



**UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II**

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato di Architettura dei Sistemi Digitali

Prof.ssa Alessandra De Benedictis, a.a. 2023-24

Studenti:

Luigi Barbato M63001632

Claudia Castellano M63001618

Valerio Domenico Conte M63001606

Saverio Dell'Aversana M63001613

Pasquale Mottola M63001577

Indice

1 Reti combinatorie elementari	1
1.1 Esercizio 1: Multiplexer 16:1	1
1.1.1 Esercizio 1.1	2
1.1.2 Progetto e architettura	2
1.1.3 Implementazione	6
1.1.4 Simulazione	11
1.1.5 Esercizio 1.2	14
1.1.6 Progetto e architettura	14
1.1.7 Implementazione	17
1.1.8 Simulazione	21
1.1.9 Esercizio 1.3	25
1.1.10 Sintesi su board di sviluppo	25
1.2 Esercizio 2: Sistema ROM+M	30
1.2.1 Esercizio 2.1	31
1.2.2 Progetto e architettura	32
1.2.3 Implementazione	33
1.2.4 Simulazione	38
1.2.5 Esercizio 2.2	41
1.2.6 Sintesi su board di sviluppo	41

2 Reti sequenziali elementari	43
2.1 Esercizio 3: Riconoscitore di sequenze	43
2.1.1 Esercizio 3.1	44
2.1.2 Progetto e architettura	44
2.1.3 Implementazione	47
2.1.4 Simulazione	52
2.1.5 Esercizio 3.2	57
2.1.6 Sintesi su board di sviluppo	57
2.1.7 Timing analysis	64
2.2 Esercizio 4: Shift register	66
2.2.1 Esercizio 4.1	67
2.2.2 Progetto e architettura	68
2.2.3 Implementazione	71
2.2.4 Simulazione	78
2.3 Esercizio 5: Cronometro	89
2.3.1 Esercizio 5.1	89
2.3.2 Progetto e architettura	90
2.3.3 Implementazione	93
2.3.4 Simulazione	107
2.3.5 Esercizio 5.2	110
2.3.6 Sintesi su board di sviluppo	111
2.3.7 Timing analysis	122
2.3.8 Esercizio 5.3	124
2.3.9 Sintesi su board di sviluppo	124
2.3.10 Timing analysis	140

2.4 Esercizio 6: Sistema di lettura-elaborazione-scrittura	
PO_PC	141
2.4.1 Esercizio 6.1	141
2.4.2 Progetto e architettura	142
2.4.3 Implementazione	144
2.4.4 Simulazione	151
2.4.5 Esercizio 6.2	153
2.4.6 Sintesi su board di sviluppo	154
2.4.7 Timing analysis	164
3 Macchine aritmetiche	166
3.1 Esercizio 7: Moltiplicatore di Booth	166
3.1.1 Esercizio 7.1	167
3.1.2 Progetto e architettura	167
3.1.3 Implementazione	171
3.1.4 Simulazione	191
3.1.5 Esercizio 7.2	195
3.1.6 Sintesi su board di sviluppo	196
3.1.7 Timing analysis	202
4 Comunicazione con handshaking	204
4.1 Esercizio 8: Comunicazione con handshaking	204
4.1.1 Esercizio 8.1	204
4.1.2 Progetto e architettura	205
4.1.3 Implementazione	208
4.1.4 Simulazione	222

5 Processore	225
5.1 Esercizio 9: Processore	225
5.1.1 Esercizio 9.1	225
5.1.2 Analisi architettura e istruzioni	226
5.1.3 Modifica del codice operativo	230
6 Interfaccia seriale	232
6.1 Esercizio 10: Interfaccia seriale	232
6.1.1 Esercizio 10.1	232
6.1.2 Progetto e architettura	233
6.1.3 Implementazione	237
6.1.4 Simulazione	251
7 Switch multistadio	256
7.1 Esercizio 11: Switch multistadio	257
7.1.1 Esercizio 11.1	257
7.1.2 Progettazione e architettura	257
7.1.3 Implementazione	260
7.1.4 Simulazione	276
8 Appendice	280
8.1 Multiplexer 2:1	280
8.1.1 Progetto e architettura	280
8.1.2 Implementazione	280
8.2 Multiplexer 4:1	282
8.2.1 Progetto e architettura	282
8.2.2 Implementazione	283

8.3	Demultiplexer 1:2	285
8.3.1	Progetto e architettura	285
8.3.2	Implementazione	285
8.4	Contatore modulo N	286
8.4.1	Progetto e architettura	286
8.4.2	Implementazione	287
8.5	ROM sequenziale	289
8.5.1	Progetto e architettura	289
8.5.2	Implementazione	290
8.6	Memoria	291
8.6.1	Progetto e architettura	291
8.6.2	Implementazione	292
8.7	Button Debouncer	294
8.7.1	Progetto e architettura	294
8.7.2	Implementazione	295

Capitolo 1

Reti combinatorie elementari

1.1 Esercizio 1: Multiplexer 16:1

Il **multiplexer**, insieme al demultiplexer, è una rete combinatoria fondamentale che si trova alla base di gran parte dei meccanismi di interconnessione tra sottosistemi che realizzando l'architettura di un sistema complesso. Il multiplexer (abbreviato MUX) consente infatti di collegare più segnali di sorgente a un'unica destinazione, operando come selettore di dati controllato. Questo dispositivo, nella sua versione più semplice, presenta tipicamente in ingresso M linee di selezione e $N = 2^M$ linee dato, mentre in uscita presenta un singolo segnale. Il dispositivo opera collegando una specifica linea di ingresso, determinata tramite il segnale di selezione, all'unica linea di uscita. In ingresso alle

linee di selezione è posta una stringa binaria che identifica una specifica linea di ingresso mediante un numero codificato in rappresentazione binaria posizionale. Nel nostro caso, il Multiplexer 16:1 presenterà una linea di uscita mentre in ingresso 16 linee dato e $\log_2 16 = 4$ linee di selezione.

1.1.1 Esercizio 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.

1.1.2 Progetto e architettura

Abbiamo adottato una progettazione bottom-up per la realizzazione della rete richiesta dalla traccia, progettando inizialmente il componente più semplice, cioè il **MUX 2:1**, la cui interfaccia è presentata in Figura 1.1. Abbiamo realizzato l'architettura di tale componente più semplice attraverso una descrizione di tipo *dataflow*, sfruttando l'espressione booleana che ne descrive il funzionamento: $Y = (I_0 \cdot \bar{S}) + (I_1 \cdot S)$. In Figura 1.2 possiamo osservare lo schema logico corrispondente a tale espressione.

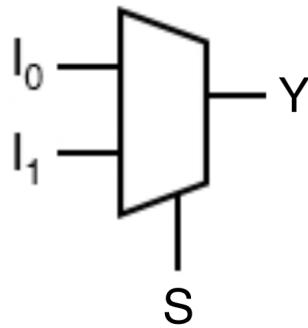


Figura 1.1: Interfaccia di un MUX 2:1

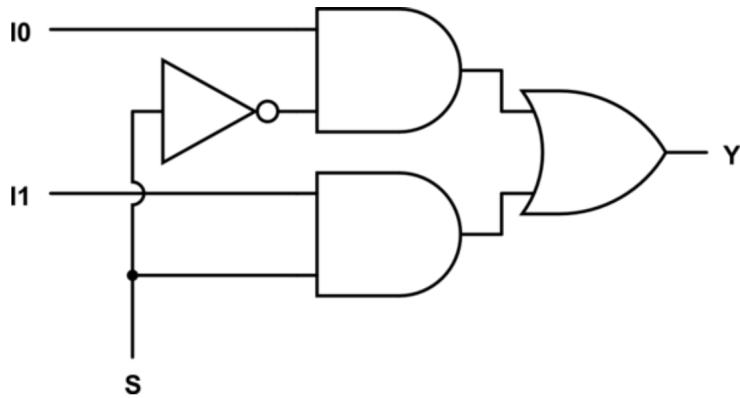


Figura 1.2: Schema logico di un MUX 2:1

Il passo successivo è stato poi quello di realizzare per composizione un **MUX 4:1**, fornendo una descrizione strutturale della sua architettura basata sull'utilizzo di tre componenti MUX 2:1. In Figura 1.3 possiamo osservare la struttura del MUX 4:1 realizzato per composizione. I primi due MUX 2:1 prendono in ingresso il segnale di input e le loro uscite sono poi collegate al terzo MUX 2:1, che fornisce l'uscita della rete complessiva. Il primo bit del segnale di selezione regola i MUX nel primo layer mentre il secondo bit interviene sul MUX del layer successivo.

