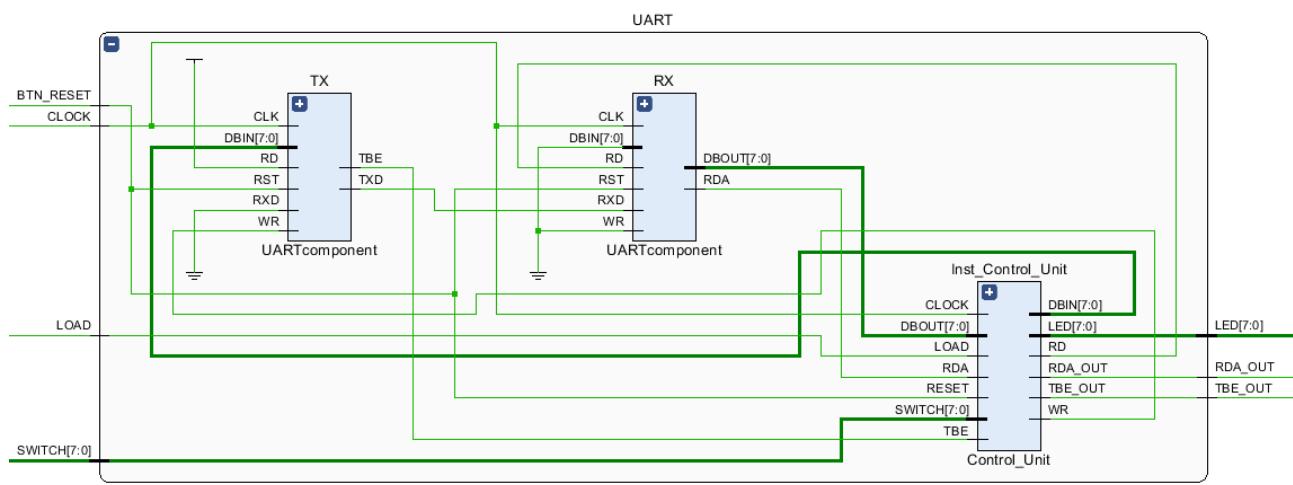


FIG. 9.7

Il Componente A è composto invece da un contatore, una ROM e un'unità di controllo. Invece, il componente B è composto da un counter, un'unità di controllo e una memoria.

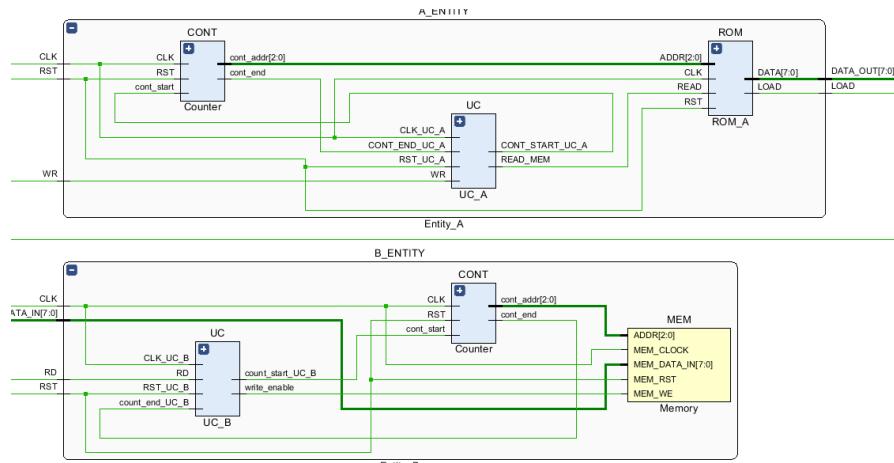


FIG. 9.8

9.2.2 Codice VHDL

UART A B.vhd

BEGIN

```

A_ENTITY : ENTITY_A
PORT MAP (
    CLK => CLK,
    RST => RST,
    --
    WR => TBE_TEMP,
    DATA_OUT => DATA_OUT_TEMP,
    LOAD => LOAD_TEMP
);
B_ENTITY : ENTITY_B
PORT MAP (
    CLK => CLK,
    RST => RST,
    --
    RD => RDA_TEMP,
    DATA_IN => DATA_IN_TEMP
);
UART : UART_On_Board
PORT MAP (
    CLOCK => CLK,
    BTN_RESET => RST,
    LOAD => LOAD_TEMP,
    SWITCH => DATA_OUT_TEMP,
    LED => DATA_IN_TEMP,
    --
    TBE_OUT => TBE_TEMP,
    RDA_OUT => RDA_TEMP
);

```

Entity_A.vhd

```
ENTITY Entity_A IS
  GENERIC (
    M : INTEGER := 8
  );
  PORT (
    --ingressi
    CLK : IN STD_LOGIC;
    RST : IN STD_LOGIC;
    WR_IN : IN STD_LOGIC;
    SR_ACK : IN STD_LOGIC;
    DONE : IN STD_LOGIC;
    --uscite
    LOAD : OUT STD_LOGIC;
    SR : OUT STD_LOGIC;
    --data out ROM
    data_out : OUT STD_LOGIC_VECTOR(M - 1 DOWNTO 0)
  );
END Entity_A;
```

```
ARCHITECTURE Structural OF Entity_A IS
```

```
COMPONENT Counter IS
  PORT (
    CLK : IN STD_LOGIC;
    RST : IN STD_LOGIC;
    cont_addr : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
    --dall'unita' di controllo
    cont_start : IN STD_LOGIC
  );
END COMPONENT;
```

```
COMPONENT ROM_A IS
  GENERIC (
    M : INTEGER := 8
  );
  PORT (
    CLK : IN STD_LOGIC; -- clock della board
    RST : IN STD_LOGIC;
    READ : IN STD_LOGIC := '0'; -- segnale da uc
    ADDR : IN STD_LOGIC_VECTOR(2 DOWNTO 0); --3 bit di
indirizzo per accedere agli elementi della ROM,
    --sono inseriti tramite gli switch (DA VEDERE LOG2N)
    DATA : OUT STD_LOGIC_VECTOR(M - 1 DOWNTO 0) -- dato su 8
bit letto dalla ROM
  );
END COMPONENT;
```

```
COMPONENT UC_A IS
  PORT (
    CLK_UC_A : IN STD_LOGIC;
    RST_UC_A : IN STD_LOGIC;
    SR_ACK : IN STD_LOGIC;
    DONE : IN STD_LOGIC;
    WR_IN : IN STD_LOGIC;
```



```
        CONT_START_UC_A : OUT STD_LOGIC;
        SR : OUT STD_LOGIC;
        LOAD : OUT STD_LOGIC
    );
END COMPONENT;

--segnali interni
SIGNAL address_in : STD_LOGIC_VECTOR(2 DOWNTO 0) := "000";
SIGNAL cont_in : STD_LOGIC := '0';

BEGIN

ROM : ROM_A
PORT MAP(
    CLK => CLK,
    RST => RST,
    ADDR => address_in,
    DATA => DATA_OUT,
    READ => WR_IN
);

CONT : Counter
PORT MAP(
    CLK => CLK,
    RST => RST,
    cont_addr => address_in,
    cont_start => cont_in
);

UC : UC_A
PORT MAP(
    CLK_UC_A => CLK,
    RST_UC_A => RST,
    CONT_START_UC_A => cont_in,
    SR_ACK => SR_ACK,
    WR_IN => WR_IN,
    DONE => DONE,
    SR => SR,
    LOAD => LOAD
);
END Structural;
```

UC_A.vhd

```
ENTITY UC_A IS
    PORT (
        CLK_UC_A : IN STD_LOGIC;
        RST_UC_A : IN STD_LOGIC;
        --
        SR_ACK : IN STD_LOGIC;
        DONE : IN STD_LOGIC;
        WR_IN : IN STD_LOGIC;
        CONT_START_UC_A : OUT STD_LOGIC;
        SR : OUT STD_LOGIC := '0';
        LOAD : OUT STD_LOGIC
    );

```



```
END UC_A;
```

```
ARCHITECTURE Behavioral OF UC_A IS
```

```
    TYPE state IS (idle, send_request, write_strobe, wait_done);
    SIGNAL current_state : state := idle;

BEGIN

    p : PROCESS (CLK_UC_A, RST_UC_A)
    BEGIN
        --automa UC_A
        IF (RST_UC_A = '1') THEN
            current_state <= idle;

        ELSIF (CLK_UC_A'event AND CLK_UC_A = '1') THEN
            CASE current_state IS

                WHEN idle =>
                    CONT_START_UC_A <= '0';
                    LOAD <= '0';
                    SR <= '0';
                    IF (WR_IN = '1') THEN
                        current_state <= send_request;
                    ELSE
                        current_state <= idle;
                    END IF;

                WHEN send_request =>
                    CONT_START_UC_A <= '0';
                    LOAD <= '0';
                    SR <= '1';
                    IF (SR_ACK = '0') THEN
                        current_state <= send_request;
                    ELSE
                        current_state <= write_strobe;
                    END IF;

                WHEN write_strobe =>
                    CONT_START_UC_A <= '1';
                    LOAD <= '1';
                    SR <= '0';
                    current_state <= wait_done;

                WHEN wait_done =>
                    CONT_START_UC_A <= '0';
                    LOAD <= '0';
                    SR <= '0';
                    IF (DONE = '0') THEN
                        current_state <= wait_done;
                    ELSE
                        current_state <= idle;
                    END IF;
            END CASE;
        END IF;
    END PROCESS p;
END;
```



```
        END IF;
    END CASE;

    END IF;
END PROCESS;

END Behavioral;
```

Entity_B.vhd

```
ENTITY Entity_B IS
    GENERIC (
        M : INTEGER := 8
    );
    PORT (
        -- ingressi
        SR : IN STD_LOGIC;
        RDA : IN STD_LOGIC;
        CLK : IN STD_LOGIC;
        RST : IN STD_LOGIC;
        -- dato da scrivere in RAM
        data_in : IN STD_LOGIC_VECTOR(M - 1 DOWNTO 0);
        --uscite
        REQACK : OUT STD_LOGIC;
        DONE : OUT STD_LOGIC
    );
END Entity_B;

ARCHITECTURE Behavioral OF Entity_B IS

COMPONENT Memory IS
    GENERIC (
        N : INTEGER := 8;
        M : INTEGER := 8
    );
    PORT (
        ADDR : IN STD_LOGIC_VECTOR(2 DOWNTO 0); -- Address to
write/read MEM
        MEM_DATA_IN : IN STD_LOGIC_VECTOR(M - 1 DOWNTO 0); --
Data to write into MEM
        MEM_WE : IN STD_LOGIC; -- Write enable
        MEM_CLOCK : IN STD_LOGIC; -- clock input for MEM
        MEM_RST : IN STD_LOGIC -- reset for MEM
    );
END COMPONENT;

COMPONENT Counter IS
    PORT (
        CLK : IN STD_LOGIC;
        RST : IN STD_LOGIC;
        cont_addr : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        --dall'unita' di controllo
        cont_start : IN STD_LOGIC
    );
END COMPONENT;
```

```
COMPONENT UC_B IS
  PORT (
    CLK_UC_B : IN STD_LOGIC; -- clock della board
    RST_UC_B : IN STD_LOGIC;
    ---gestione del contatore
    RDA : IN STD_LOGIC;
    SR : IN STD_LOGIC;
    count_start_UC_B : OUT STD_LOGIC;
    REQACK : OUT STD_LOGIC;
    DONE : OUT STD_LOGIC;
    write_enable : OUT STD_LOGIC
  );
END COMPONENT;
--segnali interni
SIGNAL address_in : STD_LOGIC_VECTOR(2 DOWNTO 0) := "000";
SIGNAL cont_in : STD_LOGIC := '0';
SIGNAL we : STD_LOGIC := '0';

BEGIN

  MEM : Memory
  GENERIC MAP(
    N => 8,
    M => 8
  )
  PORT MAP(
    ADDR => address_in,
    MEM_DATA_IN => DATA_IN,
    MEM_WE => we,
    MEM_CLOCK => CLK,
    MEM_RST => RST
  );
  UC : UC_B
  PORT MAP(
    CLK_UC_B => CLK,
    RST_UC_B => RST,
    RDA => RDA,
    SR => SR,
    REQACK => REQACK,
    DONE => DONE,
    count_start_UC_B => cont_in,
    write_enable => we
  );
  CONT : counter
  PORT MAP(
    CLK => CLK,
    RST => RST,
    cont_addr => address_in,
    cont_start => cont_in
  );

END Behavioral;
```

UC_B.vhd

```
ENTITY UC_B IS
  PORT (
    CLK_UC_B : IN STD_LOGIC; -- clock della board
    RST_UC_B : IN STD_LOGIC;
    ---gestione del contatore
    RDA : IN STD_LOGIC;
    SR : IN STD_LOGIC;
    count_start_UC_B : OUT STD_LOGIC;
    REQACK : OUT STD_LOGIC;
    DONE : OUT STD_LOGIC;
    write_enable : OUT STD_LOGIC
  );
END UC_B;

ARCHITECTURE Behavioral OF UC_B IS

  TYPE state IS (idle, send_ack, wait_rda, read_done, write_done);
  SIGNAL current_state : state := idle;

BEGIN

  p : PROCESS (CLK_UC_B, RST_UC_B)
  BEGIN
    --automa UC_B
    IF (RST_UC_B = '1') THEN
      current_state <= idle;

    ELSIF (CLK_UC_B'event AND CLK_UC_B = '1') THEN
      CASE current_state IS
        WHEN idle =>
          count_start_UC_B <= '0';
          REQACK <= '0';
          DONE <= '0';
          IF (SR = '0') THEN
            current_state <= idle;
          ELSE
            current_state <= send_ack;
          END IF;

        WHEN send_ack =>
          count_start_UC_B <= '0';
          REQACK <= '1';
          DONE <= '0';
          current_state <= wait_rda;

        WHEN wait_rda =>
          count_start_UC_B <= '0';
          REQACK <= '1';
          DONE <= '0';
      END CASE;
    END IF;
  END PROCESS p;
END;
```



```
        IF (RDA = '0') THEN
            current_state <= wait_rda;
        ELSE
            current_state <= read_done;
        END IF;

        WHEN read_done =>

            write_enable <= '1';
            count_start_UC_B <= '1';
            REQACK <= '0';
            DONE <= '0';
            current_state <= write_done;

        WHEN write_done =>

            count_start_UC_B <= '0';
            REQACK <= '0';
            write_enable <= '0';
            DONE <= '1';
            current_state <= idle;

        END CASE;

    END IF;
END PROCESS;
```

END Behavioral;

I componenti Counter, ROM e Memory si sono omessi perché uguali a quelli usati nei progetti precedenti.

9.2.3 Test Bench

tb_UART_A_B.vhd

```
ENTITY tb_UART_A_B IS
END tb_UART_A_B;

ARCHITECTURE tb OF tb_UART_A_B IS

    COMPONENT UART_A_B
        PORT (
            CLK : IN STD_LOGIC;
            RST : IN STD_LOGIC;
            ENABLE : IN STD_LOGIC);
    END COMPONENT;

    SIGNAL CLK : STD_LOGIC;
    SIGNAL RST : STD_LOGIC;
    SIGNAL ENABLE : STD_LOGIC;

    CONSTANT TbPeriod : TIME := 10 ns; -- EDIT Put right period here
    SIGNAL stoptheclock : BOOLEAN;
```



BEGIN

```
dut : UART_A_B
PORT MAP(
    CLK => CLK,
    RST => RST,
    ENABLE => ENABLE);

stimulus : PROCESS
BEGIN

    RST <= '1';
    WAIT FOR 100ns;
    RST <= '0';

    enable <= '0';
    WAIT FOR 100ns;
    enable <= '1';
    WAIT FOR 50 ns;
    enable <= '0';
    WAIT FOR 100 us;
    enable <= '1';
    WAIT FOR 50 ns;
    enable <= '0';
    WAIT FOR 100 us;
    enable <= '1';
    WAIT FOR 50 ns;
    enable <= '0';
    WAIT FOR 100 us;
    enable <= '1';
    WAIT FOR 50 ns;
    enable <= '0';
    WAIT FOR 100 us;
    enable <= '1';
    WAIT FOR 50 ns;
    enable <= '0';
    WAIT FOR 100 us;
    enable <= '1';
    WAIT FOR 50 ns;
    enable <= '0';
    WAIT FOR 100 us;
    enable <= '1';
    WAIT FOR 50 ns;
    enable <= '0';
    WAIT FOR 100 us;
    enable <= '1';
    WAIT FOR 50 ns;
    enable <= '0';
    WAIT FOR 100 us;
    enable <= '1';
    WAIT FOR 50 ns;
    enable <= '0';
    WAIT FOR 100 us;
    enable <= '1';
    WAIT FOR 50 ns;
    enable <= '0';
    WAIT;
END PROCESS;

clocking : PROCESS
```

```

BEGIN
  WHILE NOT stoptheclock LOOP
    CLK <= '0', '1' AFTER TbPeriod / 2;
    WAIT FOR TbPeriod;
  END LOOP;
  WAIT;
END PROCESS;

```

9.2.4 Simulazione



FIG. 9.9

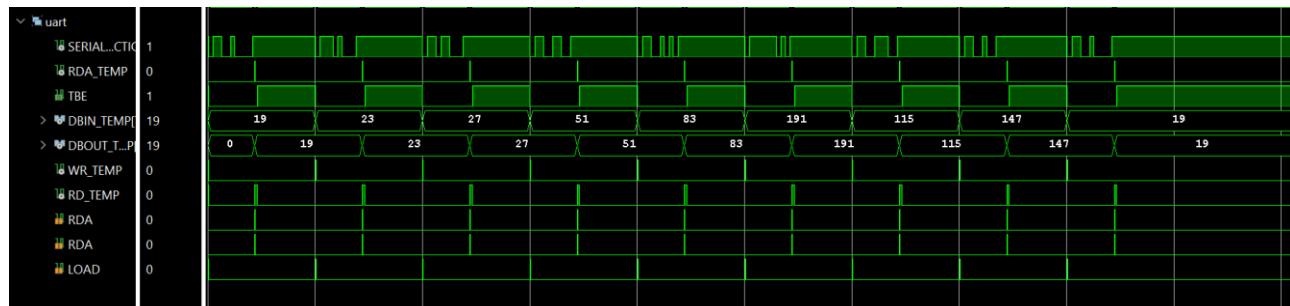


FIG. 9.10

Come si evince dalle **FIGURE**, l'entità A invia all'interfaccia seriale il dato e aspetta che B l'abbia ricevuto e memorizzato nella propria Memoria prima di inviare un nuovo dato.