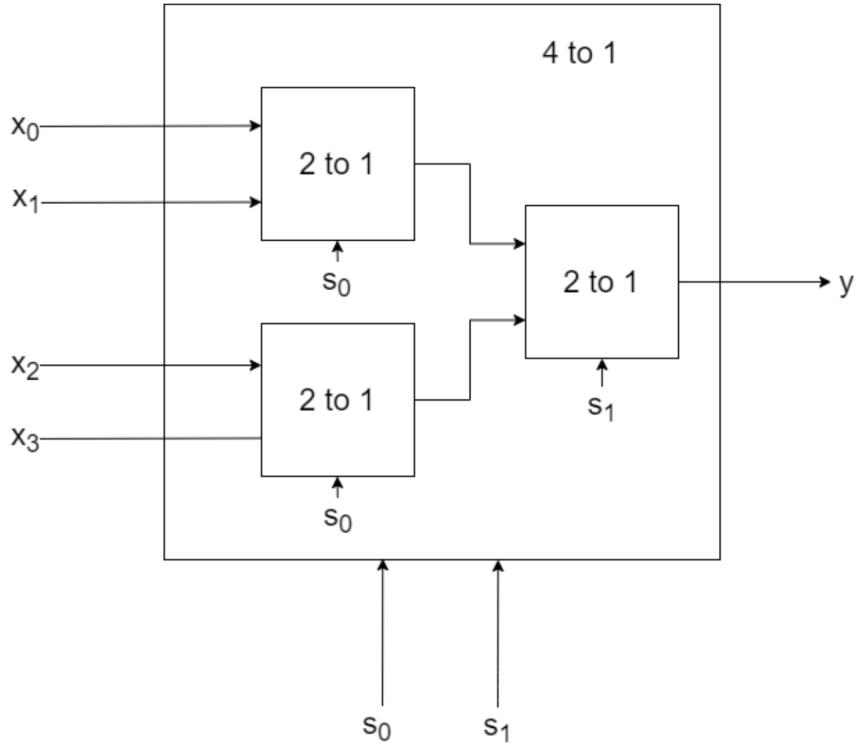


Figura 1.3: Interfaccia del MUX 4:1

Infine, seguendo un ragionamento analogo, abbiamo ottenuto una descrizione strutturale della nostra rete di interesse, cioè il **MUX 16:1**, per composizione di cinque MUX 4:1, come possiamo vedere in Figura 1.4. Questa volta l'input complessivamente presenta 16 bit e i quattro MUX 4:1 del primo layer ne ricevono quattro ognuno; la selezione è regolata dai primi due bit del segnale di selezione; il quinto MUX 4:1, disposto nel secondo layer, raccoglie le quattro linee di uscita dei MUX appartenenti al layer precedente e, a seconda degli ultimi due bit del segnale di selezioni, fornisce l'uscita della rete.

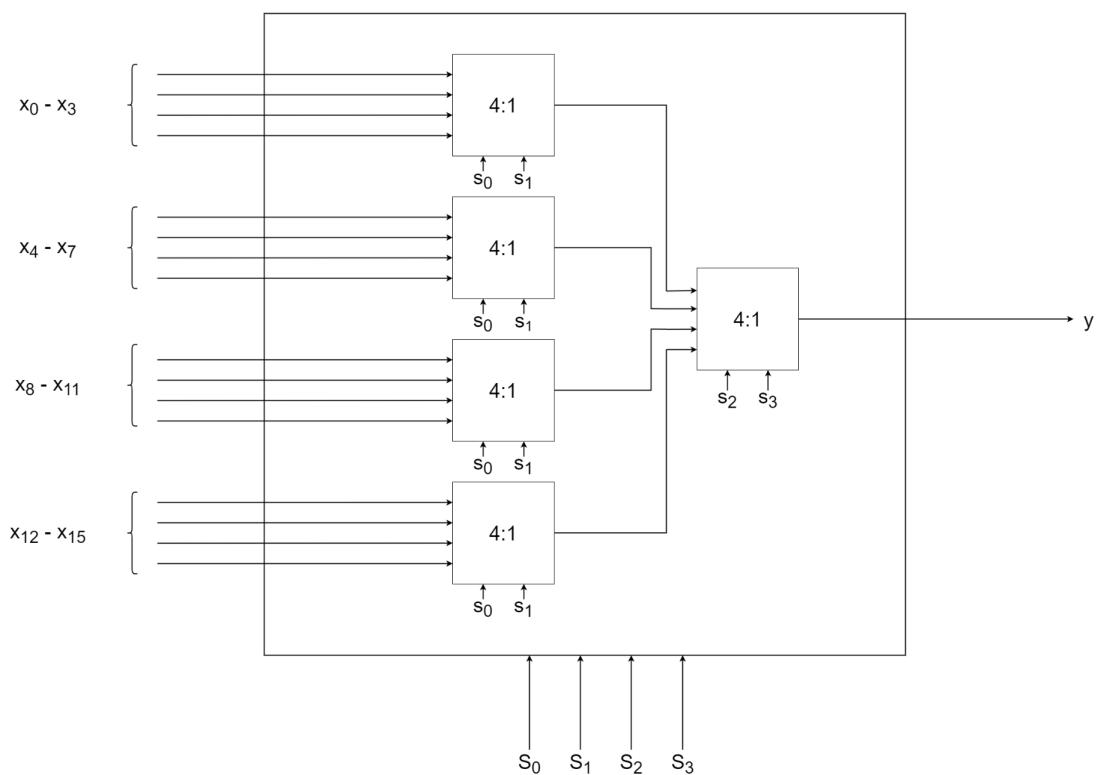


Figura 1.4: Interfaccia del MUX 16:1

1.1.3 Implementazione

Procediamo a presentare l'implementazione in linguaggio **VHDL** dei tre componenti di cui si è parlato. Come anticipato, abbiamo realizzato il **MUX 2:1** fornendone una descrizione *dataflow*, cioè basata sull'espressione logica da essa realizzata, assegnandone il risultato all'uscita y in maniera concorrente. Chiaramente, la definizione dell'*architecture* è preceduta dalla dichiarazione dell'*entity*, attraverso la quale abbiamo definito l'interfaccia della rete, avente due bit di ingresso dati a_0 e a_1 , un ingresso di selezione s e una linea di uscita y .

```
ENTITY mux_2_1 IS
PORT (
  a0 : IN STD_LOGIC;
  a1 : IN STD_LOGIC;
  s : IN STD_LOGIC;
  y : OUT STD_LOGIC
);
END mux_2_1;

ARCHITECTURE dataflow OF mux_2_1 IS

BEGIN
  y <= ((a0 AND (NOT s)) OR (a1 AND s));
END dataflow;
```

Per quanto riguarda il **MUX 4:1**, nell'*entity* abbiamo definito un

vettore di 4 bit per il segnale di ingresso dati, un vettore di due bit per l'ingresso di selezione e chiaramente un bit per l'uscita. L'architecture invece è di tipo *structural*, dunque al suo interno abbiamo definito i tre componenti MUX 2:1 che realizzano la rete e i collegamenti tra di essi, sfruttando a tal fine un segnale di appoggio per la trasmissione dei risultati intermedi.

```
ENTITY mux_4_1 IS
PORT (
  b : IN STD_LOGIC_VECTOR(0 TO 3);
  r : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
  u : OUT STD_LOGIC
);
END mux_4_1;

ARCHITECTURE structural OF mux_4_1 IS
SIGNAL temp : STD_LOGIC_VECTOR(0 TO 1);

COMPONENT mux_2_1
PORT (
  a0 : IN STD_LOGIC;
  a1 : IN STD_LOGIC;
  s : IN STD_LOGIC;
  y : OUT STD_LOGIC
);
END COMPONENT;

BEGIN
```

```

mux0 : mux_2_1
PORT MAP (
    a0 => b(0),
    a1 => b(1),
    s => r(0),
    y => temp(0)
);

```

```

mux1 : mux_2_1
PORT MAP (
    a0 => b(2),
    a1 => b(3),
    s => r(0),
    y => temp(1)
);

```

```

mux2 : mux_2_1
PORT MAP (
    a0 => temp(0),
    a1 => temp(1),
    s => r(1),
    y => u
);
END structural;

```

Il MUX 16:1 è definito in maniera del tutto analoga al MUX 4:1, con le opportune modifiche al numero di bit per le linee di ingresso dati, selezione e appoggio, e ai componenti istanziati per la composizione

della rete complessiva (cioè cinque MUX 4:1).

```
ENTITY mux_16_1 IS
PORT (
c : IN STD_LOGIC_VECTOR (0 TO 15);
t : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
Y : OUT STD_LOGIC
);
END mux_16_1;

ARCHITECTURE structural OF mux_16_1 IS
SIGNAL temp : STD_LOGIC_VECTOR (0 TO 3);

COMPONENT mux_4_1
PORT (
b : IN STD_LOGIC_VECTOR(0 TO 3);
r : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
u : OUT STD_LOGIC
);
END COMPONENT;

BEGIN
mux0 : mux_4_1
PORT MAP (
b => c(0 to 3),
r => t(0 to 1),
u => temp(0)
);
```

CAPITOLO 1. RETI COMBINATORIE ELEMENTARI

```
mux1 : mux_4_1
```

```
PORT MAP (
    b => c(4 to 7),
    r => t(0 to 1),
    u => temp(1)
);
```

```
mux2 : mux_4_1
```

```
PORT MAP (
    b => c(8 to 11),
    r => t(0 to 1),
    u => temp(2)
);
```

```
mux3 : mux_4_1
```

```
PORT MAP (
    b => c(12 to 15),
    r => t(0 to 1),
    u => temp(3)
);
```

```
mux4 : mux_4_1
```

```
PORT MAP (
    b => temp,
    r => t(2 to 3),
    u => Y
);
END structural;
```

1.1.4 Simulazione

Per verificare il corretto funzionamento della rete da noi implementata, abbiamo scritto un *testbench* per definire gli input con cui stimolare il MUX 16:1 e le asserzioni sugli output da verificare. In particolare, abbiamo considerato come input la stringa "1010101010101010" e abbiamo stimolato la rete con quattro segnali di selezione diversi, per poi verificare tramite *assert* che l'output fosse quello corretto.

```
ENTITY mux_16_1_tb IS

END mux_16_1_tb;

ARCHITECTURE behavioral OF mux_16_1_tb IS

COMPONENT mux_16_1

PORT (
    c : IN STD_LOGIC_VECTOR (0 TO 15);
    t : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
    Y : OUT STD_LOGIC
);

END COMPONENT;

SIGNAL input : STD_LOGIC_VECTOR (0 TO 15) := (OTHERS => '0');
SIGNAL control : STD_LOGIC_VECTOR (3 DOWNTO 0) := (OTHERS => '0');
SIGNAL output : STD_LOGIC := '0';

BEGIN
```

```
uut : mux_16_1
PORT MAP (
    c(0 TO 15) => input(0 TO 15),
    t(3 DOWNTO 0) => control(3 DOWNTO 0),
    Y => output
);

stim_proc : PROCESS
BEGIN

    WAIT FOR 10 ns;

    input <= "101010101010101010";

    control <= "0000";
    WAIT FOR 10 ns;
    ASSERT output = '1'
        REPORT "test case 1 fallito" SEVERITY failure;

    control <= "0001";
    WAIT FOR 10 ns;
    ASSERT output = '0'
        REPORT "test case 2 fallito" SEVERITY failure;

    control <= "0010";
    WAIT FOR 10 ns;
    ASSERT output = '1'
        REPORT "test case 3 fallito" SEVERITY failure;
```

```

control <= "0011";
WAIT FOR 10 ns;
ASSERT output = '0'
    REPORT "test case 4 fallito" SEVERITY failure;

WAIT;
END PROCESS;
END;
    
```

In Figura 1.5 possiamo osservare graficamente l'output della simulazione, effettuata usando il software **Vivado**. Il segnale di selezione viene incrementato di 1 bit ogni 10 nano-secondi, in corrispondenza di tali cambiamenti osserviamo come l'uscita del sistema si alza o si abbassa a seconda della selezione: quando il segnale di selezione è 0000, l'uscita deve essere il primo bit della stringa, cioè 1, quando invece è 0001 l'uscita deve essere il secondo bit della stringa e così via.



Figura 1.5: Simulazione del testbench per il MUX 16:1

1.1.5 Esercizio 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**.*

1.1.6 Progetto e architettura

La rete richiesta dalla traccia, oltre al MUX 16:1, ha bisogno di un demultiplexer (abbreviato DEMUX) 1:4. Un demultiplexer è una rete combinatoria il cui funzionamento è speculare a quello del multiplexer: date N linee di ingresso, a seconda del segnale di selezione, viene abilitata una delle 2^N linee di uscita, lasciando le restanti linee al valore basso 0. Abbiamo realizzato il DEMUX 1:4 secondo un approccio bottom-up anche in questo caso, partendo dalla realizzazione di un semplice DEMUX 1:2, utilizzato poi per ottenere il DEMUX 1:4 per composizione.

Il **DEMUX 1:2** presenta una linea di ingresso dati, una linea di selezione e due linee di uscita, come possiamo vedere in Figura 1.6. Le espressioni booleane che forniscono il valore delle uscite sono:

- $y_0 = i \cdot \bar{r}$

- $y_1 = i \cdot r$

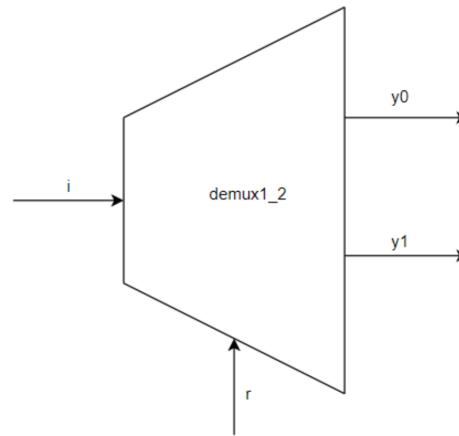


Figura 1.6: Interfaccia del DEMUX 1:2

Utilizzando tre DEMUX 1:2 in modo tale che il primo raccolga l'ingresso dati mentre gli altri due prendano in input le sue uscite, otteniamo allora una rete che a partire da una linea di ingresso e due segnali di selezione fornisce quattro linee di uscita, ed è così dunque che abbiamo realizzato il **DEMUX 1:4** in maniera modulare, come possiamo osservare in Figura 1.7.

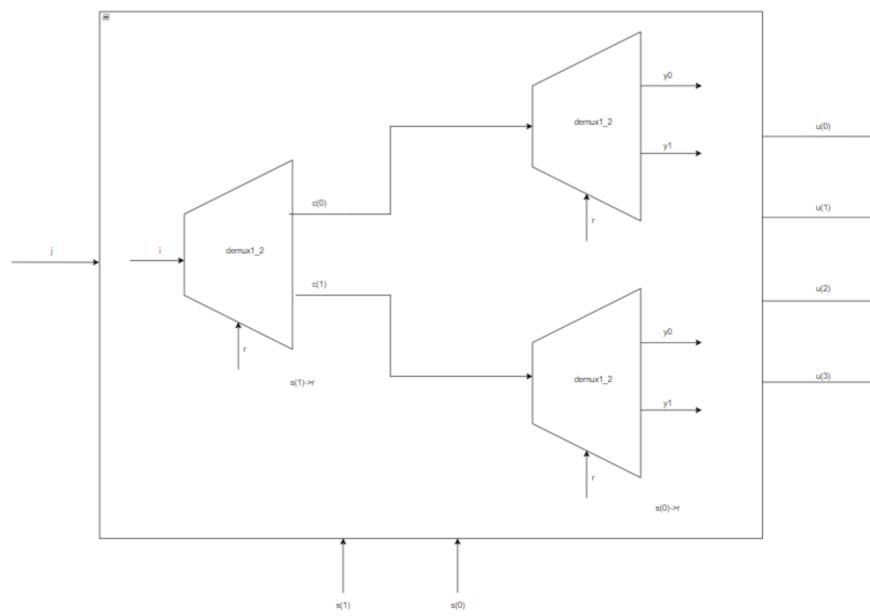


Figura 1.7: Interfaccia del DEMUX 1:4

La rete da noi implementata si ottiene ponendo la linea di uscita del MUX 16:1 in ingresso al DEMUX 1:4. Complessivamente, come rappresentato in Figura 1.8, la rete presenta 16 linee di ingresso dati, 6 linee di selezione (quattro per il MUX 16:1, due per il DEMUX 1:4) e 4 linee di uscita.