10 Switch multistadio

10.1 Traccia

Si realizza uno Switch Multistadio seguendo le specifiche richieste del committente:

- Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network. Lo switch progettato deve operare come segue:
 - Lo switch deve consentire lo scambio di messaggi di 2 bit ciascuno da un nodo sorgente a un nodo destinazione in una rete con 4 nodi, implementando uno schema a priorità fissa fra i nodi (ed. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).
 - (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).
 - (Opzionale) Si implementi un protocollo di handshaking semplice regolato da una coppia di segnali (pronto a inviare/pronto a ricevere) per l'invio di ciascun messaggio fra due nodi.

10.1.1 Descrizione del funzionamento dello switch multistadio

Lo Switch multistadio che abbiamo progettato consente lo scambio di messaggi tra un nodo sorgente e un nodo destinazione, attraverso una rete composta da 4 switch.

Il Progetto è composto da una “Parte di Controllo” e una “Parte Operativa” come in **FIG. 10.1**.

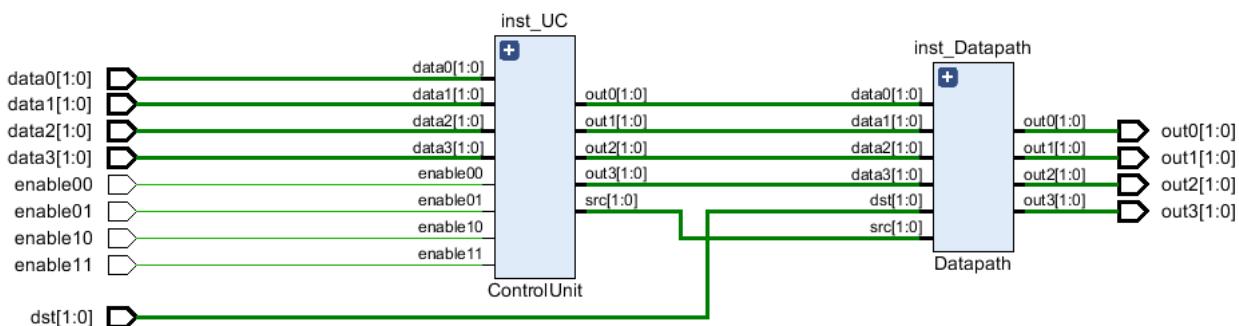
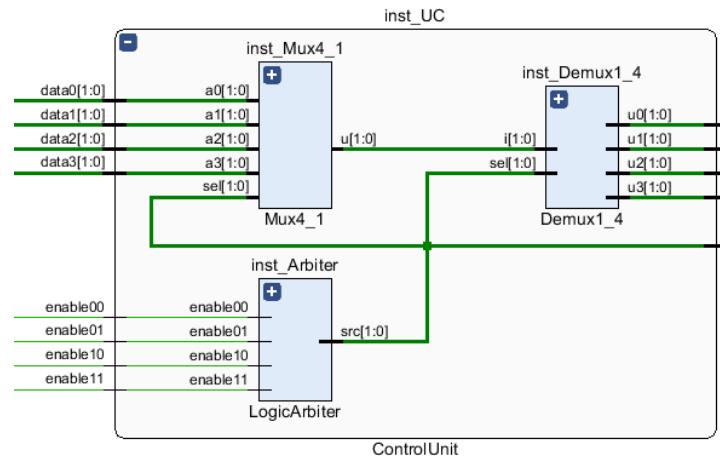


FIG 10.1

La parte di controllo (**FIG 10.2**) è formata da 3 componenti, che sono i seguenti:

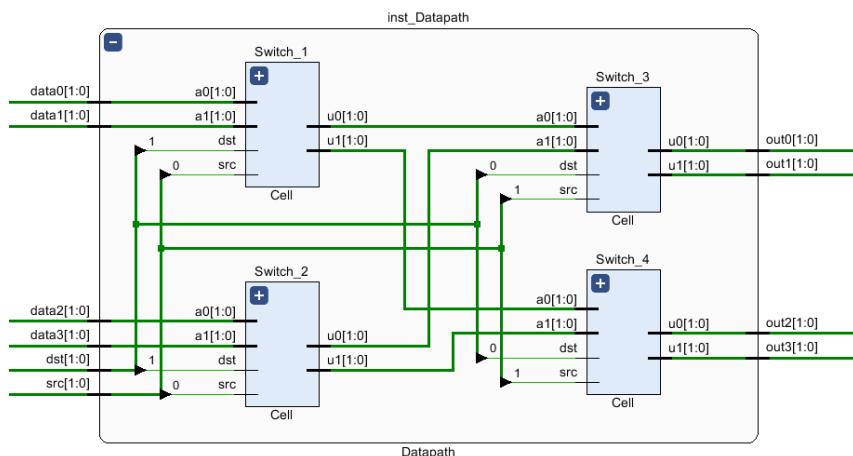
- Arbitro, ha il compito di selezionare la sorgente con priorità maggiore.
- Multiplexer 4:1, ha il compito di selezionare l'uscita in base alla sorgente con priorità maggiore.
- Demultiplexer 1:4, ha il compito di instradare il dato il dato in ingresso in base alla sorgente.

In ingresso alla parte di controllo abbiamo l'abilitazione e il dato da inviare

**FIG 10.2** ControlUnit

La parte operativa (**FIG 10.3**) è composta da 4 switch con numero di stadi pari a 2. Ogni switch è composto da una DEMUX e MUX in ingresso hanno sorgente, destinazione e due dati e in uscita hanno il valore selezionato in base alla sorgente e destinazione.

In ingresso abbiamo sorgente, destinazione e il dato da instradare. L'uscita invece sarà il dato all'uscita corrispondente alla destinazione desiderata.

**FIG 10.3** Datapath

10.1.2 Codice VHDL

Switch_Multistadio.vhd

```
ENTITY Switch_Multistadio IS
  PORT (
    enable00 : IN STD_LOGIC;
    enable01 : IN STD_LOGIC;
    enable10 : IN STD_LOGIC;
    enable11 : IN STD_LOGIC;
    data0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    data1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    data2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    data3 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    dst : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
```

```

        out0 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        out1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        out2 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        out3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
    );
END Switch_Multistadio;

```

ARCHITECTURE Structural **OF** Switch_Multistadio **IS**

```

COMPONENT Datapath IS
    PORT (
        dst : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        src : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        data0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        data1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        data2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        data3 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        out0 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        out1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        out2 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        out3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
    );
END COMPONENT;

```

```

COMPONENT ControlUnit IS
    PORT (
        data0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        data1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        data2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        data3 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        enable00 : IN STD_LOGIC;
        enable01 : IN STD_LOGIC;
        enable10 : IN STD_LOGIC;
        enable11 : IN STD_LOGIC;
        src : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        out0 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        out1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        out2 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        out3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
    );
END COMPONENT;

```

```

SIGNAL src_temp : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL data0_temp : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL data1_temp : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL data2_temp : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL data3_temp : STD_LOGIC_VECTOR(1 DOWNTO 0);

```

BEGIN

```
inst_Datapath : Datapath
PORT MAP(
    dst => dst,
    src => src_temp,
    data0 => data0_temp,
    data1 => data1_temp,
    data2 => data2_temp,
    data3 => data3_temp,
    out0 => out0,
    out1 => out1,
    out2 => out2,
    out3 => out3
);

inst_UC : ControlUnit
PORT MAP(
    data0 => data0,
    data1 => data1,
    data2 => data2,
    data3 => data3,
    enable00 => enable00,
    enable01 => enable01,
    enable10 => enable10,
    enable11 => enable11,
    src => src_temp,
    out0 => data0_temp,
    out1 => data1_temp,
    out2 => data2_temp,
    out3 => data3_temp
);

END Structural;
```

LogicArbiter.vhd

```
ENTITY LogicArbiter IS
    PORT (
        enable00 : IN STD_LOGIC;
        enable01 : IN STD_LOGIC;
        enable10 : IN STD_LOGIC;
        enable11 : IN STD_LOGIC;
        src : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
    );
END LogicArbiter;
```

```
ARCHITECTURE Dataflow OF LogicArbiter IS
```

```
BEGIN
```



```
src <= "00" WHEN enable00 = '1' ELSE
    "01" WHEN enable01 = '1' ELSE
    "10" WHEN enable10 = '1' ELSE
    "11" WHEN enable11 = '1' ELSE
    "--";
END Dataflow;
```

Inst_UC.vhd

```
BEGIN
```

```
src <= src_temp;

inst_Mux4_1 : Mux4_1
PORT MAP(
    a0 => data0,
    a1 => data1,
    a2 => data2,
    a3 => data3,
    sel => src_temp,
    u => temp
);

inst_Demux1_4 : Demux1_4
PORT MAP(
    sel => src_temp,
    i => temp,
    u0 => out0,
    u1 => out1,
    u2 => out2,
    u3 => out3
);

inst_Arbiter : LogicArbiter
PORT MAP(
    enable00 => enable00,
    enable01 => enable01,
    enable10 => enable10,
    enable11 => enable11,
    src => src_temp
);
END Structural;
```

Inst_Datapath.vhd

```
BEGIN
```

```
Switch_1 : Cell PORT MAP(
    a0 => data0,
    a1 => data1,
    src => src(0),
```



```

        dst => dst(1),
        u0 => temp0,
        u1 => temp1
    );
Switch_3 : Cell PORT MAP(
    a0 => temp0,
    a1 => temp2,
    src => src(1),
    dst => dst(0),
    u0 => out0,
    u1 => out1
);
Switch_2 : Cell PORT MAP(
    a0 => data2,
    a1 => data3,
    src => src(0),
    dst => dst(1),
    u0 => temp2,
    u1 => temp3
);
Switch_4 : Cell PORT MAP(
    a0 => temp1,
    a1 => temp3,
    src => src(1),
    dst => dst(0),
    u0 => out2,
    u1 => out3
);
END structural;

```

Cell.vhd (singolo Switch)

```

ENTITY Cell IS
    PORT (
        --ingressi
        a0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        a1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        src : IN STD_LOGIC;
        dst : IN STD_LOGIC;
        --uscite
        u0 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        u1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
    );
END Cell;

```

ARCHITECTURE Structural OF Cell IS

```

COMPONENT Mux2_1
    PORT (
        a0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        a1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        sel : IN STD_LOGIC;
        u : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
    );

```



```

        );
END COMPONENT;

COMPONENT Demux1_2
PORT (
    i : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    sel : IN STD_LOGIC;
    u0 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
    u1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END COMPONENT;

SIGNAL temp : STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN

    Inst_Mux2_1 : Mux2_1 PORT MAP(
        a0 => a0,
        a1 => a1,
        sel => src,
        u => temp
    );
    Inst_Demux1_2 : Demux1_2 PORT MAP(
        i => temp,
        sel => dst,
        u0 => u0,
        u1 => u1
    );
END Structural;

```

I componenti MUX e DEMUX non sono stati riportati in quanto sono i classici componenti usati anche nei precedenti progetti.

10.1.3 Testbench

```

stimuli : PROCESS
BEGIN
    -- EDIT Adapt initialization as needed
    enable00 <= '0';
    enable01 <= '0';
    enable10 <= '0';
    enable11 <= '0';
    data0 <= (OTHERS => '0');
    data1 <= (OTHERS => '0');
    data2 <= (OTHERS => '0');
    data3 <= (OTHERS => '0');
    dst <= (OTHERS => '0');

    WAIT FOR 100ns;

    -- EDIT Add stimuli here
    enable00 <= '1';
    enable01 <= '1';
    enable10 <= '0';

```



```
enable11 <= '0';
data0 <= "10";
data1 <= "11";
data2 <= "00";
data3 <= "01";
dst <= "11";

WAIT FOR 100ns;

enable00 <= '0';
enable01 <= '1';
enable10 <= '1';
enable11 <= '0';
data0 <= "01";
data1 <= "11";
data2 <= "11";
data3 <= "10";
dst <= "00";

WAIT FOR 100ns;

enable00 <= '0';
enable01 <= '0';
enable10 <= '1';
enable11 <= '1';
data0 <= "10";
data1 <= "11";
data2 <= "10";
data3 <= "01";
dst <= "10";

WAIT FOR 100ns;

enable00 <= '0';
enable01 <= '0';
enable10 <= '0';
enable11 <= '1';
data0 <= "11";
data1 <= "01";
data2 <= "10";
data3 <= "01";
dst <= "01";

WAIT;
END PROCESS;
```

10.1.4 Simulazione



FIG 10.4

Istante 100ns:

Inseriamo la destinazione pari a “11”, poniamo a ‘1’ i segnali enable00 e enable01. La PC determinerà la sorgente pari a 0. Dopodiché la PO stampa in output il valore corrispondente a data0 all’uscita out3.

Istante 200ns:

Inseriamo la destinazione pari a “00, poniamo a ‘1’ i segnali enable10 e enable01. La PC determinerà la sorgente pari a 1. Dopodiché la PO stampa in output il valore corrispondente a data1 all’uscita out0.

Istante 300ns:

Inseriamo la destinazione pari a “10”, poniamo a ‘1’ i segnali enable10 e enable11. La PC determinerà la sorgente pari a 2. Dopodiché la PO stampa in output il valore corrispondente a data2 all’uscita out2.

Istante 400ns:

Inseriamo la destinazione pari a “01”, poniamo a ‘1’ il segnale enable11. La PC determinerà la sorgente pari a 3. Dopodiché la PO stampa in output il valore corrispondente a data3 all’uscita out1.

10.2 Switch con protocollo di handshaking

(Opzionale) Si implementi un protocollo di handshaking semplice regolato da una coppia di segnali (pronto a inviare/pronto a ricevere) per l’invio di ciascun messaggio fra due nodi.

10.2.1 Descrizione del funzionamento dello switch multistadio con handshaking

Il progetto, come mostrato in Figura 10.5, è composto da:

- Nodo A, ha in input la sorgente e il dato da inviare. Il nodo A può trovarsi in 4 stati (idle, invia_richiesta, aspetta_risposta, invia), se ha successo la comunicazione con il nodo B va nello stato “invia”, passa il dato e la sorgente in ingresso al DMUX.
- Nodo B, ha in input la destinazione. Il nodo B può trovarsi in 4 stati (idle, ricevi_richiesta, invia_risposta), se ha successo la comunicazione con il nodo A va nello stato “invia_risposta”, e riceve il dato dalla destinazione in ingresso al MUX.
- DMUX, ha come compito di instradare il dato in base alla sorgente allo switch multistadio.
- Switch multistadio, che ha il compito di inviare il dato alla destinazione selezionata.

- MUX, seleziona l'output della destinazione selezionata.

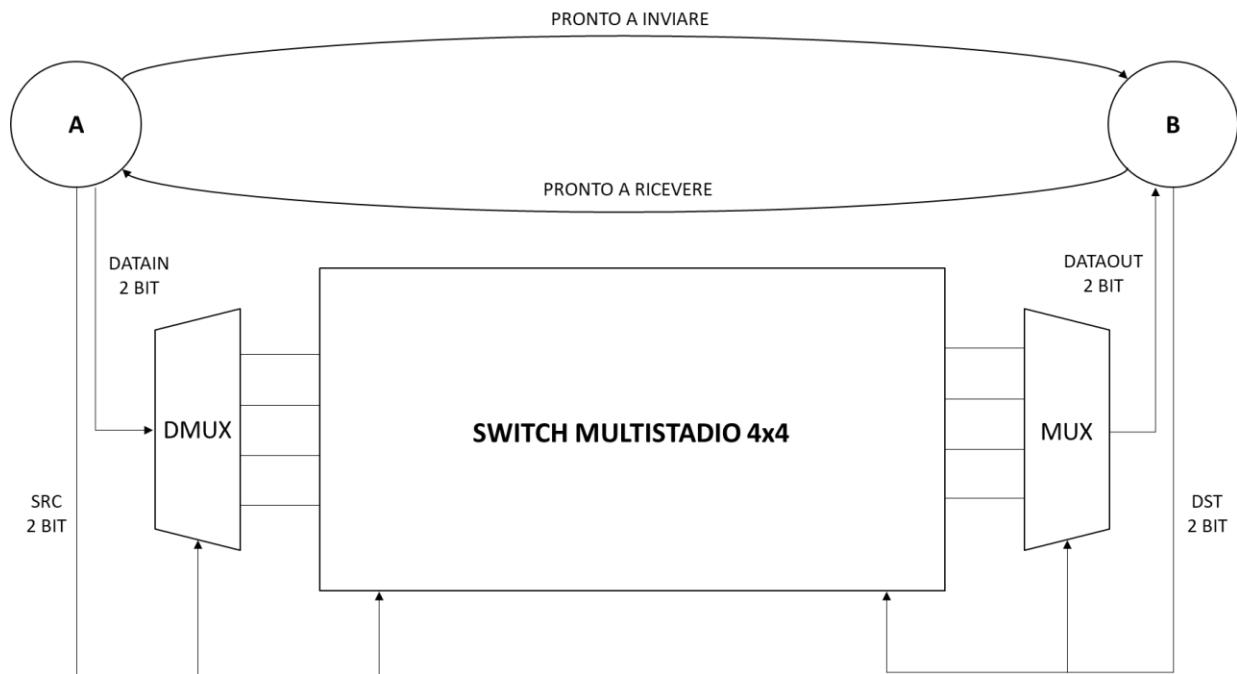


FIG 10.5

10.2.2 Schematico in RTL Analysis di Vivado

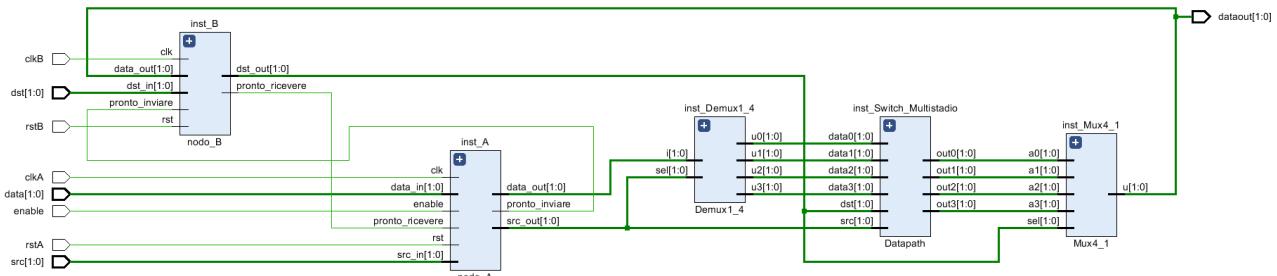


FIG 10.6

10.2.3 Codice VHDL

Switch_Handshaking.vhd

BEGIN

```

dataout <= temp;

inst_Switch_Multistadio : Datapath
PORT MAP (
    dst => dst_temp,
    src => src_temp,
    data0 => data0_temp,
    data1 => data1_temp,
    data2 => data2_temp,
    data3 => data3_temp,
    out0 => out0_temp,
    )
  