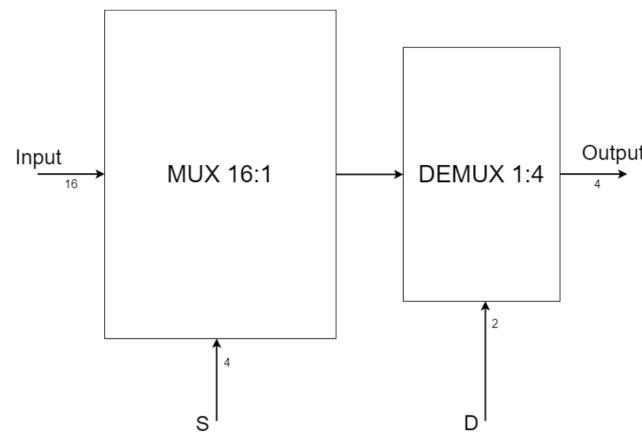


Figura 1.8: Rete di interconnessione 16:4

1.1.7 Implementazione

Abbiamo realizzato il componente **DEMUX 1:2** in VHDL definendone l'architettura secondo l'approccio dataflow; dato che il segnale di selezione deve essere tale che solo una delle due uscite sia alta, sulla base del segnale in ingresso, è stato per noi facile implementare l'espressione booleana che lo realizza.

```
ENTITY demux_1_2 IS
  PORT (
    d : IN STD_LOGIC;
    s : IN STD_LOGIC;
    y1 : OUT STD_LOGIC;
    y2 : OUT STD_LOGIC
  );
END demux_1_2;

ARCHITECTURE dataflow OF demux_1_2 IS

BEGIN
  y1 <= d AND (NOT s);
  y2 <= d AND s;

END dataflow;
```

Successivamente, abbiamo realizzato per composizione il **DEMUX 1:4**, aggiungendo una nuova linea di selezione e collegando le uscite di un primo DEMUX 1:2 agli ingressi di altri due componenti dello stesso tipo; la prima linea di selezione di occupa del primo layer mentre la seconda linea controlla i due componenti del layer successivo. In tal

modo, a partire da una sola linea di ingresso, si ottengono quattro linee di uscita.

```
ENTITY demux_1_4 IS
  PORT (
    i : IN STD_LOGIC;
    sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    y : OUT STD_LOGIC_VECTOR(0 TO 3)
  );
END demux_1_4;
```

```
ARCHITECTURE structural OF demux_1_4 IS
```

```
COMPONENT demux_1_2 IS
  PORT (
    d : IN STD_LOGIC;
    s : IN STD_LOGIC;
    y1 : OUT STD_LOGIC;
    y2 : OUT STD_LOGIC
  );
END COMPONENT;
```

```
SIGNAL u : STD_LOGIC_VECTOR(0 TO 1);
```

```
BEGIN
```

```
demux0 : demux_1_2
```

```
PORt MAP (
  d => i,
  s => sel(0),
  y1 => u(0),
```

```

y2 => u(1)

);

demux1 : demux_1_2
PORT MAP (
    d => u(0),
    s => sel(1),
    y1 => y(0),
    y2 => y(1)
);

demux2 : demux_1_2
PORT MAP (
    d => u(1),
    s => sel(1),
    y1 => y(2),
    y2 => y(3)
);

END structural;

```

A questo punto, ancora per composizione, abbiamo realizzato la nostra rete di interconnessione, definendone innanzitutto l’interfaccia all’intermo dell’entity:

- 16 linee di ingresso dati che rappresentano le sorgenti;
- 4 linee di uscita che rappresentano le destinazioni;
- 6 linee di selezione, 4 per il MUX e 2 per il DEMUX.

Dopodiché, abbiamo definito un'architecture di tipo *structural* utilizzando solo i due componenti di interesse, che abbiamo collegato in maniera opportuna tramite l'apposito segnale di appoggio *temp*.

```

ENTITY network_16_4 IS
    PORT (
        data : IN STD_LOGIC_VECTOR(0 TO 15);
        s_mux : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        s_demux : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        z : OUT STD_LOGIC_VECTOR(0 TO 3)
    );
END network_16_4;

ARCHITECTURE structural OF network_16_4 IS
    COMPONENT mux_16_1 IS
        PORT (
            c : IN STD_LOGIC_VECTOR(0 TO 15);
            t : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
            Y : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT demux_1_4 IS
        PORT (
            i : IN STD_LOGIC;
            sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
            y : OUT STD_LOGIC_VECTOR(0 TO 3)
        );
    END COMPONENT;

```

```

SIGNAL temp : STD_LOGIC := '0';

BEGIN

    mux : mux_16_1
    PORT MAP (
        c => data,
        t => s_mux,
        Y => temp
    );

    demux : demux_1_4
    PORT MAP (
        i => temp,
        sel => s_demux,
        y => z
    );
END structural;

```

1.1.8 Simulazione

Il testbench che abbiamo preparato per la rete di interconnessione utilizza lo stesso segnale di ingresso usato nel testbench precedente, cioè 10101010101010, e abbiamo verificato che, al variare dei segnali di selezione per i due componenti MUX e DEMUX, i segnali di uscita rispettassero l'espressione booleana che modella la rete. Il codice del testbench che abbiamo simulato è il seguente:

```
ENTITY network_16_4_tb IS
```

```

END network_16_4_tb;

ARCHITECTURE behavioral OF network_16_4_tb IS

COMPONENT network_16_4 IS
PORT (
    data : IN STD_LOGIC_VECTOR(0 TO 15);
    s_mux : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    s_demux : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    z : OUT STD_LOGIC_VECTOR(0 TO 3)
);
END COMPONENT;

SIGNAL ingresso : STD_LOGIC_VECTOR(0 TO 15)
:= (OTHERS => '0');

SIGNAL sel_mux : STD_LOGIC_VECTOR(3 DOWNTO 0)
:= (OTHERS => '0');

SIGNAL sel_demux : STD_LOGIC_VECTOR(1 DOWNTO 0)
:= (OTHERS => '0');

SIGNAL uscita : STD_LOGIC_VECTOR(0 TO 3);

BEGIN

uut : network_16_4
PORT MAP (
    data => ingresso,
    s_mux => sel_mux,
    s_demux => sel_demux,
    z => uscita
)

```

```

) ;

stimulus : PROCESS
BEGIN

    WAIT FOR 10 ns;

    ingresso <= "1010101010101010";

    sel_mux <= "0000";
    sel_demux <= "00";
    WAIT FOR 10 ns;
    ASSERT (uscita = "1000")
        REPORT "Test case 1 fallito" SEVERITY error;

    sel_mux <= "0000";
    sel_demux <= "10";
    WAIT FOR 10 ns;
    ASSERT (uscita = "0100")
        REPORT "Test case 2 fallito" SEVERITY error;

    sel_mux <= "0010";
    sel_demux <= "01";
    WAIT FOR 10 ns;
    ASSERT (uscita = "0010")
        REPORT "Test case 3 fallito" SEVERITY error;

    sel_mux <= "0010";
    sel_demux <= "11";

```

```

        WAIT FOR 10 ns;

        ASSERT (uscita = "0001")
            REPORT "Test case 4 fallito" SEVERITY error;

        WAIT;
    END PROCESS stimulus;

END behavioral;

```

L'output della simulazione è invece osservabile in Figura 1.9. Ogni 10 nano-secondi c'è una variazione dei segnali di selezione; quello del MUX è tale che la sua uscita sia pari ad 1 affinché possa essere visibile la variazione dell'uscita regolata dalla selezione effettuata sul DEMUX che, nel nostro caso, opera in maniera tale che le uscite corrispondano alle potenze decrescenti di 2.

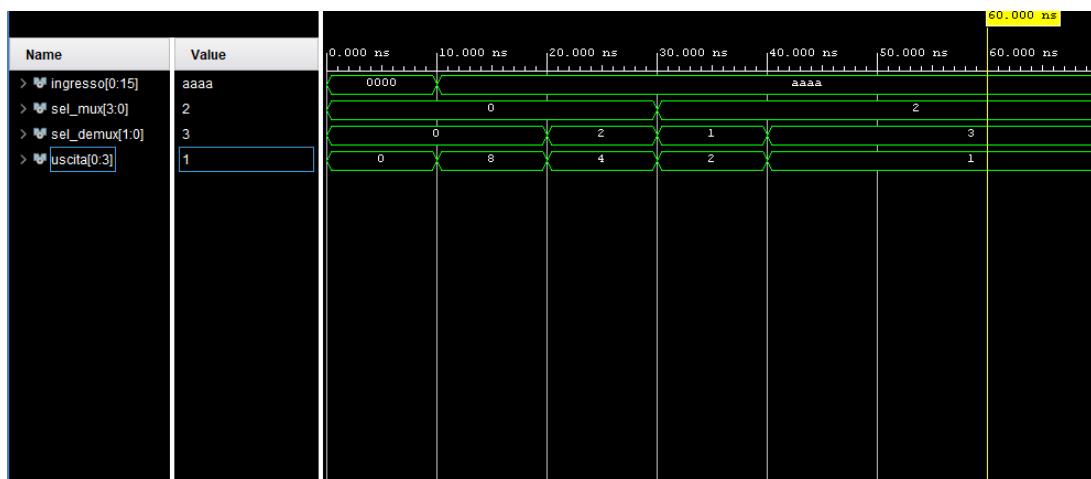


Figura 1.9: Simulazione testbench per la rete 16:4

1.1.9 Esercizio 1.3

Sintetizzare e implementare su board il progetto della rete di interconnessione sviluppato al punto 1.2, utilizzando gli switch per fornire gli input di selezione e i led per visualizzare i 4 bit di uscita. Per quanto riguarda i 16 bit dato in input, essi devono essere immessi mediante switch, 8 bit alla volta, sviluppando un'apposita “rete di controllo” per l’acquisizione che utilizzi due bottoni della board per caricare rispettivamente la prima e la seconda metà del dato in ingresso.

1.1.10 Sintesi su board di sviluppo

Per l’implementazione della rete di interconnessione su board abbiamo aggiunto, rispetto all’esercizio precedente, un’ulteriore entità che preleva gli input forniti tramite board e l’abbiamo chiamata CONTROL_UNIT.

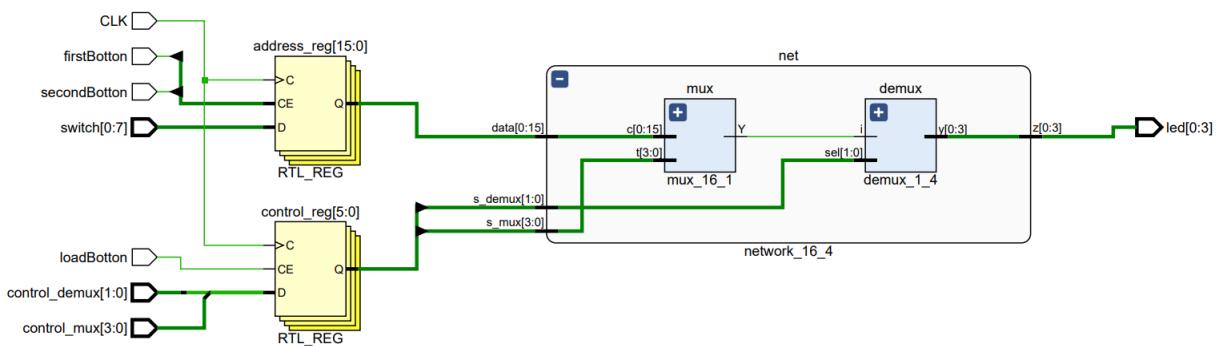


Figura 1.10: Schematic del sistema complessivo

Il codice dell’entità mostrata in Figura 1.10 è il seguente:

```

ENTITY control_unit IS
  PORT (
    firstButton: in std_logic;
    secondButton : in std_logic;
    loadButton: in std_logic;
    switch: in std_logic_vector(0 to 7);
    control_mux : in std_logic_vector(3 downto 0);
    control_demux : in std_logic_vector(1 downto 0);
    CLK: in std_logic;

    led : out std_logic_vector(0 to 3)
  );
END control_unit;

ARCHITECTURE arch OF control_unit IS
COMPONENT network_16_4 IS
  PORT (
    data : IN STD_LOGIC_VECTOR(0 TO 15);
    s_mux : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    s_demux : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    z : OUT STD_LOGIC_VECTOR(0 TO 3)
  );
END COMPONENT;
signal address: std_logic_vector(0 to 15);
signal control: std_logic_vector(5 downto 0);

BEGIN
  net: network_16_4

```

```

PORT MAP (
    data => address,
    s_mux => control(5 downto 2),
    s_demux => control(1 downto 0),
    z => led
);

process(CLK)
BEGIN
    if(rising_edge(CLK)) then
        if(firstButton = '1') then
            address(0 to 7) <= switch;
        end if;

        if(secondButton = '1') then
            address(8 to 15) <= switch;
        end if;

        if(loadButton = '1') then
            control(1 downto 0) <= control_demux;
            control(5 downto 2) <= control_mux;
        end if;
    end if;
end process;
END ARCHITECTURE;

```

Le porte utilizzate sono:

- *firstButton*: con questo bottone carichiamo i valori di input pas-

sati tramite gli switch *switch* nella prima parte dell’indirizzo *address*;

- *secondButton*: con questo bottone carichiamo i valori di input passati tramite switch nella seconda parte dell’indirizzo *address*;
- *loadButton*: con questo bottone carichiamo i valori di input passati tramite gli switch *control_mux* e *control_demux* nel segnale di appoggio *control* associato alle porte *s_mux* e *s_demux* del componente NETWORK_16_4;
- *switch*: con queste porte preleviamo 8 bit dagli switch della board che verranno assegnati nella prima/seconda parte del segnale *address*;
- *control_mux*: con queste porte preleviamo 4 bit dagli switch della board che verranno assegnati ai 4 bit più significativi del segnale *control*;
- *control_demux*: con queste porte preleviamo 2 bit dagli switch della board che verranno assegnati ai 2 bit meno significativi del segnale *control*;
- *CLK*: con questa porta acquisiamo il clock della board;
- *led*: con queste porte mostriamo l’uscita finale del sistema.

Le porte sono state mappate attraverso i seguenti constraint:

```

## Clock signal

set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 }
[get_ports { CLK }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz

##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_EMCCLK_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]

set_property -dict { PACKAGE_PIN T8       IOSTANDARD LVCMOS18 }
[get_ports { control_mux[0] }]; #IO_L24N_T3_34 Sch=sw[8]

set_property -dict { PACKAGE_PIN U8       IOSTANDARD LVCMOS18 }
[get_ports { control_mux[1] }]; #IO_25_34 Sch=sw[9]

set_property -dict { PACKAGE_PIN R16      IOSTANDARD LVCMOS33 }
[get_ports { control_mux[2] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]

set_property -dict { PACKAGE_PIN T13      IOSTANDARD LVCMOS33 }

```

```

[get_ports { control_mux[3] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6      IOSTANDARD LVCMOS33 }

[get_ports { control_demux[0] }]; #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12     IOSTANDARD LVCMOS33 }

[get_ports { control_demux[1] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]

## 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]

##Buttons
set_property -dict { PACKAGE_PIN N17     IOSTANDARD LVCMOS33 }
[get_ports { loadButton }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN P17     IOSTANDARD LVCMOS33 }
[get_ports { firstButton }]; #IO_L12P_T1_MRCC_14 Sch=btln1
set_property -dict { PACKAGE_PIN M17     IOSTANDARD LVCMOS33 }
[get_ports { secondButton }]; #IO_L10N_T1_D15_14 Sch=btnr

```

1.2 Esercizio 2: Sistema ROM+M

Una **ROM** (Read-Only Memory), come suggerisce il nome, è una memoria di sola lettura ed è un componente fondamentale nell'architettura.

ra dei sistemi digitali, svolgendo un ruolo cruciale nella conservazione di dati e istruzioni. Si tratta di un tipo di memoria non volatile in cui le informazioni sono pre-scrivibili durante la fase di produzione e generalmente non modificabili dall'utente finale, dunque adatta per esempio a conservare i firmware di vari dispositivi elettronici, come BIOS o micro-controllori. A seconda delle dimensioni, una ROM è dotata di un certo numero di locazioni di memoria che possono conservare ognuna un certo numero di bit; per accedere ai dati memorizzati si utilizza l'indirizzo delle corrispondenti locazioni di memoria che li mantengono.