```



```
        out1 => out1_temp,
        out2 => out2_temp,
        out3 => out3_temp
    ) ;

inst_A : nodo_A
PORT MAP(
    clk => clkA,
    rst => rstA,
    enable => enable,
    pronto_ricevere => pronto_ricevere_temp,
    src_in => src,
    src_out => src_temp,
    data_in => data,
    data_out => data_temp,
    pronto_inviare => pronto_inviare_temp
) ;

inst_B : nodo_B
PORT MAP(
    clk => clkB,
    rst => rstB,
    pronto_ricevere => pronto_ricevere_temp,
    dst_in => dst,
    dst_out => dst_temp,
    data_out => temp,
    pronto_inviare => pronto_inviare_temp
) ;

inst_Mux4_1 : Mux4_1
PORT MAP(
    a0 => out0_temp,
    a1 => out1_temp,
    a2 => out2_temp,
    a3 => out3_temp,
    sel => dst_temp,
    u => temp
) ;

inst_Demux1_4 : Demux1_4
PORT MAP(
    u0 => data0_temp,
    u1 => data1_temp,
    u2 => data2_temp,
    u3 => data3_temp,
    sel => src_temp,
    i => data_temp
) ;

END Structural;
```

Inst_A.vhd**BEGIN****p : PROCESS** (clk, rst)**BEGIN**

--automa nodo_A

IF (rst = '1') **THEN**

current_state <= idle;

ELSIF (clk'event **AND** clk = '1') **THEN****CASE** current_state **IS****WHEN** idle =>

src_out <= "##";

data_out <= "##";

pronto_inviare <= '0';

IF (enable = '1') **THEN**

current_state <= invia_richiesta;

ELSE

current_state <= idle;

END IF;**WHEN** invia_richiesta =>

src_out <= "##";

data_out <= "##";

pronto_inviare <= '1';

current_state <= aspetta_risposta;

WHEN aspetta_risposta =>

src_out <= "##";

data_out <= "##";

pronto_inviare <= '1';

IF (pronto_ricevere = '1') **THEN**

current_state <= invia;

ELSE

current_state <= aspetta_risposta;

END IF;**WHEN** invia =>

src_out <= src_in;

data_out <= data_in;

pronto_inviare <= '0';

current_state <= idle;

END CASE;**END IF;****END PROCESS;****END Behavioral;**

Inst_B.vhd

```

BEGIN

p : PROCESS (clk, rst)
BEGIN
    --automa nodo_A
    IF (rst = '1') THEN
        current_state <= idle;

    ELSIF (clk'event AND clk = '1') THEN

        CASE current_state IS

            WHEN idle =>

                dst_out <= "--";
                pronto_ricevere <= '0';
                IF (pronto_inviare = '1') THEN
                    current_state <= ricevi_richiesta;
                ELSE
                    current_state <= idle;
                END IF;

            WHEN ricevi_richiesta =>

                dst_out <= "--";
                pronto_ricevere <= '1';
                current_state <= invia_risposta;

            WHEN invia_risposta =>

                dst_out <= dst_in;
                pronto_ricevere <= '0';
                current_state <= idle;

        END CASE;

    END IF;
END PROCESS;
END Behavioral;

```

10.2.4 Test Bench***tb_Switch_Handshaking.vhd***

```

BEGIN

dut : Switch_Handshaking
PORT MAP(
    clkA => clkA,
    rstA => rstA,
    clkB => clkB,
    rstB => rstB,
    enable => enable,

```



```
src => src,
dst => dst,
data => data,
dataout => dataout);

-- Clock generation
TbClockA <= NOT TbClockA AFTER TbPeriodA/2 WHEN TbSimEndedA /= '1' ELSE
'0';
TbClockB <= NOT TbClockB AFTER TbPeriodB/2 WHEN TbSimEndedB /= '1' ELSE
'0';

-- EDIT: Check that clkA is really your main clock signal
clkA <= TbClockA;
clkB <= TbClockB;

stimuli : PROCESS
BEGIN
    -- EDIT Adapt initialization as needed
    --      clkA <= '0';
    --      rstA <= '0';
    --      clkB <= '0';
    --      rstB <= '0';
    enable <= '0';
    src <= (OTHERS => '0');
    dst <= (OTHERS => '0');
    data <= (OTHERS => '0');

    -- Reset generation
    -- EDIT: Check that rstA is really your reset signal
    rstA <= '1';
    rstB <= '1';
    WAIT FOR 100 ns;
    rstA <= '0';
    rstB <= '0';
    WAIT FOR 100 ns;

    -- EDIT Add stimuli here
    enable <= '1';
    src <= "01";
    dst <= "11";
    data <= "01";

    WAIT FOR 50ns;

    enable <= '0';

    WAIT FOR 50ns;

    enable <= '1';
    src <= "00";
    dst <= "10";
    data <= "11";

    WAIT FOR 50ns;

    enable <= '0';
```

```

WAIT FOR 50ns;

enable <= '1';
src <= "01";
dst <= "00";
data <= "10";

WAIT FOR 50ns;

enable <= '0';

WAIT FOR 100 * TbPeriodA;
WAIT FOR 100 * TbPeriodB;

-- Stop the clock and hence terminate the simulation
TbSimEndedA <= '1';
TbSimEndedB <= '1';
WAIT;
END PROCESS;

END tb;

```

10.2.5 Simulazione



FIG 10.7

Istante 200ns:

Alzo il segnale di enable, pongo la destinazione a "11" e la sorgente a "01" e il dato è pari a "01", dopodiché viene effettuato l'handshaking, ed il dato viene posto in out3.

Istante 250ns:

Pongo l'enable a "0" e il valore in uscita è pari al dato inserito.

Istante 300ns:

Alzo il segnale di enable, pongo la destinazione a "10" e la sorgente a "00" e il dato è pari a "11", dopodiché viene effettuato l'handshaking, ed il dato viene posto in out2.

Istante 350ns:

Pongo l'enable a "0" e il valore in uscita è pari al dato inserito.

Istante 400ns:

Alzo il segnale di enable, pongo la destinazione a "00" e la sorgente a "01" e il dato è pari a "10", dopodiché viene effettuato l'handshaking, ed il dato viene posto in out0.

Istante 450ns:

Pongo l'enable a "0" e il valore in uscita è pari al dato inserito.

11 Moltiplicatore di Robertson

Si realizza una macchina aritmetica seguendo le specifiche richieste dal cliente:

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

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

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

11.1 Progettazione del Moltiplicatore di Robertson

Il moltiplicatore di Robertson è un moltiplicatore sequenziale, per capirne il funzionamento, analizziamo come si effettua la moltiplicazione in binario, passo per passo, cominciando dall'algoritmo manuale di moltiplicazione binaria. Ad esempio, se dobbiamo moltiplicare 0111 per 0111, prendiamo il moltiplicando, lo moltiplichiamo per la cifra i-esima del moltiplicatore e poi dobbiamo sommare delle righe che non hanno sempre lo stesso peso ma sono ogni volta shiftate di una posizione a sinistra. Dunque, prima generiamo le righe e poi ne facciamo la somma, se entrambi gli operandi hanno dimensione n allora il prodotto ha dimensione $2n$. Ad ogni passo prendiamo la cifra i-esima, la moltiplichiamo per il moltiplicando, poi ogni volta moltiplichiamo per un fattore 2^i , ovvero facciamo uno shift a sinistra di i posizioni.

Otteniamo la seguente sommatoria:

$$P = \sum_{i=0}^{n-1} x_i 2^i Y$$

Ogni termine della sommatoria è composto dalla cifra i-esima del moltiplicatore, dal moltiplicando Y, ed un termine 2^i che rappresenta lo shift della posizione i-esima. Possiamo modificare la procedura manuale in una procedura iterativa, attraverso la conservazione di tutte le righe e poi facendo la somma: consideriamo come somma parziale iniziale tutti 0 e poi sommiamo il prodotto $2^i x_i$, il prodotto ogni volta è shiftato di una posizione a sinistra. La somma non la facciamo alla fine per n righe ma iterativamente per ogni riga. Se dobbiamo trasformarlo in un algoritmo, al passo i-esimo il numero di shift dipende da i, ovvero non è fisso. Possiamo fare ancora meglio, invertiamo l'ordine della somma e degli shift, facciamo prima la somma parziale e poi lo shift a destra di una posizione:

$$P_i = P_{i-1} + x_i Y$$

$$P_{i+1} = 2^{-1} P_i$$

Osserviamo la figura 11.1, in rosso troviamo il prodotto tra la cifra i-esima del moltiplicatore e il moltiplicando; quindi, se la cifra è 1 è pari al moltiplicando, se la cifra è 0 è pari a tutti 0. P0 è il prodotto parziale al passo 0, dunque il prodotto parziale iniziale, pari alla stringa di 0. Dato che la



prima cifra del moltiplicatore è 1, dobbiamo semplicemente sommare 0111. Normalmente alla successiva cifra considereremmo il valore (anche in questo caso 0111) shiftato di una posizione a sinistra. Ora invece prendiamo la somma di 00000000 con 0111, shiftato di una posizione a destra (P1), questo equivale ad allineare il successivo prodotto sotto le cifre giuste, per fare la somma. Facendo così ci ritroviamo che, con l'ultimo shift, abbiamo un prodotto che possiamo vedere sugli 8 bit in blu, infatti abbiamo che $0111 \times 0111 = 00110001$, ovvero $7 \times 7 = 49$.

Y 0111	
X 0111	
0000 0000	P0
0111	
0000 0111	ADD;shift
000 0011	1 P1
0111	
000 1010	1 ADD;shift
00 0101	01 P2
0111	
00 1100	01 ADD;shift
0 0110	001 P3
0000	
0 0110	001 ADD;shift
0011	0001
A	Q

FIG. 11.1: Moltiplicazione binaria

Cerchiamo di immaginare il datapath necessario a una macchina che opera in tal modo. A livello di registri ne serve uno per il moltiplicando, uno per il moltiplicatore e in teoria uno di lunghezza doppia per il risultato. Cerchiamo di ottimizzare le risorse: mettiamo il moltiplicatore in un registro da 4 bit, ad esempio, e il moltiplicatore invece in un registro che ha già lunghezza doppia, quindi 8 bit. Se pensiamo ai bit in blu in figura 11.1, è un registro in tutto grande 8 bit che si riempie piano piano, con A e Q. All'inizio esso potrebbe contenere la somma parziale che sono 8 zeri, poi ad ogni iterazione l'ultimo bit, il meno significativo, viene buttato, dunque man mano delle 8 cifre ne servono meno. Anziché usare un registro a parte per mantenere gli altri dati che ci servono, sfruttiamo il registro in cui le cifre a ogni iterazione si liberano. La porzione meno significativa del registro, Q, all'inizio è vuota e viene riempita man mano che avvengono gli shift, alla fine avremo sugli 8 bit che prima contenevano tutti 0 (prodotto parziale iniziale, P0) un pezzo di A e di Q che infine fanno il risultato. Possiamo fare questo algoritmo in modo sequenziale, inizializziamo P, poi moltiplichiamo il moltiplicando per una cifra, una volta 0 e una 1 (possiamo anche usare un multiplexer per questo, che una volta dà 0 e una volta dà il moltiplicando) poi abbiamo uno shift a destra e una somma, che otteniamo tramite un addizionatore. Come datapath allora avremo un sommatore, un multiplexer, che seleziona lo 0 o il moltiplicando, e il risultato della somma che deve andare in un registro che sia in grado di shiftare, quindi uno shift-register.

Tuttavia, questo tipo di moltiplicatore ha un grande limite, ovvero che non gestisce i numeri relativi e quindi non è possibile effettuare moltiplicazioni con numeri negativi come uno dei due operandi.

11.1.1 Descrizione del funzionamento del Moltiplicatore di Robertson

Dunque, il moltiplicatore di Robertson riceve in ingresso due operandi ad n bit codificati in complementi a due e restituisce un prodotto su $2n$ bit. Tale moltiplicatore sfrutta questa notazione:

$$X = -2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

Per un numero positivo viene assegnato peso 2^{n-1} al bit di segno x_{n-1} : poiché è nullo il suo contributo allo 0 sulla cifra più significativa. Per un numero negativo viene assegnato peso -2^{n-1} al bit di segno x_{n-1} : poiché esso vale 1 il suo contributo al numero è -1 sulla cifra più significativa. Tale metodo ci consente di utilizzare una tecnica unsigned facendo attenzione ad effettuare una sottrazione invece di una addizione quando si incontra un segno negativo. A tal scopo è stato implementato un componente che effettua sia l'addizione che la sottrazione di due operandi a seconda del valore di un bit (0,1) in ingresso. Riassumiamo dunque i 4 possibili casi:

- caso 1.** $X > 0$ e $Y > 0$: caso standard, moltiplicazione tra unsigned fatta con addizione e shift.
- caso 2.** $X > 0$ e $Y < 0$: quando si moltiplica Y per il i-esimo valore di X diverso da 0, il prodotto parziale è negativo e quindi il bit più significativo dell'accumulatore A diventa 1.
- caso 3.** $X < 0$ e $Y > 0$: per l'ultimo prodotto va effettuato un ulteriore operazione di correzione A-M.
- caso 4.** $X < 0$ e $Y < 0$: anche in questo caso è necessario per l'ultimo prodotto l'operazione di correzione A-M. Il bit più significativo di A resta 0 e diventa 1 quando il i-esimo valore di X è 1.

L'algoritmo di Robertson è mostrato di seguito:

```
2CMultiplier:      (in:INBUS; OUT:OUTBUS)
                    register A[7:0],M[7:0],Q[7:0],COUNT[2:0],F;
                    bus INBUS[7:0],OUTBUS[7:0];

BEGIN:            A:=0,COUNT:=0,F:=0,
INPUT:            M:=INBUS;Q:=INBUS;

ADD:              A[7:0]:= A[7:0] + M [7:0] x Q[0],
                  F:= (M[7] and Q[0]) or F;
RSHIFT:           A[7]:= F,  A[6:0].Q:= A.Q[7:1];
INCREMENT:        COUNT:=COUNT+1

TEST:             if COUNT<7 then go to ADD;

SUBTRACT:         A[7:0]:=A[7:0]-M[7:0]xQ[0];    {l'ultima op è sempre SUB}
RSHIFT:           A[7]:= A[7],  A[6:0].Q:= A.Q[7:1];

OUTPUT:           OUTBUS:=Q; OUTBUS:=A;
END 2CMultiplier;
```

L'algoritmo inizia ponendo in M e Q i due operandi X e Y (inbus) e inizializzando il registro A (accumulatore), il contatore e il registro F. L'addizione si esegue tra A ed M e si pone il risultato nel registro A. Successivamente viene effettuata l'operazione $F = ((M(7) \text{ and } Q(0)) \text{ or } F)$, che serve a gestire il **caso 2** (moltiplicando negativo e moltiplicatore positivo). Il valore di F viene posto nel bit più significativo di A e il contenuto di quest'ultimo viene shiftato a destra in modo tale che il bit Q(1) si trovi in Q(0) per la prossima operazione. Il contatore delle operazioni viene incrementato e finché è minore di 7 le operazioni vanno avanti. Lo stato di correzione viene richiamato quando il contatore si ferma e viene fatta la correzione tramite sottrazione di A ed M. Dunque, a partire dall'algoritmo possiamo derivare lo schema logico del Moltiplicatore di Robertson in Figura 11.2.

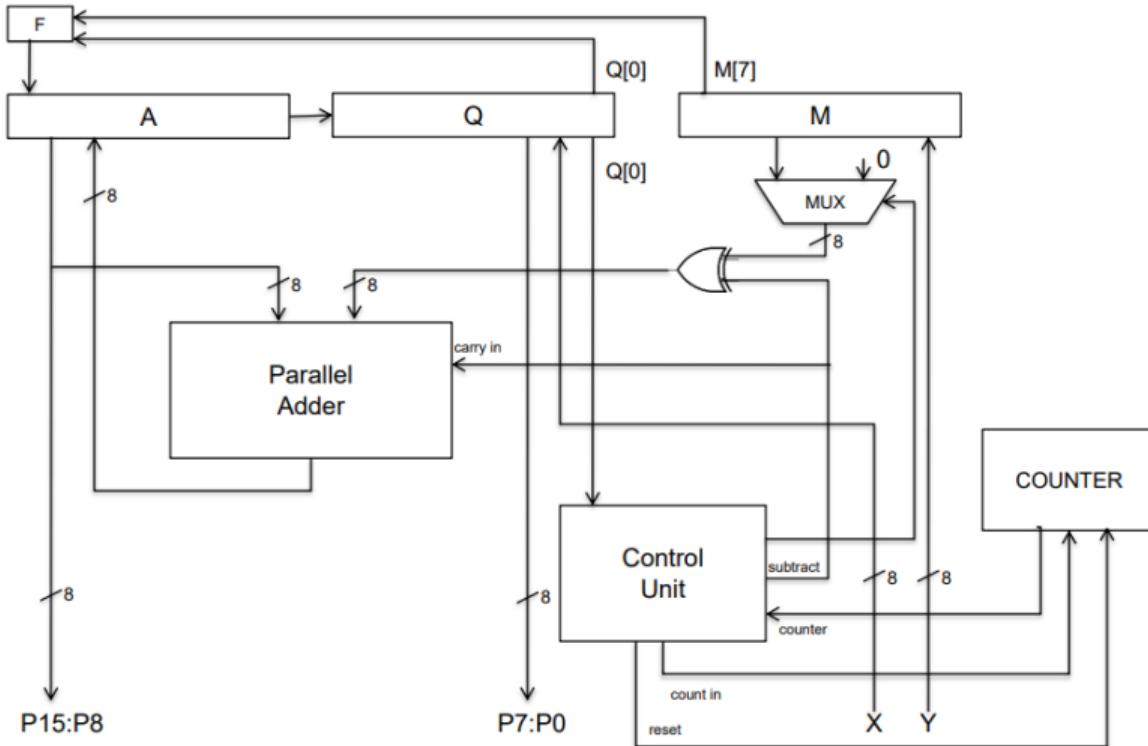


FIG. 11.2: Schema logico del Moltiplicatore di Robertson

Ritornando all'esempio, $7 \times 7 = 49$, applichiamo l'algoritmo di Robertson:

$$\begin{aligned} X = 7 &\rightarrow 0111 \text{ (moltiplicatore)} \\ Y = 7 &\rightarrow 0111 \text{ (moltiplicando } M\text{)} \\ P = X * Y &\rightarrow 00110001 \end{aligned}$$

FASE	F [(M(7) AND Q(0)) OR F]	A	Q	COUNT
INIT	0	0000	0111	0
M*Q(0)		0111		1
ADD	0	0111	0111	
RSHIFT	0	0011	1011	
M*Q(0)		0111		2
ADD	0	1010	1011	
RSHIFT	0	0101	0101	
M*Q(0)		0111		3
ADD	0	1100	0101	
RSHIFT	0	0110	0010	
M*Q(0)		0000		
SUB	0	0110	0010	
RSHIFT		0011	0001	