1.2.1 Esercizio 2.1

*Progettare, implementare in VHDL e testare mediante simulazione un sistema **S** composto da una **ROM** puramente combinatoria di 16 locazioni da 8 bit ciascuna e da una macchina combinatoria **M** che opera come segue: fornito al sistema un indirizzo **A** di 4 bit, il sistema restituisce il valore contenuto nella ROM all'indirizzo **A** opportunamente “trasformato” attraverso la macchina **M**. Il comportamento della macchina **M** è totalmente a scelta dello studente, l'unico vincolo è che essa prenda in ingresso 8 bit e ne fornisca in uscita 4.*

1.2.2 Progetto e architettura

Poiché il sistema richiesto prevede l’interazione di due componenti, abbiamo innanzitutto realizzato singolarmente la memoria ROM e poi la macchina combinatoria M, per poi connetterle opportunamente.

La **ROM** da noi realizzata, come è possibile osservare in Figura 1.11 ha un’interfaccia che prevede un segnale di ingresso di 4 bit, cioè l’indirizzo di una delle $2^4 = 16$ locazioni di memoria a cui si vuole accedere in lettura, e un segnale di uscita di 8 bit, contenente il valore letto in quella posizione. Dal punto di vista architettonale abbiamo pensato di gestire le locazioni come un vettore di 16 elementi di dimensione 8 bit ciascuno, come richiesto dalla traccia, e abbiamo definito un’architettura di tipo *behavioral* per la nostra ROM per accedere ai valori memorizzati.

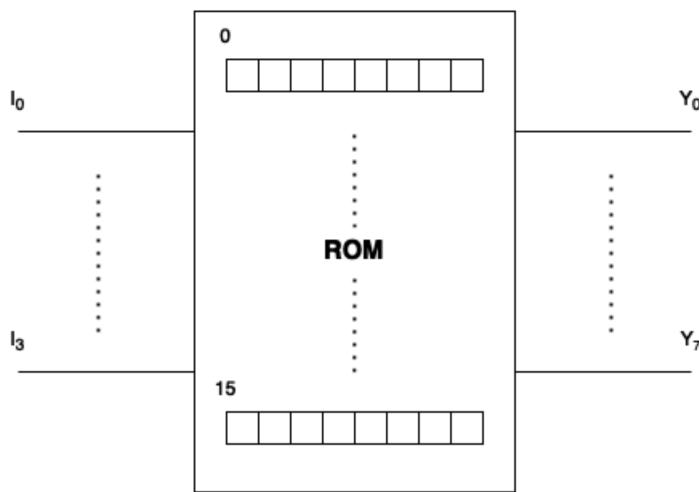


Figura 1.11: Interfaccia ROM

La macchina combinatoria **M** prende in ingresso l’uscita a 8 bit della ROM e, a seconda del valore del segnale di abilitazione, fornisce

un'uscita su 4 bit che coincide con l'uscita del sistema complessivo. Per garantire una coerenza tra dato in memoria e dato in uscita, abbiamo pensato di realizzare M in modo tale che, in presenza di un segnale di selezione alto, vengano posti in uscita i 4 bit meno significativi del segnale in ingresso: in tal modo, passando dal binario al decimale, il valore espresso su 4 bit in uscita dal sistema S coincide con il valore rappresentato su 8 bit che è stato letto in ROM e poi trasformato dalla macchina M in quella determinata uscita. Complessivamente, il sistema **S** risulta realizzato come in Figura 1.12.

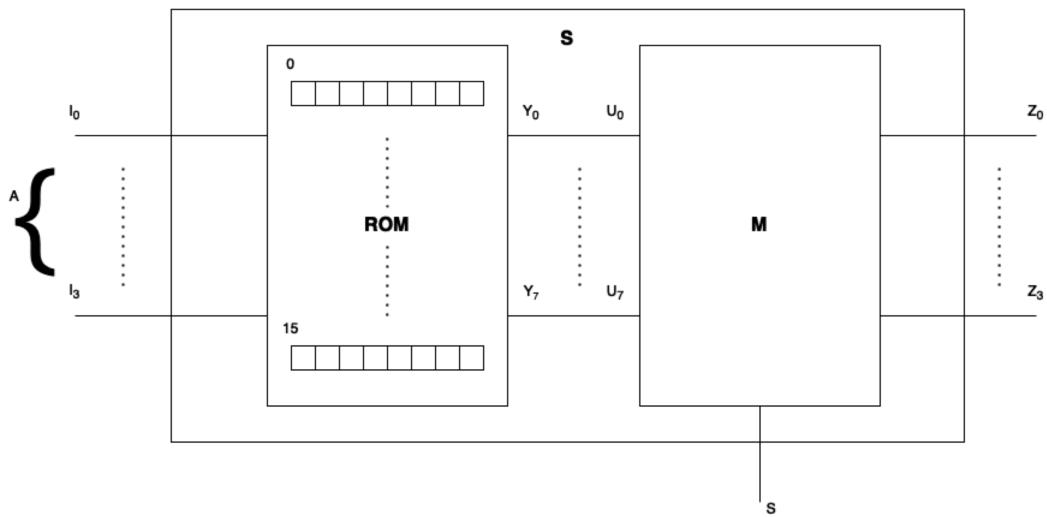


Figura 1.12: Sistema S

1.2.3 Implementazione

L'entity della **ROM** definisce il segnale di 4 bit *address* in ingresso, contenente l'indirizzo A della locazione a cui si vuole accedere, e il segnale di 8 bit *dout* in uscita, che rappresenta il dato letto dalla

ROM. Nell'architecture abbiamo definito innanzitutto il vettore tipo che astrae le locazioni di memoria, per poi istanziarne uno contenente i numeri da 0 a 15; essendo un'architettura behavioral, abbiamo definito inoltre la procedura *get_data* che, dato un indirizzo in ingresso, restituisce il valore contenuto nella locazione identificata da quell'indirizzo.

```

ENTITY ROM IS
    PORT (
        address : IN STD_LOGIC_VECTOR(0 TO 3);
        dout : OUT STD_LOGIC_VECTOR(0 TO 7)
    );
END ENTITY ROM;

ARCHITECTURE RTL OF ROM IS
    TYPE MEMORY_16_8 IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(0 TO 7);
    CONSTANT ROM_16_8 : MEMORY_16_8 := (
        x"00", x"01", x"02", x"03", x"04", x"05", x"06", x"07",
        x"08", x"09", x"0A", x"0B", x"0C", x"0D", x"0E", x"0F"
    );
BEGIN
    get_data : PROCESS (address)
        BEGIN
            dout <= ROM_16_8(to_integer(unsigned(address)));
        END PROCESS get_data;
END ARCHITECTURE RTL;

```

Viceversa, l'entity della macchina combinatoria M presenta in ingresso il valore di 8 bit letto in memoria *data_in* e in uscita il corrispondente valore trasformato su 4 bit *data_out*; inoltre in ingresso riceve anche un segnale di selezione *sel*. L'architecture di M è anch'essa di tipo behavioral, al suo interno abbiamo inserito la procedura *convert_16_to_4*, grazie alla quale l'uscita conterrà proprio il valore letto in memoria ma espresso su 4 bit quando il segnale di selezione è alto.

```
ENTITY M IS
    PORT (
        data_in : IN STD_LOGIC_VECTOR(0 TO 7);
        sel : IN STD_LOGIC;
        data_out : OUT STD_LOGIC_VECTOR(0 TO 3)
    );
END M;
```

```
ARCHITECTURE behavioral OF M IS

BEGIN
    convert_16_to_4 : process(data_in)
    begin
        if sel = '1' then
            data_out <= data_in(4) & data_in(5)
                & data_in(6) & data_in(7);
        else
            data_out <= data_in(0) & data_in(1)
                & data_in(2) & data_in(3);
```

```

        end if;
    end process;
END behavioral;
```

Il sistema **S** è stato realizzato di conseguenza per composizione, utilizzando i due componenti visti; la sua interfaccia, definita nell'entity, prevede:

- *address_in*: l'indirizzo A della locazione a cui accedere;
- *selection*: segnale di selezione, necessario per regolare il funzionamento di M;
- *data_output*: segnale di uscita.

L'architecture è di tipo structural, per cui abbiamo definito innanzitutto i componenti necessari e poi i collegamenti tra le diverse linee di ingresso e di uscita; a tal fine, sfruttiamo il segnale di appoggio *rom_data* per stabilire il collegamento tra output della ROM e input di M.

```

ENTITY S IS
    PORT (
        address_in : IN STD_LOGIC_VECTOR(0 TO 3);
        selection : IN STD_LOGIC;
        data_output : OUT STD_LOGIC_VECTOR(0 TO 3)
    );
END S;
```

CAPITOLO 1. RETI COMBINATORIE ELEMENTARI

```
ARCHITECTURE structural OF S IS
COMPONENT ROM IS
    PORT (
        address : IN STD_LOGIC_VECTOR(0 TO 3);
        dout : OUT STD_LOGIC_VECTOR(0 TO 7)
    );
END COMPONENT;

COMPONENT M IS
    PORT (
        data_in : IN STD_LOGIC_VECTOR(0 TO 7);
        sel : IN STD_LOGIC;
        data_out : OUT STD_LOGIC_VECTOR(0 TO 3)
    );
END COMPONENT;

SIGNAL rom_data : STD_LOGIC_VECTOR(0 TO 7) := "00000000";

BEGIN
    rom_16_8 : ROM
    PORT MAP (
        address => address_in,
        dout => rom_data
    );

    m_8_4 : M
    PORT MAP (
        data_in => rom_data,
```

```

        sel => selection,
        data_out => data_output
    );
END structural;

```

1.2.4 Simulazione

Abbiamo simulato il funzionamento del sistema S attraverso un test-bench che, dati quattro indirizzi di memoria e il segnale di selezione alto, verifica se l'uscita del sistema coincide con il valore contenuto nelle locazioni di memoria con quegli indirizzi.

```

ENTITY s_tb IS
END s_tb;

ARCHITECTURE tb_arch OF s_tb IS
COMPONENT S IS
PORT (
    address_in : IN STD_LOGIC_VECTOR(0 TO 3);
    selection : IN STD_LOGIC;
    data_output : OUT STD_LOGIC_VECTOR(0 TO 3)
);
END COMPONENT;

SIGNAL address_tb : STD_LOGIC_VECTOR(0 TO 3) := "0000";
SIGNAL selection_tb : STD_LOGIC := '0';
SIGNAL data_output_tb : STD_LOGIC_VECTOR(0 TO 3);

```

```
BEGIN

    uut : S
    PORT MAP (
        address_in => address_tb,
        selection => selection_tb,
        data_output => data_output_tb
    );

PROCESS
BEGIN
    WAIT FOR 10 ns;

    address_tb <= "0000";
    selection_tb <= '1';
    WAIT FOR 10 ns;
    ASSERT data_output_tb = "0000" REPORT "Test case 1 fallito"

    address_tb <= "0001";
    selection_tb <= '1';
    WAIT FOR 10 ns;
    ASSERT data_output_tb = "0001" REPORT "Test case 2 fallito"

    address_tb <= "0010";
    selection_tb <= '1';
    WAIT FOR 10 ns;
    ASSERT data_output_tb = "0010" REPORT "Test case 3 fallito"

    address_tb <= "0011";
```

```

selection_tb <= '1';

WAIT FOR 10 ns;

ASSERT data_output_tb = "0011" REPORT "Test case 4 fallito"

WAIT;

END PROCESS;

END tb_arch;

```

In Figura 1.13 possiamo osservare l'output della simulazione. Dopo aver alzato il segnale di selezione, ogni 10 nano-secondi avviene una variazione dell'indirizzo in ingresso. Poiché ogni indirizzo contiene il valore numerico corrispondente a quell'indirizzo, possiamo verificare che l'output varia correttamente: l'indirizzo 0 contiene il valore 0 e anche l'uscita è 0; analogamente quando diamo in ingresso l'indirizzo 1, che memorizza il valore 1, il segnale di uscita fornisce 1.

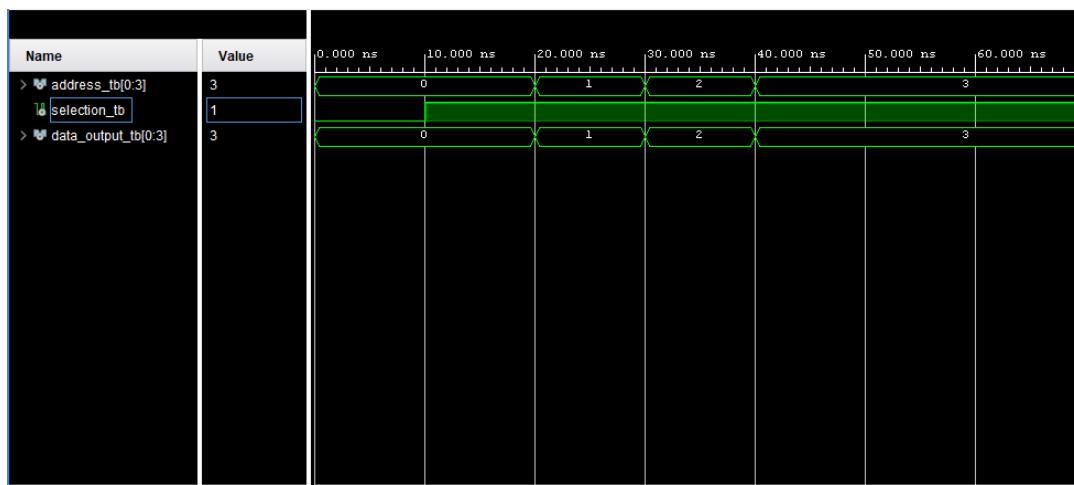


Figura 1.13: Simulazione testbench per il sistema S

1.2.5 Esercizio 2.2

Sintetizzare e implementare su board il progetto del sistema ROM+M sviluppato al punto 2.1, utilizzando gli switch per fornire l'indirizzo della ROM da cui leggere i valori da trasformare e i led per visualizzare i 4 bit di uscita.

1.2.6 Sintesi su board di sviluppo

Per l'implementazione su board della ROM non è stato necessario aggiungere alcun elemento rispetto al sistema dell'esercizio precedente.

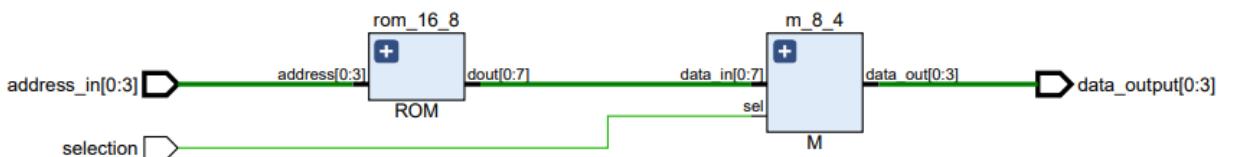


Figura 1.14: Schematic ROM+M su board

E' bastato modificare il file di constraint come segue:

```

##Switches

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

set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 }
[get_ports { address_in[1] }]; #IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]

set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 }
[get_ports { address_in[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]

set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 }
[get_ports { address_in[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
    
```