11.1.2 Schematico in RTL Analysis di Vivado

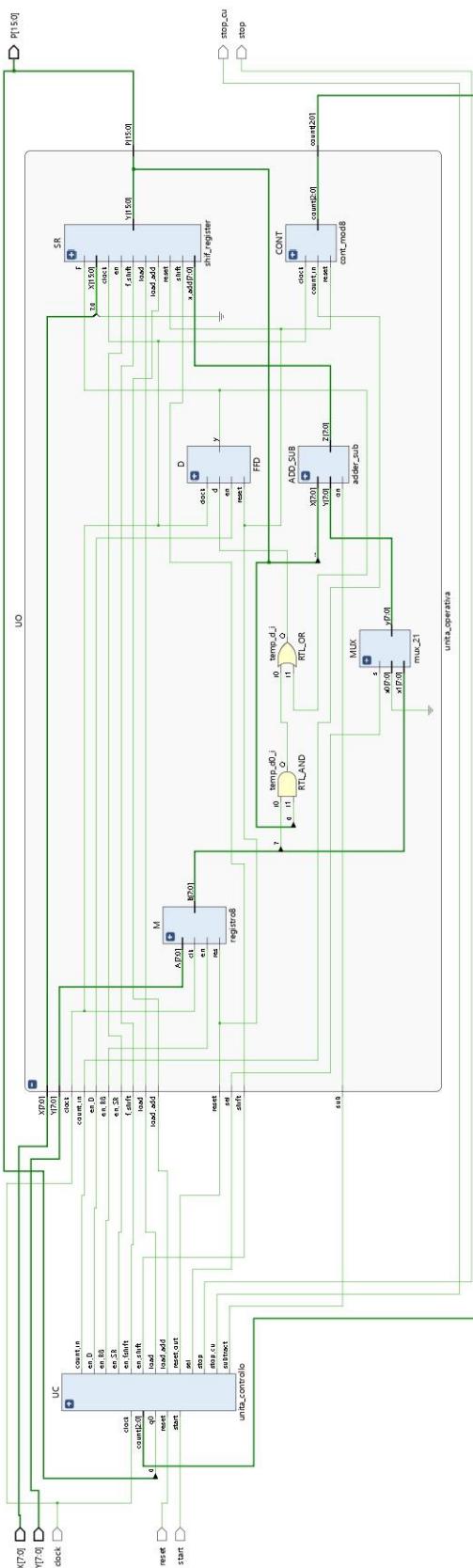


FIG 11.3

11.1.3 Codice VHDL del Moltiplicatore di Robertson

Robertson.vhd(top-module)

```

ENTITY Robertson IS
    PORT (
        clock, reset, start : IN STD_LOGIC;
        X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        stop : OUT STD_LOGIC;
        P : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        stop_cu : OUT STD_LOGIC);
END Robertson;

ARCHITECTURE structural OF Robertson IS
    COMPONENT unita_controllo IS
        PORT (
            q0, clock, reset, start : IN STD_LOGIC;
            count : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
            en_SR, en_R8, en_D, count_in, load_add, en_shift : OUT STD_LOGIC;
            sel, reset_out, subtract, load, stop, en_fshift, stop_cu
            : OUT STD_LOGIC);
        END COMPONENT;

    COMPONENT unita_operativa IS
        PORT (
            X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            clock, reset, load, sub, sel : IN STD_LOGIC;
            en_SR, en_R8, en_D, count_in, load_add, shift, f_shift :
            IN STD_LOGIC;
            count : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
            P : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
        END COMPONENT;

    SIGNAL tempq0, temp_sel, temp_clock, temp_res, temp_sub, temp_load :
    STD_LOGIC;
    SIGNAL temp_count : STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL temp_p : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL ab_sr, ab_r8, ab_d, ab_c, t_load_add : STD_LOGIC;
    SIGNAL fine conteggio : STD_LOGIC;
    SIGNAL temp_shift, temp_fshift : STD_LOGIC;

BEGIN

    UC : unita_controllo PORT MAP(tempq0, clock, reset, start,
    temp_count, ab_sr, ab_r8, ab_d, ab_c, t_load_add, temp_shift,
    temp_sel, temp_res, temp_sub, temp_load, stop, temp_fshift, stop_cu);
    UO : unita_operativa PORT MAP(X, Y, clock, temp_res, temp_load,
    temp_sub, temp_sel, ab_sr, ab_r8, ab_d, ab_c, t_load_add, temp_shift,
    temp_fshift, temp_count, temp_p);

    tempq0 <= temp_p(0);
    P <= temp_p;
END structural;

```

Per il progetto dell'unità di controllo del Moltiplicatore di Robertson, si è seguito l'automa in Figura 11.4, ovvero si è seguito un approccio in logica cablata. Sarebbe stato possibile effettuare anche un'UC in logica microprogrammata, tuttavia, dato che il moltiplicatore è una macchina sequenziale che esegue sempre le stesse istruzioni in sequenza tale approccio risulta poco vantaggioso rispetto a quello cablato.

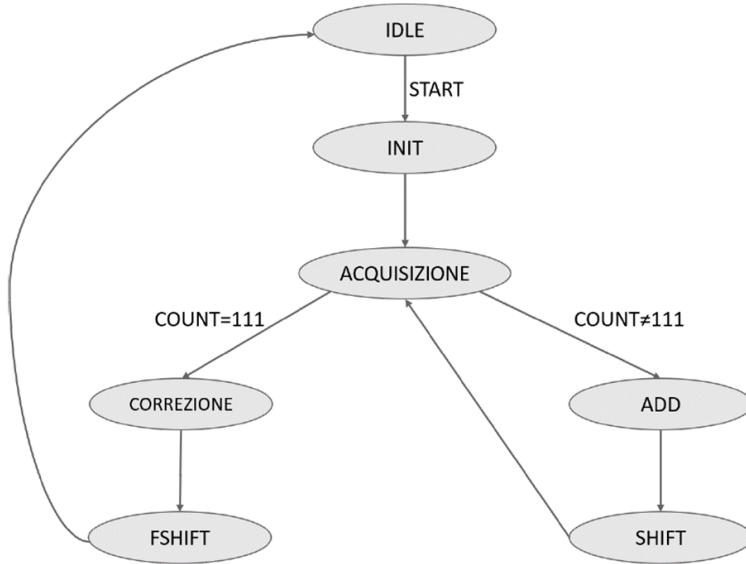


FIG. 11.4: Automa dell'unità di controllo del Moltiplicatore di Robertson

Il codice VHDL del componente è il seguente:

Unita_controllo.vhd

```

ARCHITECTURE behavioral OF unita_controllo IS
    TYPE state IS (idle, init, acquisizione, add, shift, correzione,
f_shift);
    SIGNAL current_state, next_state : state;

BEGIN
    reg_stato : PROCESS (clock, reset)
    BEGIN
        IF (reset = '1') THEN
            current_state <= init;
        ELSIF (clock'event AND clock = '1') THEN
            current_state <= next_state;
        END IF;
    END PROCESS;

    comb : PROCESS (current_state, count, start)
    BEGIN
        next_state <= idle;

        CASE current_state IS

            WHEN idle => reset_out <= '1';
                en_D <= '1'; --serve l'abilitazione alta altrimenti
non resetta

```

```

en_R8 <= '1';
en_SR <= '1';
count_in <= '0';
subtract <= '0';
load_add <= '0';
load <= '0';
sel <= '0';
stop <= '0';
stop_cu <= '0';
en_fshift <= '0';
en_shift <= '0';
IF (start = '1') THEN
    next_state <= init;
ELSE next_state <= idle;
END IF;
--in ogni stato dobbiamo predisporre i valori dei segnali per
lo stadio successivo
WHEN init => reset_out <= '0';
count_in <= '0';
load_add <= '0';
en_D <= '1';
en_R8 <= '1'; --carica y in acquisizione
en_SR <= '1';
load <= '1'; --abilitiamo SR per caricare x in
acquisizione
subtract <= '0';
stop <= '0';
sel <= q0;
en_shift <= '0';
en_fshift <= '0';
stop_cu <= '0';
next_state <= acquisizione;
--acquisisce gli operandi e predisponde la add
WHEN acquisizione => en_SR <= '1'; --disabilitiamo Sr
così non shifta
load <= '0';
reset_out <= '0';
en_D <= '1';
load_add <= '1';
en_R8 <= '1';
count_in <= '1';
subtract <= '0';
sel <= q0;
stop <= '0';
stop_cu <= '0';
en_fshift <= '0';
en_shift <= '0';
IF (count = "111") THEN
    subtract <= '1';
    next_state <= correzione;
    --predisponde la sottrazione
ELSE next_state <= add;
END IF;
--esegue la somma e predisponde lo shift
WHEN add => count_in <= '0';
reset_out <= '0';

```



```
sel <= q0;
subtract <= '0';
load_add <= '0'; --carico il valore nello shift
en_SR <= '1';
en_shift <= '1';
load <= '0';
en_D <= '0';
en_R8 <= '1';
stop_cu <= '0';
stop <= '0';
en_fshift <= '0';
next_state <= shift;
--esegue lo shift e predisponde per acquisizione
WHEN shift => load <= '0';
load_add <= '0';
en_SR <= '1'; --sr abilitato senza caricare x
reset_out <= '0';
sel <= q0;
en_shift <= '0';
count_in <= '0';
subtract <= '0';
en_D <= '0';
en_R8 <= '1';
stop_cu <= '0';
en_fshift <= '0';
stop <= '0';
next_state <= acquisizione;
--in correzione fa la sottrazione e predisponde per
l'ultimo shift
WHEN correzione => en_SR <= '1';
subtract <= '0';
load <= '0';
reset_out <= '0';
load_add <= '0';
en_D <= '0'; --a(7)=a(7)
en_R8 <= '1';
count_in <= '0';
en_shift <= '0';
en_fshift <= '1';
sel <= q0;
stop <= '1';
stop_cu <= '0';
next_state <= f_shift;
WHEN f_shift => subtract <= '0';
en_SR <= '0';
load <= '0';
reset_out <= '0';
load_add <= '0';
en_D <= '0';
en_R8 <= '1';
en_shift <= '0';
en_fshift <= '0';
count_in <= '0';
sel <= q0;
stop <= '0';
stop_cu <= '1';
```

```

        next_state <= idle;
    END CASE;
END PROCESS;

END behavioral;
```

Unita_operativa.vhd

```

-----  

ENTITY unita_operativa IS
    PORT (
        X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        clock, reset, load, sub, sel : IN STD_LOGIC;
        en_SR, en_R8, en_D, count_in, load_add, shift, f_shift : IN STD_LOGIC;
        count : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        P : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END unita_operativa;  

ARCHITECTURE structural OF unita_operativa IS
  

    COMPONENT adder_sub IS
        PORT (
            X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            cin : IN STD_LOGIC;
            Z : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            cout : OUT STD_LOGIC);
END COMPONENT;
  

    COMPONENT registro8 IS
        PORT (
            A : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            clk, res, en : IN STD_LOGIC;
            B : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;
  

    COMPONENT mux_21 IS
        PORT (
            x0, x1 : IN STD_LOGIC_VECTOR(7 DOWNTO 0); --x1 è il
multicicatore e x0 è                                la stringa di 0
            s : IN STD_LOGIC;
            y : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;
  

    COMPONENT shif_register IS
        PORT (
            X : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
            F : IN STD_LOGIC;
            clock, reset, load, en, load_add, shift, f_shift : IN STD_LOGIC;
            x_add : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            Y : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END COMPONENT;
    COMPONENT FFD IS
        PORT (
            clock, reset, d, en : IN STD_LOGIC;
```



```

        y : OUT STD_LOGIC);
END COMPONENT;

COMPONENT cont_mod8 IS
  PORT (
    clock, reset : IN STD_LOGIC;
    count_in : IN STD_LOGIC;
    count_end : OUT STD_LOGIC;
    count : OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END COMPONENT;

SIGNAL temp1 : STD_LOGIC_VECTOR(7 DOWNTO 0); --segnale temporaneo tra reg8 M e
                                                 mux 21
SIGNAL op1 : STD_LOGIC_VECTOR(7 DOWNTO 0); --segnale temporaneo di uscita dal
                                              multiplexer tra mux e
                                              adder
SIGNAL temp2 : STD_LOGIC_VECTOR(15 DOWNTO 0); --segnale temporaneo per definire
                                                 input all'SR
SIGNAL temp_p : STD_LOGIC_VECTOR(15 DOWNTO 0); --segnale temporaneo uscita dell'SR
SIGNAL temp_d : STD_LOGIC := '0';
SIGNAL temp_F : STD_LOGIC;
SIGNAL tempadd : STD_LOGIC_VECTOR(7 DOWNTO 0); --uscita del parallel adder
SIGNAL riporto : STD_LOGIC; -- riporto in uscita dell'adder che non utilizziamo
SIGNAL temp_countend : STD_LOGIC;

BEGIN

  M : registro8 PORT MAP(Y, clock, reset, en_R8, temp1);
  MUX : mux_21 PORT MAP("00000000", temp1, sel, op1);

  temp2 <= "00000000" & X;

  SR : shif_register PORT MAP(temp2, temp_F, clock, reset, load, en_SR,
    load_add, shift, f_shift, tempadd, temp_p);

  temp_d <= (temp1(7) AND temp_p(0)) OR temp_F;

  D : FFD PORT MAP(clock, reset, temp_d, en_D, temp_F);
  ADD_SUB : adder_sub PORT MAP(temp_p(15 DOWNTO 8), op1, sub, tempadd,
    riporto);
  CONT : cont_mod8 PORT MAP(clock, reset, count_in, temp_countend,
    count);

  P <= temp_p;

END structural;

```

11.1.4 La simulazione del Moltiplicatore di Robertson***TB_robertson.vhd***

```
ENTITY tb_Robertson IS
    -- Port();
END tb_Robertson;

ARCHITECTURE tb OF tb_Robertson IS

COMPONENT Robertson
PORT (
    clock : IN STD_LOGIC;
    reset : IN STD_LOGIC;
    start : IN STD_LOGIC;
    X : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    Y : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    stop : OUT STD_LOGIC;
    P : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    stop_cu : OUT STD_LOGIC);
END COMPONENT;

SIGNAL clock : STD_LOGIC;
SIGNAL reset : STD_LOGIC;
SIGNAL start : STD_LOGIC;
SIGNAL X : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Y : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL stop : STD_LOGIC;
SIGNAL P : STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL stop_cu : STD_LOGIC;

CONSTANT TbPeriod : TIME := 10 ns; -- EDIT Put right period here
SIGNAL TbClock : STD_LOGIC := '0';
SIGNAL TbSimEnded : STD_LOGIC := '0';

BEGIN

dut : Robertson
PORT MAP(
    clock => clock,
    reset => reset,
    start => start,
    X => X,
    Y => Y,
    stop => stop,
    P => P,
    stop_cu => stop_cu);

-- Clock generation
TbClock <= NOT TbClock AFTER TbPeriod/2 WHEN TbSimEnded /= '1' ELSE
'0';

-- EDIT: Check that clock is really your main clock signal
clock <= TbClock;
stimuli : PROCESS
```



```
BEGIN
    -- EDIT Adapt initialization as needed
    -- Inizializzazione, si mette tutto a 0
    start <= '0';
    Y <= "00000000";
    X <= "00000000";

    -- Reset generation
    -- EDIT: Check that reset is really your reset signal
    reset <= '1';
    WAIT FOR 10 ns;
    reset <= '0';
    WAIT FOR 10 ns;

    -- Si inizializza l'automa
    WAIT FOR 10 ns;
    start <= '1';

    WAIT FOR 100 ns;
    start <= '0';

    ---- Attesa

    WAIT FOR 200 ns;
    -----
    -- Si fa 7 x 7 = 49
    start <= '0';
    Y <= "00000111";
    X <= "00000111";

    -- Si resetta per nuovi operandi
    reset <= '1';
    WAIT FOR 10 ns;
    reset <= '0';
    WAIT FOR 10 ns;

    -- Si restarta l'automa
    WAIT FOR 10 ns;
    start <= '1';
    WAIT FOR 100 ns;
    -- Si spegne l'automa
    start <= '0';

    ---- Attesa

    WAIT FOR 200 ns;
    -----
    -- Si fa un X < 0 e un Y > 0
    start <= '0';
    Y <= "00001000";
    X <= "11111110";

    -- Si resetta per nuovi operandi
```

```
reset <= '1';
WAIT FOR 10 ns;
reset <= '0';
WAIT FOR 10 ns;

-- Si restarta l'automa
WAIT FOR 10 ns;
start <= '1';

WAIT FOR 100 ns;
-- Si spegne l'automa
start <= '0';

----- Attesa

WAIT FOR 200 ns;
-----

-- Si fa un X < 0 e un Y < 0
start <= '0';
Y <= "11111101";
X <= "11111100";

-- Si resetta per nuovi operandi
reset <= '1';
WAIT FOR 10 ns;
reset <= '0';
WAIT FOR 10 ns;

-- Si restarta l'automa
WAIT FOR 10 ns;
start <= '1';

WAIT FOR 100 ns;
-- Si spegne l'automa
start <= '0';

WAIT FOR 1000 * TbPeriod;

-- Stop the clock and hence terminate the simulation
TbSimEnded <= '1';

WAIT;
END PROCESS;

END tb;
```

Come è possibile notare dalla Figura 11.5, il risultato dell'operazione $7 (00000111) \times 7 (00000111)$ viene calcolato dal Moltiplicatore con una sequenza di shift e somme, dunque, l'uscita P assumerà durante quest'operazione diversi valori prima di mostrare il risultato corretto. Quindi, al fine di verificare che la moltiplicazione sia andata a buon fine occorre vedere il valore di P solo all'istante di tempo in cui il segnale *stop_cu* si alza, infatti, come si può notare dalla simulazione stessa solo allora P assume il valore corretto, ovvero 49 (00110001). Accade la stessa cosa anche nelle simulazioni successive.

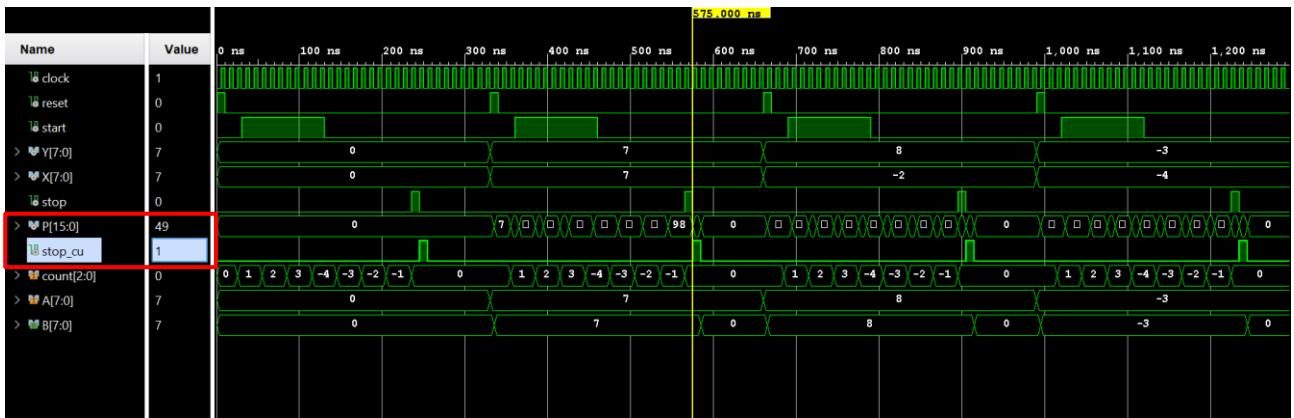


FIG. 11.5: Moltiplicazione 7x7

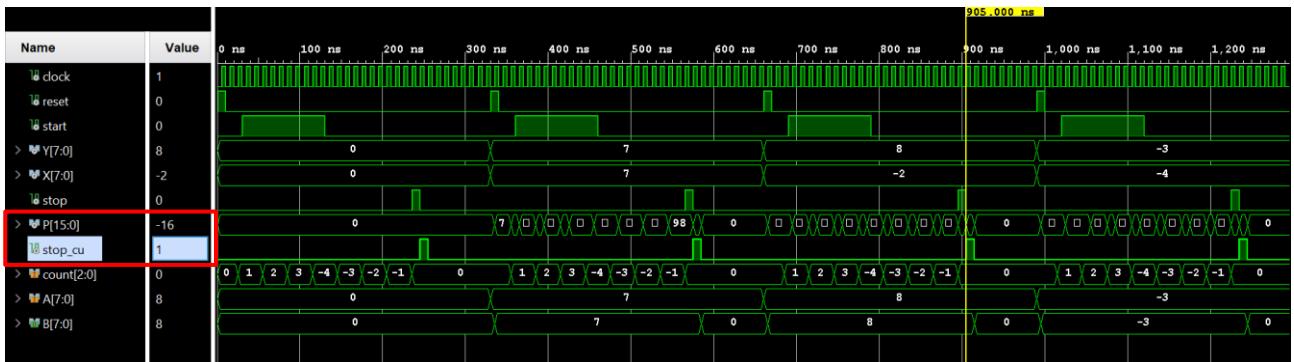


FIG. 11.6: Moltiplicazione 8x(-2)

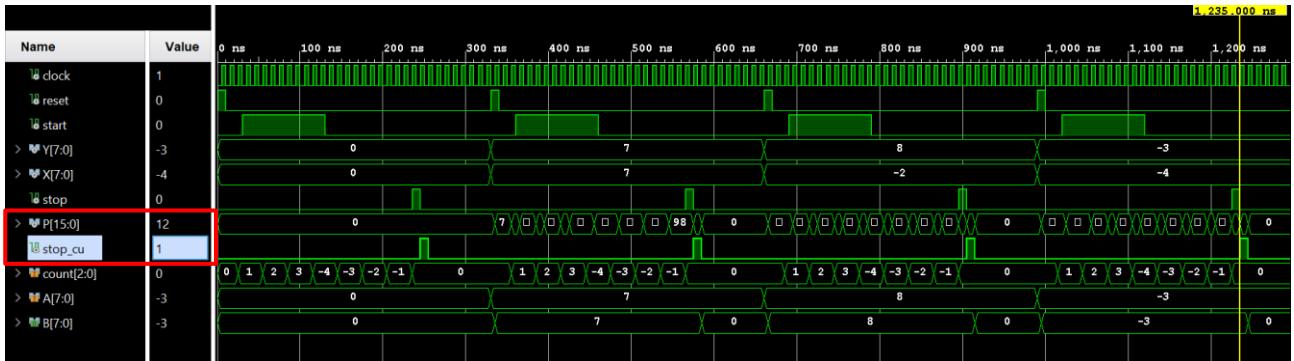


FIG. 11.7: Moltiplicazione -3x(-4)

11.1.5 Implementazione sulla board tramite switch e display – versione esadecimale

RobertsonOnBoard.vhd (top-module)

```
ENTITY RobertsonOnBoard IS
  PORT (
    clock, reset, start : IN STD_LOGIC;
    X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    anodes_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    cathodes_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END RobertsonOnBoard;
```

ARCHITECTURE Structural OF RobertsonOnBoard IS



```

COMPONENT Robertson IS
  PORT (
    clock, reset, start : IN STD_LOGIC;
    X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    stop : OUT STD_LOGIC;
    P : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    stop_cu : OUT STD_LOGIC
  );
END COMPONENT;

COMPONENT display_seven_segments IS
  GENERIC (
    CLKIN_freq : INTEGER := 100000000;
    CLKOUT_freq : INTEGER := 500
  );
  PORT (
    CLK : IN STD_LOGIC;
    RST : IN STD_LOGIC;
    VALUE : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    ENABLE : IN STD_LOGIC_VECTOR (7 DOWNTO 0); -- decide quali
cifre abilitare
    DOTS : IN STD_LOGIC_VECTOR (7 DOWNTO 0); -- decide quali
punti
    ANODES : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    CATHODES : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
  );
END COMPONENT;
COMPONENT ButtonDebouncer IS
  GENERIC (
    CLK_period : INTEGER := 10; -- periodo del clock della
board 10 nanosecondi
    btn_noise_time : INTEGER := 6500000 --intervallo di tempo
in cui si ha
    l'oscillazione      del
    bottone
  );
  PORT (
    RST : IN STD_LOGIC;
    CLK : IN STD_LOGIC;
    BTN : IN STD_LOGIC;
    CLEARED_BTN : OUT STD_LOGIC);
END COMPONENT;

SIGNAL stop_cu_temp : STD_LOGIC;
SIGNAL out_temp : STD_LOGIC_VECTOR(31 DOWNTO 0) := (others => '0');
SIGNAL out_temp_display : STD_LOGIC_VECTOR(31 DOWNTO 0) := (others
=> '0');
SIGNAL cleaned_start : STD_LOGIC;
SIGNAL cleaned_reset : STD_LOGIC;

BEGIN
  out_temp_display <= (others => '0'); WHEN cleaned_reset = '1' ELSE
    out_temp WHEN stop_cu_temp = '1' ELSE
      out_temp_display;

```

```

inst_Robertson : Robertson
PORT MAP(
    clock => clock,
    reset => cleaned_reset,
    start => cleaned_start,
    X => X,
    Y => Y,
    stop => OPEN,
    P => out_temp(15 DOWNTO 0),
    stop_cu => stop_cu_temp
);

inst_Display : display_seven_segments
PORT MAP(
    CLK => clock,
    RST => cleaned_reset,
    VALUE => out_temp_display,
    ENABLE => "11111111", -- decide quali cifre abilitare
    DOTS => "00000000", -- decide quali punti visualizzare
    ANODES => anodes_out,
    CATHODES => cathodes_out
);

start_debouncer : ButtonDebouncer
PORT MAP(
    RST => cleaned_reset,
    CLK => clock,
    BTN => start,
    CLEARED_BTN => cleaned_start
);
reset_debouncer : ButtonDebouncer
PORT MAP(
    RST => '0',
    CLK => clock,
    BTN => reset,
    CLEARED_BTN => cleaned_reset
);

END Structural;

```

Per abilitare il display su cui vedere il risultato, i bottoni da premere per effettuare l'operazione ed infine, gli switch da cui prendere i due operandi X e Y si imposta il constraint file nel seguente modo:

```

##Clock
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports {
clock }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5}
[get_ports {clock}];100mhz

##Switches
set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports {
X[0] }];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports {
X[1] }];
#IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]w[0]

```

```

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

```

##7 segment display

```

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

```



```

set_property -dict { PACKAGE_PIN T14      IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[5] }];
#IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2       IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[6] }];
#IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13      IOSTANDARD LVCMOS33 } [get_ports {
anodes_out[7] }];
#IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons
set_property -dict { PACKAGE_PIN P17      IOSTANDARD LVCMOS33 } [get_ports {
start }];
#IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17      IOSTANDARD LVCMOS33 } [get_ports {
reset }];
#IO_L10N_T1_D15_14 Sch=bt(nr

```

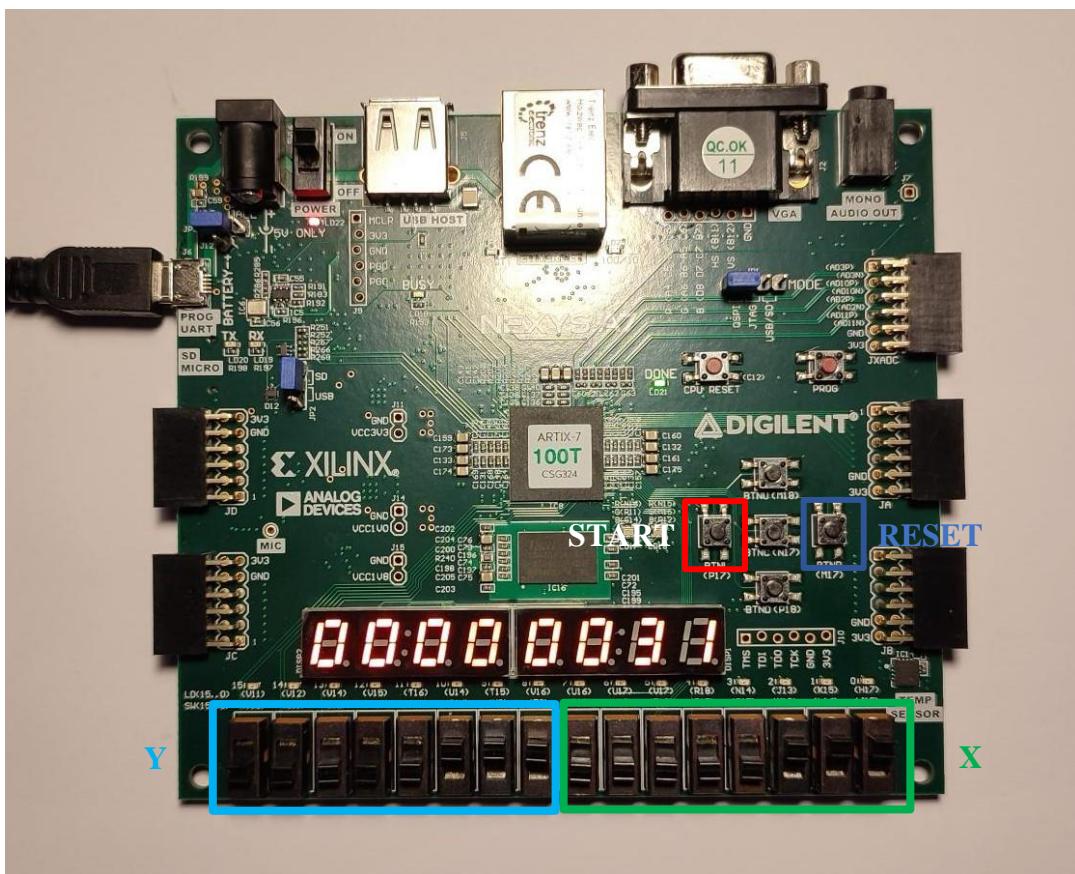


FIG 11.8: Moltiplicazione $7 \times 7 = 49$, in esadecimale 0x31

11.1.6 Implementazione sulla board tramite switch e display – versione decimale

Al fine di mostrare il risultato della moltiplicazione in formato decimale, abbiamo sviluppato un ulteriore componente “*converter_bin_to_dec*”, che oltre a convertire il risultato da binario a decimale, si occupa di gestire anche i risultati negativi.

Convert_bin_to_dec.vhd

```

ENTITY converter_bin_to_dec IS
  PORT (
    data_bin : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    data_dec : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
  );

```

```
END converter_bin_to_dec;

ARCHITECTURE Behavioral OF converter_bin_to_dec IS

    SIGNAL Cif1 : STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL Cif2 : STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL Cif3 : STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL Cif4 : STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL Cif5 : STD_LOGIC_VECTOR(3 DOWNTO 0);

    SIGNAL Cif1_temp : INTEGER;
    SIGNAL Cif2_temp : INTEGER;
    SIGNAL Cif3_temp : INTEGER;
    SIGNAL Cif4_temp : INTEGER;
    SIGNAL Cif5_temp : INTEGER;

    SIGNAL negative : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";

BEGIN

    conv : PROCESS (data_bin)
    BEGIN
        Cif1_temp <= to_integer(ABS(signed(data_bin))) MOD 10;
        Cif2_temp <= (to_integer(ABS(signed(data_bin))) MOD 100)/10;
        Cif3_temp <= (to_integer(ABS(signed(data_bin))) MOD 1000)/100;
        Cif4_temp <= (to_integer(ABS(signed(data_bin))) MOD 10000)/1000;
        Cif5_temp <= (to_integer(ABS(signed(data_bin))) MOD 100000)/10000;

        Cif1 <= STD_LOGIC_VECTOR(to_signed(Cif1_temp, 4));
        Cif2 <= STD_LOGIC_VECTOR(to_signed(Cif2_temp, 4));
        Cif3 <= STD_LOGIC_VECTOR(to_signed(Cif3_temp, 4));
        Cif4 <= STD_LOGIC_VECTOR(to_signed(Cif4_temp, 4));
        Cif5 <= STD_LOGIC_VECTOR(to_signed(Cif5_temp, 4));

        IF (data_bin(15) = '1') THEN
            negative <= "1010";
        ELSE
            negative <= "0000";
        END IF;

        data_dec <= negative & "0000" & "0000" & Cif5 & Cif4 & Cif3 &
Cif2 & Cif1;

    END PROCESS;

END Behavioral;
```

Tuttavia, solo tale componente non è sufficiente a mostrare sul display i numeri negativi, infatti, occorre modificare anche la logica di controllo dei catodi presente nel componente *"cathodes_manager"*.

Cathodes_manager.vhd

```

ENTITY cathodes_manager IS
  PORT (
    counter : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
    value : IN STD_LOGIC_VECTOR (31 DOWNTO 0); --dato di mostrare
sugli 8 display
    dots : IN STD_LOGIC_VECTOR (7 DOWNTO 0); --configurazione punti
da accendere
    cathodes : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)); --sono i 7 catodi
più il punto
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 trattino : STD_LOGIC_VECTOR(6 DOWNTO 0) := "0111111";

  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);
  ALIAS digit_4 IS value (19 DOWNTO 16);
  ALIAS digit_5 IS value (23 DOWNTO 20);
  ALIAS digit_6 IS value (27 DOWNTO 24);
  ALIAS digit_7 IS value (31 DOWNTO 28);

  SIGNAL cathodes_for_digit : STD_LOGIC_VECTOR(6 DOWNTO 0) := (OTHERS
=> '0');
  SIGNAL nibble : STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
  SIGNAL dot : STD_LOGIC := '0'; --stabilisce se il punto relativo alla
cifra visualizzata deve
                                essere acceso o spento
--nota: dot=1 significa che deve essere acceso, ma il segnale deve essere
negato per andare sui catodi

BEGIN

  -- questo processo multiplexa le cifre da mostrare
  digit_switching : PROCESS (counter)

```

```
BEGIN
  CASE counter IS
    WHEN "000" =>
      nibble <= digit_0;
      dot <= dots(0);
    WHEN "001" =>
      nibble <= digit_1;
      dot <= dots(1);
    WHEN "010" =>
      nibble <= digit_2;
      dot <= dots(2);
    WHEN "011" =>
      nibble <= digit_3;
      dot <= dots(3);
    WHEN "100" =>
      nibble <= digit_4;
      dot <= dots(4);
    WHEN "101" =>
      nibble <= digit_5;
      dot <= dots(5);
    WHEN "110" =>
      nibble <= digit_6;
      dot <= dots(6);
    WHEN "111" =>
      nibble <= digit_7;
      dot <= dots(7);
    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 "1010" => cathodes_for_digit <= trattino;
    WHEN OTHERS => cathodes_for_digit <= (OTHERS => '0');
  END CASE;
END PROCESS seven_segment_decoder_process;
```

```
cathodes <= (NOT dot) & cathodes_for_digit;
END Behavioral;
```

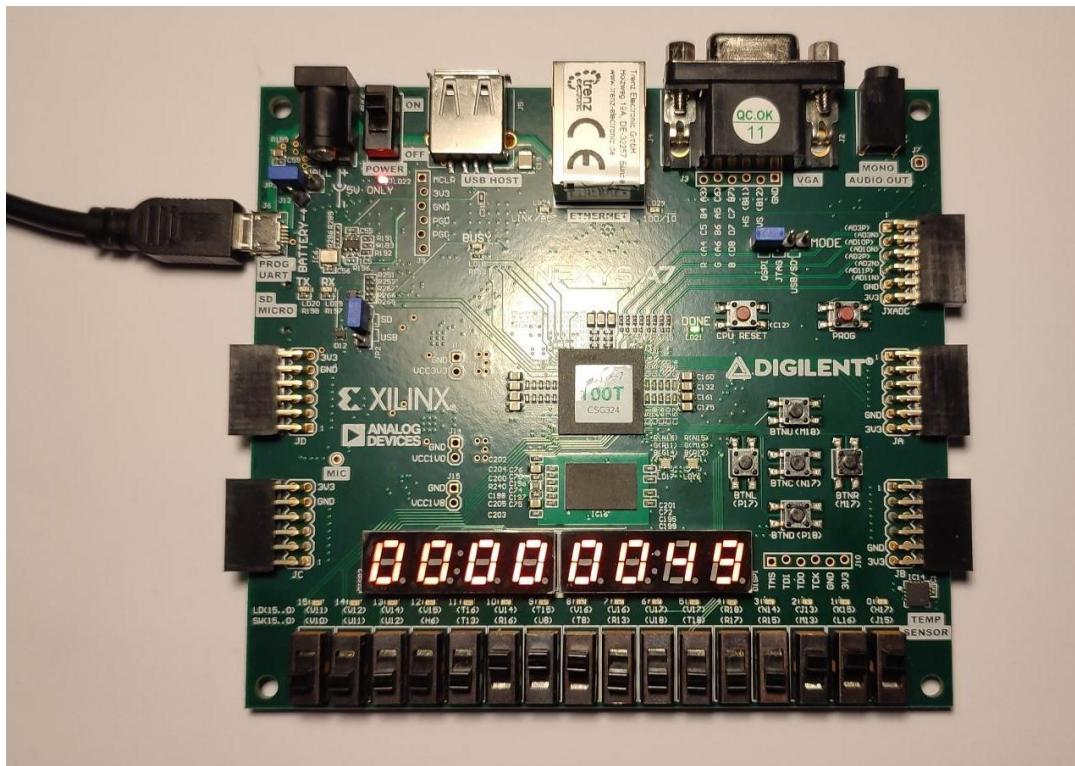


FIG. 11.9: Moltiplicazione $7 \times 7 = 49$ in decimale

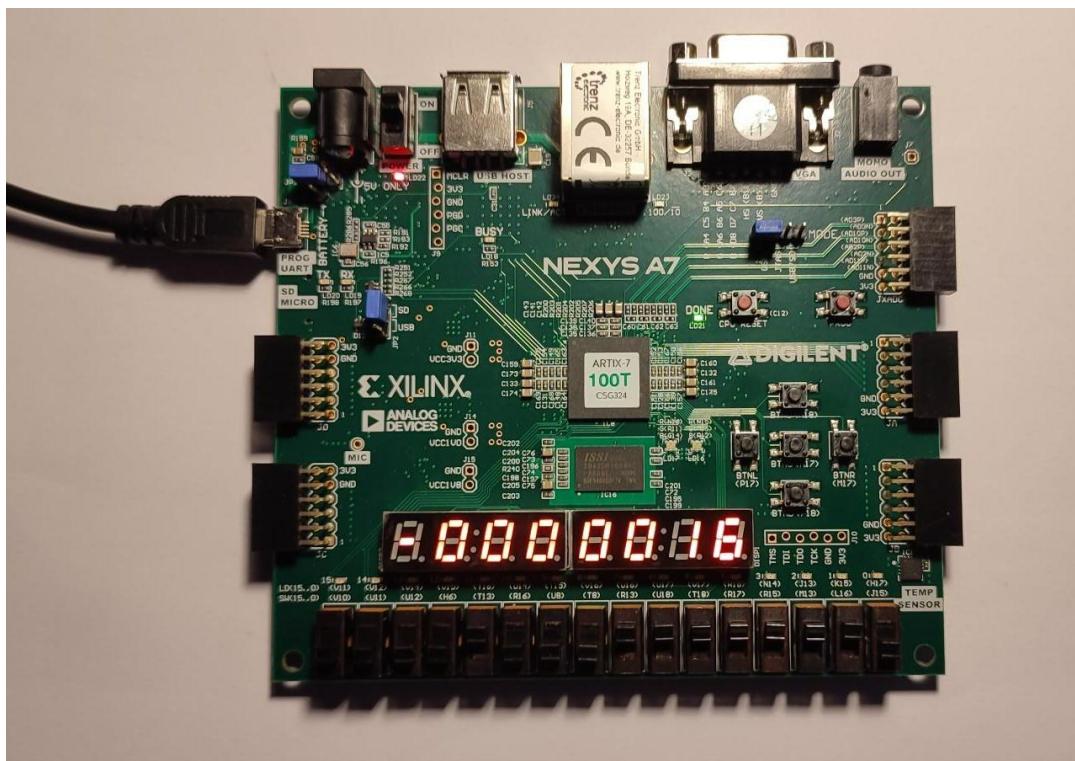


FIG. 11.10: Moltiplicazione $8 \times (-2) = -16$ in decimale

11.2 Applicazioni reali e timing analysis del Moltiplicatore di Robertson

Una possibile applicazione del moltiplicatore di Robertson è quella di calcolo degli ingressi di una funzione di attivazione di 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.

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 11.9. Ciascun neurone riceve un certo numero di ingressi ed esegue la somma riportata nella seguente equazione:

$$h = \sum_{i=0}^{n-1} x_i w_i + b$$

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.

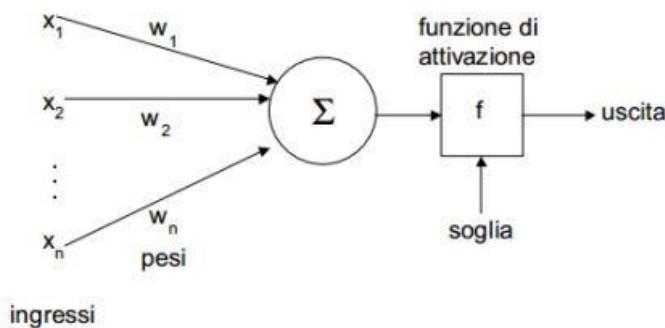


FIG 11.11: Struttura di un neurone artificiale

Tuttavia, una rete neurale non è composta da un singolo neurone, ma da molteplici, disposti su diversi livelli che portano i dati dall'ingresso all'uscita eseguendo numerose volte somme e prodotti per calcolare le varie funzioni di attivazione. Uno schema completo di rete neurale è quello della Figura 11.10, in cui ogni nodo è un neurone del tipo visto in precedenza.

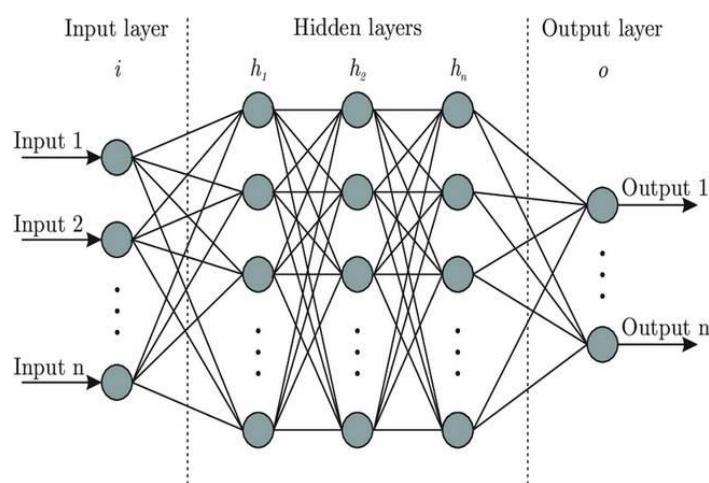


FIG 11.12: Rete neurale

MOLTIPLICATORE DI ROBERTSON

Quindi, affinché una rete neurale lavori efficientemente, occorre che i componenti che effettuano la moltiplicazione e la somma siano a bassa latenza; quindi, si effettua la timing analysis del moltiplicatore di Robertson nelle due versioni prima mostrate sulla board.

WNS(ns)	TNS(ns)	TNS Failing Endpoints	TNS Total Endpoints	WHS(ns)	THS(ns)	THS Failing Endpoints
4.881	0.000	0	314	0.184	0.000	0

All user specified timing constraints are met.

Max Delay Paths

```

Slack (MET) : 4.881ns (required time - arrival time)
Source: inst_Robertson/UC/FSM_onehot_current_state_reg[1]/C
          (rising edge-triggered cell FDCE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns}
Destination: inst_Robertson/UO/SR/temp_reg[14]/D
          (rising edge-triggered cell FDCE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns}

Path Group: sys_clk_pin
Path Type: Setup (Max at Slow Process Corner)
Requirement: 10.000ns (sys_clk_pin rise@10.000ns - sys_clk_pin rise@0.000ns)
Data Path Delay: 5.090ns (logic 1.492ns (29.312%) route 3.598ns (70.688%))
Logic Levels: 6 (LUT2=1 LUT5=2 LUT6=3)
Clock Path Skew: -0.025ns (DCD - SCD + CPR)
  Destination Clock Delay (DCD): 5.017ns = ( 15.017 - 10.000 )
  Source Clock Delay (SCD): 5.314ns
  Clock Pessimism Removal (CPR): 0.272ns
Clock Uncertainty: 0.035ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
  Total System Jitter (TSJ): 0.071ns
  Total Input Jitter (TIJ): 0.000ns
  Discrete Jitter (DJ): 0.000ns
  Phase Error (PE): 0.000ns

```

FIG. 11.13: Timing analysis del Moltiplicatore di Robertson - versione esadecimale

WNS(ns)	TNS(ns)	TNS Failing Endpoints	TNS Total Endpoints	WHS(ns)	THS(ns)	THS Failing Endpoints
3.621	0.000	0	314	0.229	0.000	0

All user specified timing constraints are met.

Max Delay Paths

```

Slack (MET) : 4.881ns (required time - arrival time)
Source: inst_Robertson/UC/FSM_onehot_current_state_reg[1]/C
          (rising edge-triggered cell FDCE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns}
Destination: inst_Robertson/UO/SR/temp_reg[14]/D
          (rising edge-triggered cell FDCE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns}

Path Group: sys_clk_pin
Path Type: Setup (Max at Slow Process Corner)
Requirement: 10.000ns (sys_clk_pin rise@10.000ns - sys_clk_pin rise@0.000ns)
Data Path Delay: 5.090ns (logic 1.492ns (29.312%) route 3.598ns (70.688%))
Logic Levels: 6 (LUT2=1 LUT5=2 LUT6=3)
Clock Path Skew: -0.025ns (DCD - SCD + CPR)
  Destination Clock Delay (DCD): 5.017ns = ( 15.017 - 10.000 )
  Source Clock Delay (SCD): 5.314ns
  Clock Pessimism Removal (CPR): 0.272ns
Clock Uncertainty: 0.035ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
  Total System Jitter (TSJ): 0.071ns
  Total Input Jitter (TIJ): 0.000ns
  Discrete Jitter (DJ): 0.000ns
  Phase Error (PE): 0.000ns

```

FIG. 11.14: Timing analysis del Moltiplicatore di Robertson - versione decimale



12 Divisore Non-Restoring

Come esercizio libero del progetto, si è scelto di progettare ed implementare in VHDL, un'altra macchina aritmetica sequenziale, in particolare, si è realizzato il Divisore Non-Restoring secondo le caratteristiche di funzionamento viste anche a lezione. Inoltre, si sono fatte delle osservazioni in merito alla sua tempificazione ed al suo utilizzo in un ambito più realistico.

12.1 Progettazione del Divisore Non-Restoring

Il Divisore Non-Restoring è una macchina aritmetica, il cui compito è quello di determinare, a partire da un D dividendo $D_{m-1} \dots D_0$ (su m bit) e da un $V \neq 0$ divisore $V_{n-1} \dots V_0$ (su n bit) un quoziente Q e un eventuale resto R in modo che sia verificata la seguente condizione:

$$D = QxV + R \text{ con } 0 \leq R < V$$

Se V è espresso su *n bit*, allora poiché il resto può assumere al massimo il valore $R=V-1$, R è anche esso espresso su al più *n bit*. Il massimo quoziente si ha col minimo divisore, ossia con $V=1$; in questo caso Q coincide con D e quindi se D è espresso su *m bit* anche Q sarà espresso su al più *m bit*, in generale Q è espresso su al più $m-n+1$ bit.

Come visto per il Moltiplicatore di Robertson nel Capitolo 11, prima di mostrare il Divisore Non-Restoring, si analizza il procedimento manuale di divisione; in particolare si ha che il dividendo viene scandito da sinistra verso destra, e i bit del quoziente vengono determinati uno alla volta a partire da quello più significativo, procedendo con una serie di confronti e sottrazioni.

- **PROCEDIMENTO MANUALE DI DIVISIONE**

Passo iniziale:

- 1) Si confrontano gli *n bit* più significativi del dividendo (dividendo parziale D_0) con gli *n* bit del divisore;
- 2) Si calcola la prima cifra (da sinistra) del quoziente q_0 , che sarà 1 oppure 0 a seconda che il dividendo parziale D_0 contenga o no il divisore;
- 3) Si effettua la sottrazione fra il dividendo parziale D_0 e il prodotto $q_0 V$, determinando il primo resto parziale R_1

Generico passo i:

- 1) Si pone $D_i = R_{i-1}$, con R_i resto parziale determinato al passo $i-1$; si confronta D_i con il divisore;
- 2) Si calcola la *i*-esima cifra (da sinistra) del quoziente q_i , che sarà 1 oppure 0 a seconda che il dividendo parziale D_i contenga o meno il divisore;
- 3) Si effettua la sottrazione fra il dividendo parziale D_i e il prodotto $q_i V$ shiftato a destra di *i* posizioni, determinando il nuovo resto parziale R_{i+1}

La procedura termina quando il dividendo è stato scandito completamente: il resto parziale R_i determinato in questo passo costituisce il resto finale R della divisione.

Come esempio di applicazione della procedura manuale, si fa riferimento all'esempio in Figura 12.1, con $D = 29$ (011101) e $V = 4$ (100), il cui risultato è $Q = 7$ (0111) e $R = 1$ (001).



dividendo	divisore
0 1 1 1 0 1	1 0 0
0 0 0	0 1 1 1
1 1 1	
1 0 0	
0 1 1 0	
1 0 0	
0 1 0 1	
1 0 0	
0 0 1	
	resto

FIG 12.1: Esempio di divisione con procedura manuale

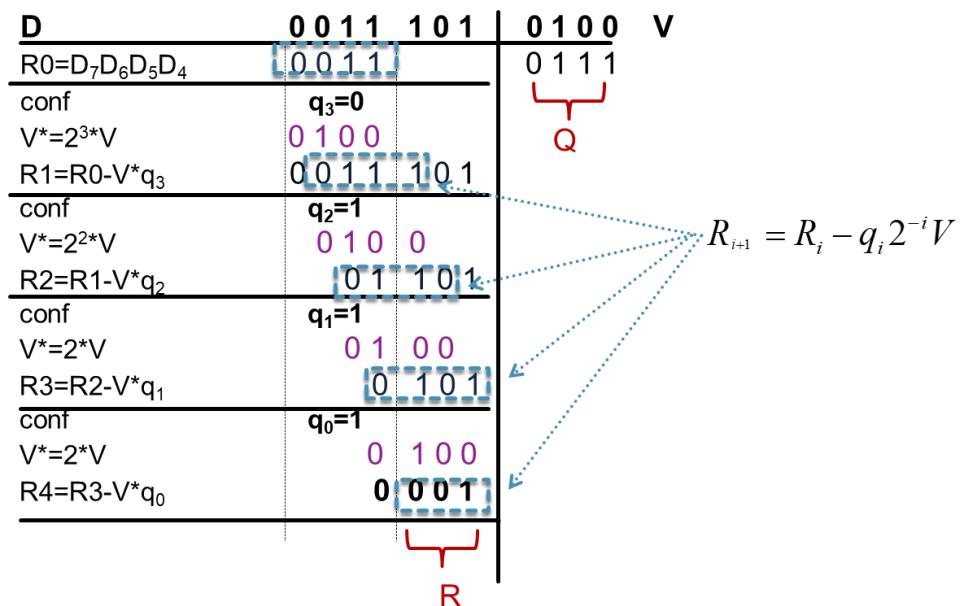
Si sarà notato che all'inizio dell'algoritmo manuale è necessario allineare il divisore al dividendo parziale shiftandolo a sinistra di $m-n$ posizioni. Al passo i -esimo (con $i=0 \dots m-n$) vengono effettuate le seguenti operazioni:

- 1) Si confronta $R_i = D_i$ con V : se D_i contiene V , la cifra i -esima (da sinistra) del quoziente q_i sarà 1, altrimenti sarà 0;
- 2) Si calcola il nuovo dividendo parziale.

Queste operazioni possono essere riassunte nella seguente formula:

$$R_{i+1} = R_i - q_i 2^{-i} V$$

Ad ogni passo è necessario effettuare un'operazione di shift del divisore di i posizioni a destra e una sottrazione tra il dividendo parziale e $q_i V$. Riprendendo l'esempio precedente, nella Figura 12.2, si possono apprezzare le operazioni di shift e sottrazione effettuate ad ogni passo della procedura.

**FIG 12.2:** Operazioni di shift e sottrazione della procedura manuale di divisione

- ALGORITMO ALTERNATIVO DI DIVISIONE

Al passo i -esimo, invece, della sequenza di operazioni:

1. confronto di R_i con V per determinare q_i
2. prodotto di V per q_i e shift di i posizioni a destra
3. sottrazione del prodotto shiftato da R_i per calcolare R_{i+1}

Si può considerare una versione alternativa dell'algoritmo che considera la nuova sequenza di operazioni:

1. left shift di una posizione di R_i
2. confronto di $2R_i$ con V per determinare q_i sottrazione del prodotto q_iV da $2R_i$

$$R_i := 2R_i; R_{i+1} := R_i - q_iV;$$

$$R_{i+1} := 2R_i - q_iV;$$

L'algoritmo alternativo è del tutto equivalente a quello derivato dalla procedura manuale, ma ha il vantaggio che ad ogni passo si effettua sempre uno shift di una sola posizione a sinistra, questo rende più facile la sua conversione in un algoritmo da far eseguire ad una macchina. In Figura 12.3, con il solito esempio di 29 diviso 4, si illustrano le operazioni effettuate per calcolare la divisione nell'algoritmo alternativo.

D=R0	0 0 1 1 1 0 1	0 1 0 0	V
2R0	0 0 1 1 1 0 1 -	0 1 1 1	
conf	0 0 0 0		
2R0-V*=R1	0 0 1 1 1 0 1 -		
2R1	0 0 1 1 1 0 1 - -		
conf	0 1 0 0		
2R1-V*=R2	0 0 1 1 0 1 - -		
2R2	0 1 1 0 1 - - -		
conf	0 1 0 0		
2R1-V*=R3	0 0 1 0 1 - - -		
2R3	0 1 0 1 - - - -		
conf	0 1 0 0		
2R1-V*=R4=R	0 0 0 1 - - - -		
		A=R	Q

FIG. 12.3: Algoritmo alternativo per il calcolo della divisione

Guardando l'esempio si possono fare diverse considerazioni, innanzitutto, ad ogni passo il dividendo parziale R_i viene shiftato a sinistra, e i suoi primi n bit vengono «confrontati» con V ; se risultano maggiori, la cifra corrente di q viene posta a 1 e viene effettuata la sottrazione R_i-V .

Inoltre, il dividendo può essere caricato in una coppia di registri **A.Q**: Q viene “svuotato” man mano che avvengono gli shift e può essere usato per memorizzare le cifre del quoziente calcolate a ogni passo. A conterrà il resto alla fine del processo.

Come visto per la procedura manuale, l'operazione di divisione richiede una successione di sottrazioni e di confronti. Idealmente il confronto potrebbe essere effettuato da un circuito **comparatore**, che riceve in ingresso due numeri interi A e B e produce due uscite α e β tali che:

- $\alpha = 1 \Rightarrow A > B$
- $\beta = 1 \Rightarrow A < B$
- $\alpha = \beta = 0 \Rightarrow A = B$



Per realizzare il comparatore su n bit è possibile collegare opportunamente dei comparatori elementari che confrontano cifre di un solo bit. Se ambedue i numeri A e B sono interi senza segno si verifica facilmente che confrontando i bit di A e quelli di B a partire da sinistra verso destra (iniziando quindi dai bit più significativi) il risultato del confronto viene determinato non appena uno dei due numeri ha in posizione i -esima un bit di valore 0 e l'altro lo ha di valore 1: il numero col bit a 1 è certamente il maggiore, indipendentemente dai valori dei bit meno significativi.

Il problema centrale della divisione è il calcolo della cifra del quoziente q_i come confronto fra V e $2R_i$ al passo i -esimo:

- Se $V > 2R_i \Rightarrow q_i = 0$
- Se $V \leq 2R_i \Rightarrow q_i = 1$

Se V è costituito da un numero elevato di cifre l'utilizzo di comparatori potrebbe risultare oneroso in termini di circuiti logici. Mentre, q_i può essere calcolato sottraendo V da $2R_i$ ed esaminando il segno della differenza: se è negativo $q_i = 0$, altrimenti $q_i = 1$. Si noti che la differenza $2R_i - V$ andrebbe comunque calcolata se $q_i = 1$ e in tal caso fornirebbe R_{i+1} .

I processi di determinazione di q_i e R_{i+1} possono essere fra loro combinati secondo due principali algoritmi: restoring and non restoring.

L'algoritmo di divisione prevede ad ogni passo i , il calcolo di:

$$R_{i+1} := 2R_i - q_i V;$$

Come si è visto, q_i può essere determinato con la differenza:

$$\Delta = 2R_i - V$$

- Se $\Delta \geq 0$ allora $q_i = 1$, e quindi effettivamente la differenza calcolata fornisce il valore di R_{i+1} ;
- Se $\Delta < 0$ allora $q_i = 0$ e quindi ho calcolato la quantità $2R_i - V$, mentre avrei dovuto calcolare $R_{i+1} = 2R_i$

Per avere il risultato corretto è necessario in tal caso effettuare un'operazione di *restoring* che consiste nel sommare V alla quantità Δ calcolata:

$$\begin{aligned}\Delta &:= R_{i+1} := 2R_i - V \\ R_{i+1} &:= R_{i+1} + V\end{aligned}$$

La tecnica di *non-restoring division* prende spunto dal fatto che a un'eventuale operazione di restoring (a) effettuata al passo i -esimo nel caso in cui $q_i = 0$, segue sempre, al passo $(i+1)$ -esimo, la sottrazione (b).

- a) $R_i := R_i + V$
- b) $R_{i+1} := 2R_i - V$

Le due operazioni possono essere fuse:

- i) $\Delta_i = R_{i+1} = 2R_i - V < 0 \Rightarrow q_i = 0$ effettuo il restoring:
 $R_{i+1} = R_{i+1} + V$
- i+1) $\Delta_{i+1} = R_{i+2} = 2R_{i+1} - V = 2(R_{i+1} + V) - V = 2R_{i+1} + 2V - V = 2R_{i+1} + V$

Se al passo i , dopo aver calcolato la differenza $(2R_i - V)$ risulta $q_i = 0$, la prossima operazione eseguita sarà una somma:

$$\Delta = 2R_{i+1} + V$$



Quindi occorre che l'algoritmo debba prevedere esplicitamente la possibilità di avere risultati negativi per la sottrazione e quindi sarà necessario un bit per memorizzare il segno, che ci permette di verificare se effettuare o meno l'operazione di restoring.

12.1.1 Descrizione del funzionamento del Divisore Non-Restoring

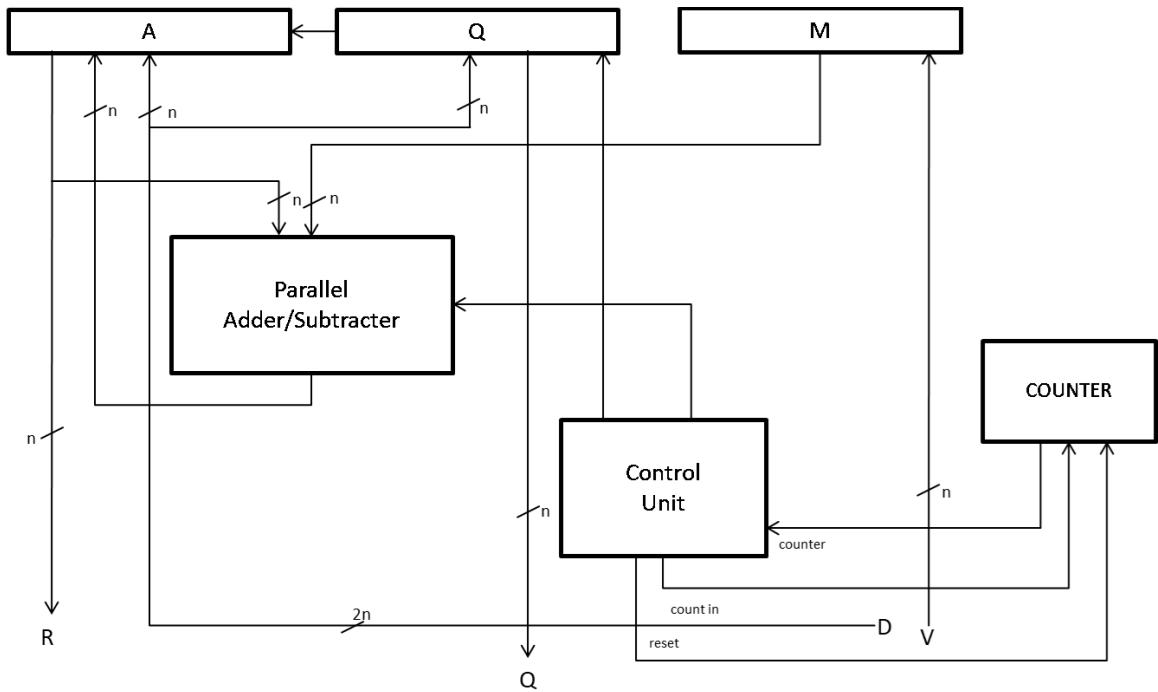


FIG 12.4: Struttura del Divisore

Grazie alle considerazioni fatte dall'analisi della procedura manuale, si deriva in Figura 12.4 la struttura del Divisore. In questo modello il divisore V, espresso su n bit, viene caricato in un registro M che rimane costante per tutti i passi del calcolo. Il dividendo D, espresso su $2n-1$ bit, viene memorizzato in A[n-2:0].Q nella fase di inizializzazione. I registri A e Q hanno parallelismo n . Il primo bit di A viene posto a 0 e la prima operazione è un left shift di A: in questo modo la cifra Q[0] è "libera" e può essere sostituita con la prima cifra calcolata del quoziente. Ad ogni passo i Q[0] conterrà la cifra del quoziente appena calcolata.

- Se si usa la tecnica del **restoring**, ad ogni passo dell'algoritmo viene effettuata la sottrazione del divisore dal dividendo parziale (shiftato) contenuto in A. Il segno della differenza determina la cifra q_i del quoziente e triggerà l'eventuale operazione di restoring.
- Se si usa il metodo del **non restoring**, ad ogni passo dell'algoritmo viene effettuata la somma o la sottrazione del divisore dal dividendo parziale (shiftato) contenuto in A, a seconda che il segno della precedente operazione di somma algebrica sia negativo o positivo.

Il **segno** viene memorizzato in un flip-flop S posto in testa al registro A e viene usato per determinare la cifra q_i del quoziente.

L'algoritmo del Divisore Non-Restoring è il seguente:

```

NRDivide:      (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

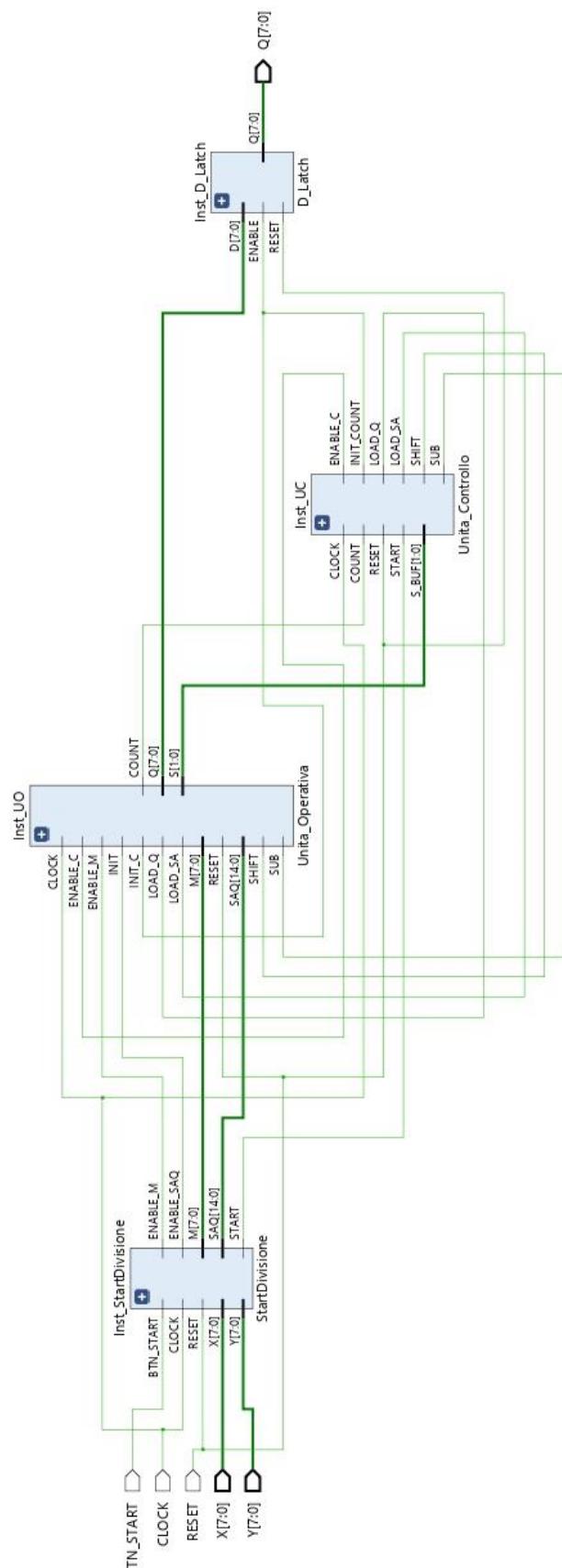
SETq: Q[0]:=not S;
 COUNT:=COUNT+1;

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 NRDivider;

12.1.2 Schematico in RTL Analysis di Vivado

**FIG. 12.5**

12.1.3 Codice VHDL del Divisore Non-Restoring

Divisore_NonRestoring.vhd

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Divisore_NonRestoring IS
    GENERIC (N : INTEGER := 8);
    PORT (
        CLOCK : IN STD_LOGIC;
        RESET : IN STD_LOGIC;
        X : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        BTN_START : IN STD_LOGIC;
        Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END Divisore_NonRestoring;

ARCHITECTURE Structural OF Divisore_NonRestoring IS
    COMPONENT Unita_Operativa
        GENERIC (N : INTEGER := 8);
        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 Unita_Controllo
        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;
        );
    END COMPONENT;

```



```

        INIT_COUNT : OUT STD_LOGIC
    );
END COMPONENT;

COMPONENT StartDivisione
    GENERIC (N : INTEGER);
    PORT (
        CLOCK : IN STD_LOGIC;
        RESET : IN STD_LOGIC;
        X : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
        Y : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0);
        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;

COMPONENT D_Latch IS
    PORT (
        Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        ENABLE : IN STD_LOGIC;
        RESET : IN STD_LOGIC;
        D : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END COMPONENT;

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 ENABLE_EXIT : STD_LOGIC;
SIGNAL INIT_COUNT_TEMP : STD_LOGIC;
SIGNAL ERR_TEMP : STD_LOGIC;

BEGIN

    Inst_D_Latch : D_Latch
    PORT MAP (
        Q => Q,
        ENABLE => INIT_COUNT_TEMP,
        RESET => RESET,
        D => Q_TEMP

```



```

);
Inst_UO : Unita_Operativa
GENERIC MAP(N => 8)
PORT MAP(
    CLOCK => CLOCK,
    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_UC : Unita_Controllo
PORT MAP(
    CLOCK => CLOCK,
    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_StartDivisione : StartDivisione
GENERIC MAP(N => 8)
PORT MAP(
    CLOCK => CLOCK,
    RESET => RESET,
    X => X,
    Y => Y,
    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
);

```

END Structural;

Per la progettazione dell'unità di controllo si è seguito l'algoritmo mostrato nel paragrafo 12.1.1, in particolare si è ricavato l'automa in Figura 12.5, da cui deriva il corrispondente codice VHDL.

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.

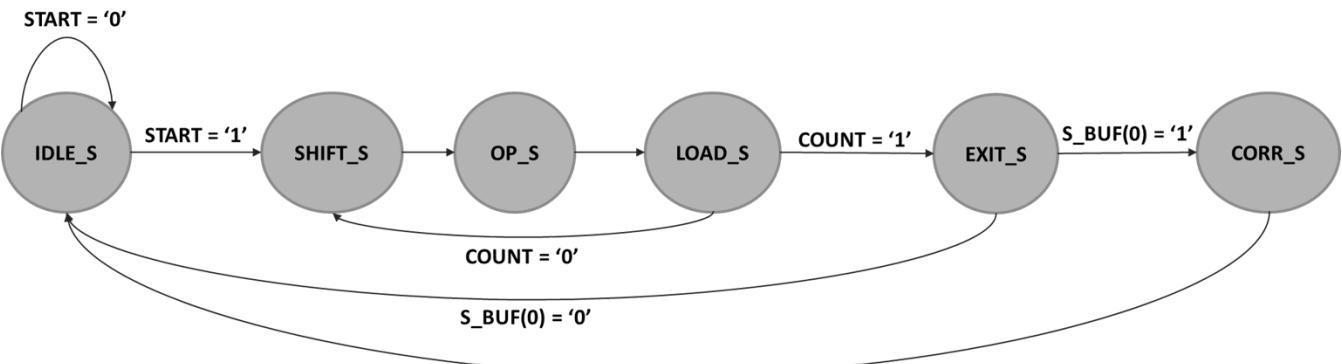


FIG. 12.6: Automa dell'unità di controllo del Divisore Non-Restoring

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 segno 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

Unita_Controllo.vhd

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Unita_Controllo 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 Unita_Controllo;

ARCHITECTURE Behavioral OF Unita_Controllo IS

TYPE STATE_TYPE IS
IDLE_S,
SHIFT_S,
OP_S,
LOAD_S,
EXIT_S,
CORR_S);

-- SEGNALI TEMPORANEI
SIGNAL CURRENT_STATE, NEXT_STATE : STATE_TYPE;

BEGIN

CURRENt_STATE_PROC : PROCESS (CLOCK, RESET)
BEGIN
    IF (RESET = '1') THEN
        CURRENT_STATE <= IDLE_S;
    ELSIF (rising_edge(CLOCK)) THEN
        CURRENT_STATE <= NEXT_STATE;
    END IF;
END PROCESS CURRENt_STATE_PROC;

NEXT_STATE_PROC : PROCESS (CURRENT_STATE, S_BUF, COUNT, START)
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 =>
            INIT_COUNT <= '1';
            IF (START = '1') THEN
                NEXT_STATE <= SHIFT_S;
            ELSE
                NEXT_STATE <= IDLE_S;
            END IF;

        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;

        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;

```

StartDivisione.vhd

```

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 StartDivisione IS
    GENERIC (N : INTEGER := 8);
    PORT (
        CLOCK : IN STD_LOGIC;

```



```

RESET : IN STD_LOGIC;
X : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0); --switch
Y : IN STD_LOGIC_VECTOR(N - 1 DOWNTO 0); --switch
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 StartDivisione;

```

ARCHITECTURE Behavioral **OF** StartDivisione **IS**

```

SIGNAL ERR_TEMP : STD_LOGIC;
SIGNAL M_TEMP : STD_LOGIC_VECTOR(N - 1 DOWNTO 0);

CONSTANT CONST : INTEGER := 0;
BEGIN
    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;
        ELSIF (rising_edge(CLOCK)) THEN ---qui ci mette il DIVIDENDO
            IF (BTN_START = '1' AND i = 0) THEN
                SAQ <= "0000000" & X;
                ENABLE_SAQ <= '1';
                M <= Y;
                M_TEMP <= Y;
                ENABLE_M <= '1';
                i := 1;
            ELSIF (i = 1) THEN
                IF (M_TEMP = STD_LOGIC_VECTOR(to_unsigned(CONST,
                M_TEMP' length))) THEN
                    ERR_TEMP <= '1'; --divisione con lo 0 grande
error
                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;

```



```

    END IF;
END PROCESS;

ERR <= ERR_TEMP;

END Behavioral;

```

Unita_Operativa.vhd

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Unita_Operativa IS
  GENERIC (N : INTEGER := 8);
  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 Unita_Operativa;

```

```
ARCHITECTURE Structural OF Unita_Operativa 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;
  );
END COMPONENT;

```



```

        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));
    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;
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
) ;

Inst_Counter : Counter
GENERIC MAP(
    M => 8,
    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 Structural;

```

12.1.4 La simulazione del Divisore Non-Restoring

tb_Divisore_NonRestoring.vhd

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY tb_Divisore_NonRestoring IS
END tb_Divisore_NonRestoring;

ARCHITECTURE Behavioral OF tb_Divisore_NonRestoring IS

COMPONENT Divisore_NonRestoring IS
  GENERIC (N : INTEGER := 8);
  PORT (
    CLOCK : IN STD_LOGIC;
    RESET : IN STD_LOGIC;
    X : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    BTN_START : IN STD_LOGIC;
    Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END COMPONENT;
--Inputs
SIGNAL CLOCK : STD_LOGIC := '0';
SIGNAL RESET : STD_LOGIC := '0';
SIGNAL X : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
SIGNAL Y : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
SIGNAL BTN_START : STD_LOGIC := '0';

--Outputs
SIGNAL Q : STD_LOGIC_VECTOR(7 DOWNTO 0);

-- 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
    X <= "00011101"; -- 8

    WAIT FOR 12 ns;

    Y <= "00000100"; -- 4

    WAIT FOR 12 ns;

    BTN_START <= '1';

    WAIT FOR 12 ns;
    BTN_START <= '0';
    WAIT FOR 400 ns;
    --          RESET <= '1';
    WAIT FOR 100 ns;
    --          RESET <= '0';
    -- insert stimulus here
    X <= "11111111"; -- 255

    WAIT FOR 12 ns;

    Y <= "00001110"; -- 14

    WAIT FOR 12 ns;

    BTN_START <= '1';

    WAIT FOR 12 ns;
    BTN_START <= '0';

    WAIT FOR 400 ns;
    --          RESET <= '1';
    WAIT FOR 100 ns;
    --          RESET <= '0';

    WAIT;
END PROCESS;

END Behavioral;
```



FIG. 12.7: Simulazione del Divisore Non-Restoring

12.2 Applicazioni reali e timing analysis del Divisore Non-Restoring

Il Divisore Non-Restoring visto nel paragrafo precedente, può essere usato come componente in un'applicazione reale, ad esempio per calcolare la funzione di hash crittografica tramite il metodo della divisione, oppure come componente di una calcolatrice.

Tuttavia, prima di inserire questo componente nella Calcolatrice, effettuiamo una timing analysis per verificare che i constraint sul tempo sono soddisfatti e quindi non ci sono problemi di temporizzazione. Come è possibile notare dalla Figura 12.7, il worst negative slack è di 0.687, quindi il Divisore Non-Restoring soddisfa il vincolo sul periodo di 10ns del clock della board.

Design Timing Summary						
WNS (ns)	TNS (ns)	TNS Failing Endpoints	TNS Total Endpoints	WHS (ns)	THS (ns)	THS Failing Endpoints
0.687	0.000	0	107	0.227	0.000	0

All user specified timing constraints are met.

Max Delay Paths

```

Slack (MET) : 0.687ns (required time - arrival time)
Source: Inst_UC/CURRENT_STATE_reg[1]/C
        (rising edge-triggered cell FDCE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Destination: Inst_UO/Inst_RegisterSAQ/Y_TEMP_reg[0]/D
        (falling edge-triggered cell FDCE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
Path Group: sys_clk_pin
Path Type: Setup (Max at Slow Process Corner)
Requirement: 5.000ns (sys_clk_pin fall@5.000ns - sys_clk_pin rise@0.000ns)
Data Path Delay: 4.290ns (logic 1.430ns (33.333%) route 2.860ns (66.667%))
Logic Levels: 6 (LUT3=1 LUT5=2 LUT6=3)
Clock Path Skew: -0.023ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD): 5.018ns = (10.018 - 5.000)
    Source Clock Delay (SCD): 5.314ns
    Clock Pessimism Removal (CPR): 0.273ns
Clock Uncertainty: 0.035ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter (TSJ): 0.071ns
    Total Input Jitter (TIJ): 0.000ns
    Discrete Jitter (DJ): 0.000ns
    Phase Error (PE): 0.000ns

```

FIG. 12.8: Timing Analysis del Divisore Non-Restoring



12.2.1 Descrizione del funzionamento della Calcolatrice

Per il progetto della Calcolatrice si segue lo schema in Figura 12.8, in cui si sono utilizzati come componenti della Calcolatrice, il Moltiplicatore di Robertson del Capitolo 11 ed il componente Adder/Subber usato al suo interno, una volta come Sommatore ed un'altra volta come Sottrattore. Ovviamente, come Divisore si utilizza quello Non-Restoring, i componenti Encoder4_2 e Mux_4_1 servono per mostrare in output il risultato dell'operazione che ha dato il relativo start/stop.

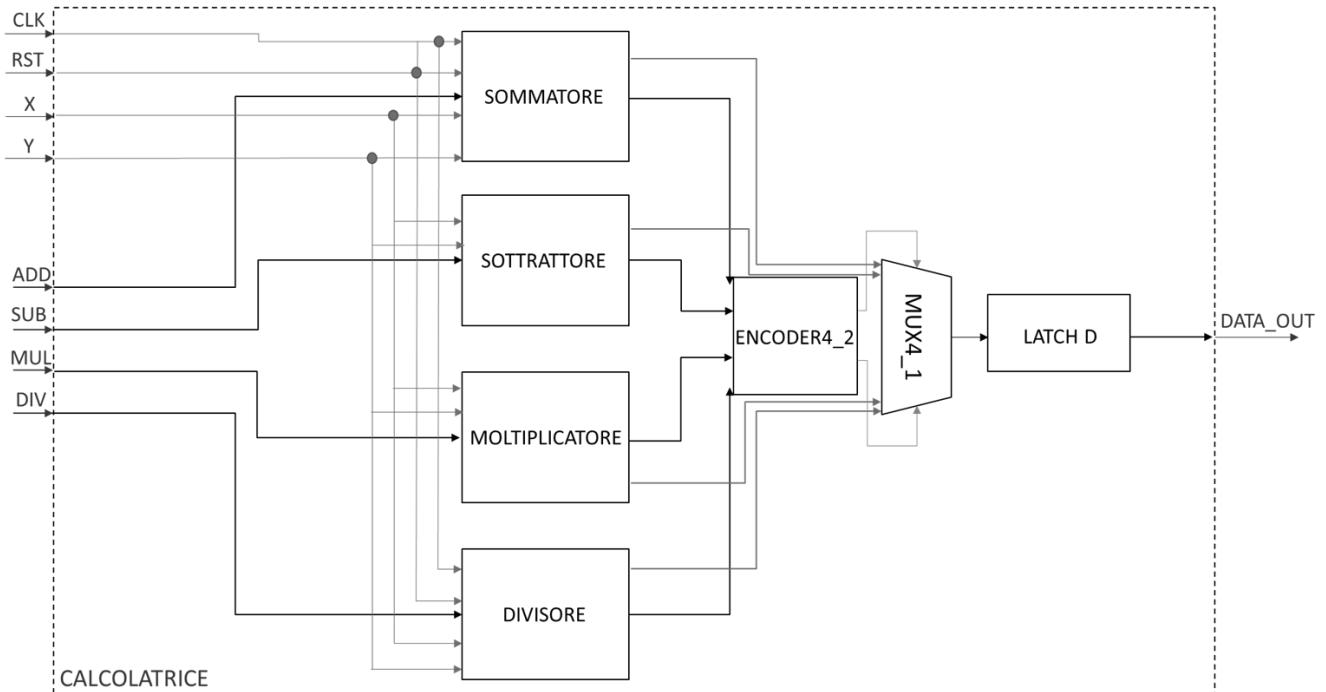


FIG. 12.9: Struttura della Calcolatrice

12.2.2 Schematico in RTL Analysis di Vivado

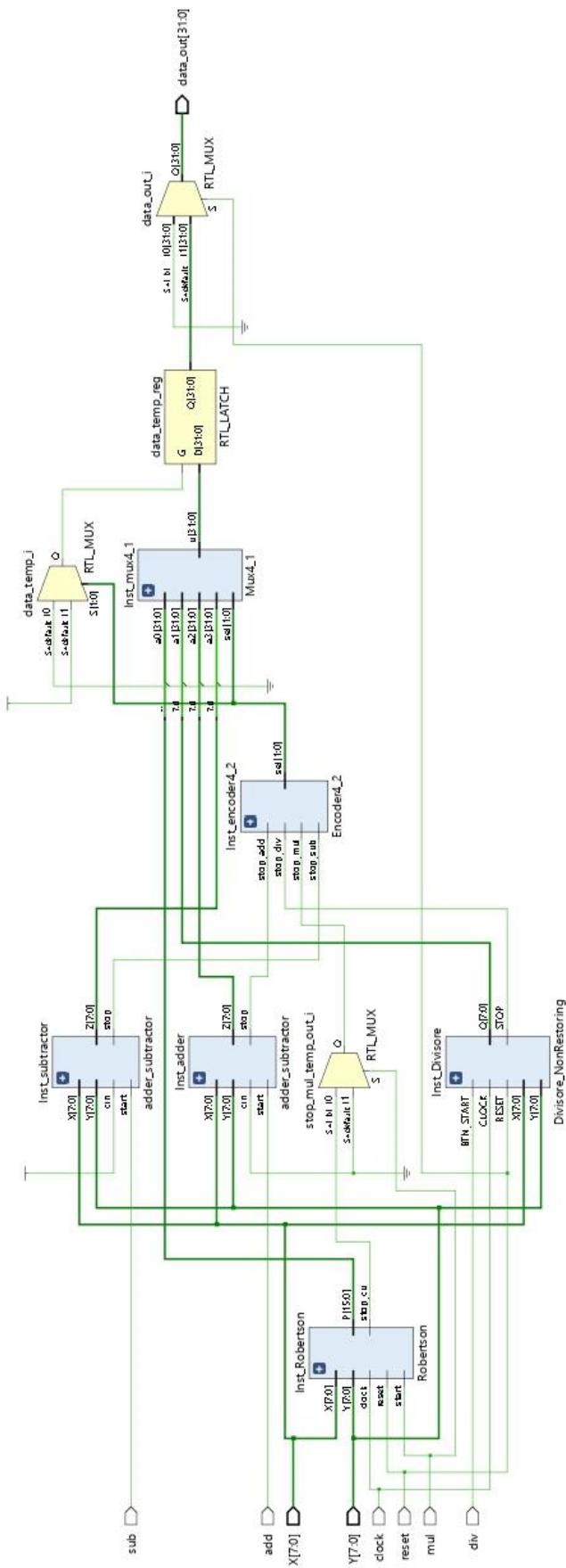


FIG. 12.10

12.2.3 Codice VHDL della Calcolatrice

Calcolatrice.vhd

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Calcolatrice IS
    PORT (
        clock, reset : IN STD_LOGIC;
        add, sub, mul, div : IN STD_LOGIC;
        X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        data_out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END Calcolatrice;

ARCHITECTURE Structural OF Calcolatrice IS

COMPONENT Robertson IS
    PORT (
        clock, reset, start : IN STD_LOGIC;
        X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        stop : OUT STD_LOGIC;
        P : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        stop_cu : OUT STD_LOGIC);
    END COMPONENT;

COMPONENT Divisore_NonRestoring IS
    PORT (
        CLOCK : IN STD_LOGIC;
        RESET : IN STD_LOGIC;
        X : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        BTN_START : IN STD_LOGIC;
        STOP : OUT STD_LOGIC;
        Q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
    END COMPONENT;

COMPONENT adder_subtractor IS
    PORT (
        X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        cin : IN STD_LOGIC;
        start : IN STD_LOGIC;
        Z : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        stop : OUT STD_LOGIC;
        cout : OUT STD_LOGIC);
    END COMPONENT;

COMPONENT Encoder4_2 IS
    PORT (
        stop_add, stop_sub, stop_mul, stop_div : IN STD_LOGIC;
        sel : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
    END COMPONENT;

COMPONENT Mux4_1 IS

```



```

PORT (
    a0 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    a1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    a2 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    a3 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    u : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END COMPONENT;
SIGNAL data_dec_out : STD_LOGIC_VECTOR(31 DOWNTO 0) := (OTHERS => '0');
SIGNAL data_temp : STD_LOGIC_VECTOR(31 DOWNTO 0) := (OTHERS => '0');
SIGNAL add_out : STD_LOGIC_VECTOR(31 DOWNTO 0) := (OTHERS => '0');
SIGNAL sub_out : STD_LOGIC_VECTOR(31 DOWNTO 0) := (OTHERS => '0');
SIGNAL mul_out : STD_LOGIC_VECTOR(31 DOWNTO 0) := (OTHERS => '0');
SIGNAL div_out : STD_LOGIC_VECTOR(31 DOWNTO 0) := (OTHERS => '0');
SIGNAL stop_mul_temp_out : STD_LOGIC;
SIGNAL stop_mul_temp : STD_LOGIC;
SIGNAL stop_add_temp : STD_LOGIC;
SIGNAL stop_sub_temp : STD_LOGIC;
SIGNAL stop_div_temp : STD_LOGIC;
SIGNAL sel_temp : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL carry_out_add : STD_LOGIC;
SIGNAL carry_out_sub : STD_LOGIC;
BEGIN
    data_out <= (OTHERS => '0') WHEN reset = '1' ELSE
        data_temp;

    stop_mul_temp_out <= stop_mul_temp WHEN mul = '1' ELSE
        '0';
    Inst_Robertson : Robertson
    PORT MAP(
        clock => clock,
        reset => reset,
        start => mul,
        X => X,
        Y => Y,
        stop => OPEN,
        P => mul_out(15 DOWNTO 0),
        stop_cu => stop_mul_temp
    );
    Inst_adder : adder_subtractor
    PORT MAP(
        X => X,
        Y => Y,
        start => add,
        cin => '0',
        Z => add_out(7 DOWNTO 0),
        stop => stop_add_temp,
        cout => carry_out_add
    );
    Inst_subtractor : adder_subtractor
    PORT MAP(

```



```

        X => X,
        Y => Y,
        start => sub,
        cin => '1',
        Z => sub_out(7 DOWNTO 0),
        stop => stop_sub_temp,
        cout => carry_out_sub
    ) ;

Inst_Divisore : Divisore_NonRestoring
PORT MAP(
    CLOCK => clock,
    RESET => reset,
    X => X,
    Y => Y,
    BTN_START => div,
    STOP => stop_div_temp,
    Q => div_out(7 DOWNTO 0)
) ;

Inst_encoder4_2 : Encoder4_2
PORT MAP(
    stop_add => stop_add_temp,
    stop_sub => stop_sub_temp,
    stop_mul => stop_mul_temp_out,
    stop_div => stop_div_temp,
    sel => sel_temp
) ;

data_temp <= data_temp WHEN sel_temp = "--
                           data_dec_out;

Inst_mux4_1 : Mux4_1
PORT MAP(
    a0 => mul_out,
    a1 => div_out,
    a2 => add_out,
    a3 => sub_out,
    sel => sel_temp,
    u => data_dec_out
) ;

END Structural;

```

Encoder4_2.vhd

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Encoder4_2 IS
    PORT (
        stop_add, stop_sub, stop_mul, stop_div : IN STD_LOGIC;
        sel : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END Encoder4_2;

```

```
ARCHITECTURE Behavioral OF Encoder4_2 IS
```



BEGIN

```

sel <= "00" WHEN stop_mul = '1' ELSE
    "01" WHEN stop_div = '1' ELSE
    "10" WHEN stop_add = '1' ELSE
    "11" WHEN stop_sub = '1' ELSE
"--";

```

END Behavioral;**Mux4_1.vhd**

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

```

```

ENTITY Mux4_1 IS
  PORT (
    a0 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    a1 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    a2 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    a3 : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    u : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END Mux4_1;

```

ARCHITECTURE dataflow **OF** Mux4_1 **IS****BEGIN**

```

u <= a0 WHEN sel = "00" ELSE
  a1 WHEN sel = "01" ELSE
  a2 WHEN sel = "10" ELSE
  a3 WHEN sel = "11" ELSE
  (OTHERS => '0');

```

END dataflow;**12.2.4 La simulazione della Calcolatrice****tb_Calcolatrice.vhd**

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

```

```

ENTITY tb_Calcolatrice IS
  -- Port();
END tb_Calcolatrice;

```

ARCHITECTURE tb **OF** tb_Calcolatrice **IS**

```

COMPONENT Calcolatrice
  PORT (
    clock : IN STD_LOGIC;

```



```

        reset : IN STD_LOGIC;
        add : IN STD_LOGIC;
        sub : IN STD_LOGIC;
        mul : IN STD_LOGIC;
        div : IN STD_LOGIC;
        X : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Y : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        data_out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
END COMPONENT;

SIGNAL clock : STD_LOGIC;
SIGNAL reset : STD_LOGIC;
SIGNAL add : STD_LOGIC;
SIGNAL sub : STD_LOGIC;
SIGNAL mul : STD_LOGIC;
SIGNAL div : STD_LOGIC;
SIGNAL X : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL Y : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL data_out : STD_LOGIC_VECTOR(31 DOWNTO 0);

CONSTANT TbPeriod : TIME := 10 ns; -- EDIT Put right period here
SIGNAL TbClock : STD_LOGIC := '0';
SIGNAL TbSimEnded : STD_LOGIC := '0';

BEGIN

dut : Calcolatrice
PORT MAP(
    clock => clock,
    reset => reset,
    add => add,
    sub => sub,
    mul => mul,
    div => div,
    X => X,
    Y => Y,
    data_out => data_out);

-- Clock generation
TbClock <= NOT TbClock AFTER TbPeriod/2 WHEN TbSimEnded /= '1' ELSE
'0';

-- EDIT: Check that clock is really your main clock signal
clock <= TbClock;

stimuli : PROCESS
BEGIN
    -- EDIT Adapt initialization as needed
    add <= '0';
    sub <= '0';
    mul <= '0';
    div <= '0';
    X <= (OTHERS => '0');
    Y <= (OTHERS => '0');

    -- Reset generation

```

```
-- EDIT: Check that reset is really your reset signal
reset <= '1';
WAIT FOR 100 ns;
reset <= '0';
WAIT FOR 100 ns;

-- EDIT Add stimuli here

X <= "00001110";
Y <= "00010000";

add <= '1';

WAIT FOR 12ns;

add <= '0';

WAIT FOR 100ns;

X <= "00011110";
Y <= "00010000";

sub <= '1';

WAIT FOR 12ns;

sub <= '0';

WAIT FOR 100ns;

X <= "01000110";
Y <= "00001001";

mul <= '1';

WAIT FOR 500ns;

mul <= '0';

WAIT FOR 100ns;

X <= "01011110";
Y <= "00001111";

div <= '1';

WAIT FOR 12ns;

div <= '0';

WAIT FOR 100ns;

WAIT FOR 1000 * TbPeriod;

-- Stop the clock and hence terminate the simulation
TbSimEnded <= '1';
```

```

    WAIT;
END PROCESS;

```

```
END tb;
```

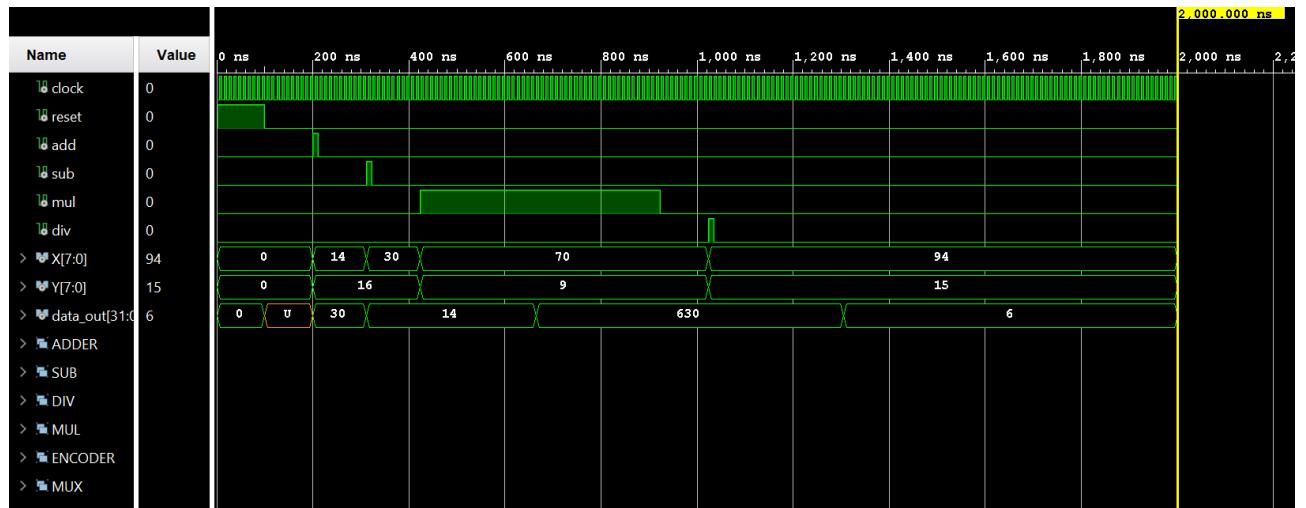


FIG 12.11: Simulazione della Calcolatrice