CAPITOLO 1. RETI COMBINATORIE ELEMENTARI

```
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 }
[get_ports { selection }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]

## LEDs

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

Capitolo 2

Reti sequenziali elementari

2.1 Esercizio 3: Riconoscitore di sequenze

Un **riconoscitore di sequenze** è una macchina sequenziale notevole.

Una **macchina sequenziale** è un sistema la cui uscita in un certo istante non dipende solo dal valore degli ingressi nello stesso istante, ma anche dagli ingressi precedenti. Viene quindi introdotto il concetto di stato, che può anche essere definito come memoria e che rappresenta le informazioni relative all'attività passata della macchina. In questo caso, il concetto di stato ci è utile in quanto, per riconoscere una sequenza di bit, non basta conoscere solo l'ingresso all'istante corrente, ma anche bisogna anche ricordare in qualche modo gli ingressi passati. Una macchina sequenziale possono essere modellata come un **automa**

a stati finiti (ASF), in particolare sono possibili due opzioni: progettare una macchina di *Mealy*, in cui l'uscita è funzione sia dello stato corrente sia dell'ingresso, o una macchina di *Moore*, in cui invece l'uscita è funzione solo dello stato corrente, dunque non c'è dipendenza dell'uscita rispetto all'ingresso corrente.

2.1.1 Esercizio 3.1

*Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza **101**. 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 3 (sequenze non sovrapposte);
- 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 (sequenze parzialmente sovrapposte).

2.1.2 Progetto e architettura

Innanzitutto abbiamo deciso di progettare il riconoscitore come un *ASF di Mealy*, per cui abbiamo realizzato il grafo degli stati e delle

transizioni sia per il caso di riconoscimento senza sovrapposizione sia per quello di riconoscimento con sovrapposizione parziale. Nel primo caso, come possiamo vedere in Figura 2.1, il riconoscitore analizza la sequenza in ingresso considerando gruppi di tre bit alla volta, dunque l’evoluzione del riconoscitore procede sempre verso uno stato successivo diverso da quello corrente: la sequenza 101 viene riconosciuta solo se inizia dopo la fine della sequenza di 3 bit precedente (e chiaramente se corrisponde ai primi 3 bit della stringa in ingresso).

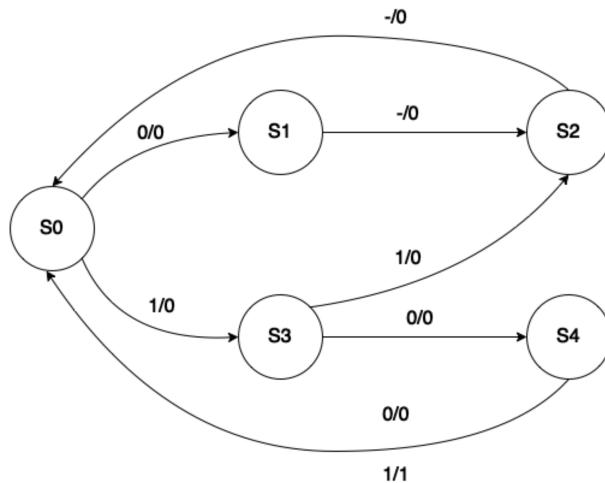


Figura 2.1: Riconoscitore di sequenze non sovrapposte

Nel secondo caso invece, come possiamo osservare in Figura 2.2, da 5 stati si passa a 3 in quanto per S0 e S1 è contemplata la permanenza nello stato corrente in presenza di determinati ingressi. In tal modo, la sequenza 101 può essere riconosciuta anche se inizia prima della fine della sequenza di 3 bit precedente, ma solo fino a un certo punto perché una volta riconosciuta si torna obbligatoriamente nello stato iniziale S0 e si riparte da zero.

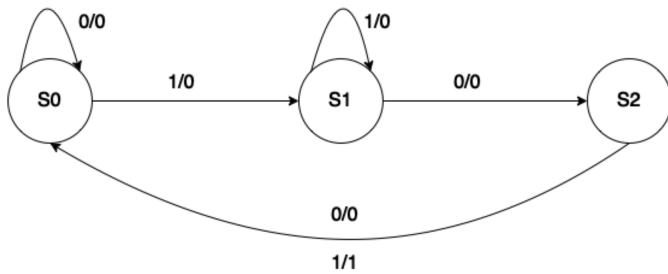


Figura 2.2: Riconoscitore di sequenze parzialmente sovrapposte

Dopo aver modellato l'automa attraverso i grafi necessari, abbiamo pensato di progettare il riconoscitore descrivendolo secondo un approccio *behavioral*, in particolare abbiamo definito due comportamenti diversi:

- **macchina combinatoria:** si occupa di determinare le transizioni di stato e le uscite sulla base del modo selezionato, dello stato attuale e dell'ingresso applicato;
- **memoria:** si occupa di aggiornare lo stato e l'uscita in corrispondenza del fronte di salita (*rising edge*) del segnale di *clock*.

Il segnale di **clock** è un segnale di tempificazione che regola la corretta evoluzione della macchina e la sua presenza è fondamentale in quanto il riconoscitore è una macchina **sincrona**, dunque c'è bisogno di questo segnale per garantire che non avvengano transizioni indesiderate e che si giunga in stati di funzionamento non corretti, dato che altrimenti l'evoluzione della macchina continuerebbe fintanto che è applicato l'ingresso.

2.1.3 Implementazione

Dal punto di vista dell’interfaccia, nell’entity del riconoscitore abbiamo dichiarato i seguenti segnali:

- i : segnale ingresso dati, qui trasmettiamo in input la sequenza in maniera seriale;
- A : segnale di clock;
- RST : segnale di reset;
- M : segnale di modo (0 per sequenze non sovrapposte, 1 per sequenze parzialmente sovrapposte);
- Y : segnale di uscita, alto se è stata riconosciuta una sequenza 101, basso altrimenti.

Dopodiché, all’interno dell’architecture, che come anticipato è di tipo *behavioral*, abbiamo definito il comportamento interno della macchina, dichiarando innanzitutto gli stati di funzionamento e i segnali di appoggio per mantenere stato corrente, stato prossimo e uscita per ogni transizione; tali segnali sono necessari in quanto, come abbiamo visto, stato e uscita non possono essere aggiornati immediatamente ma solo in corrispondenza di un fronte di salita del clock, in tal modo l’uscita è sincrona col clock. La macchina combinatoria e la memoria che compongono il riconoscitore sono descritti utilizzando rispettivamente i *processes*:

- *f_stato_uscita*: process sensibile alle variazioni dei segnali di ingresso dati, modo e stato attuale, sulla base di questi segnali determina tutte le possibili transizioni di stato e le corrispondenti uscite;
- *memoria_stato*: process sensibile alle variazioni del clock e del modo, si occupa di sincronizzare lo stato corrente e l'uscita effettiva della macchina sulla base dell'ultima transizione avvenuta prima del fronte di salita del clock, oppure di resettare lo stato corrente e uscita quando richiesto (segnale di reset alto o variazione del segnale di modo).

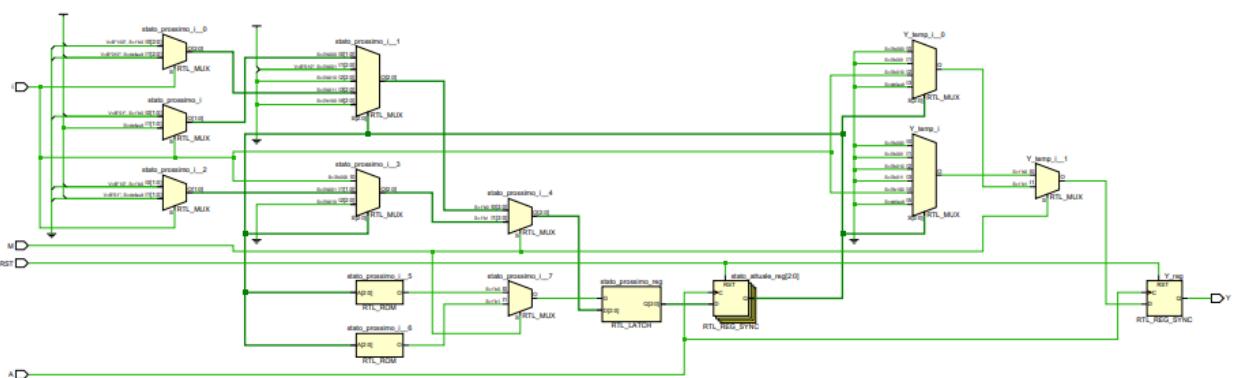


Figura 2.3: Schematic riconoscitore

Di seguito l'implementazione in VHDL del riconoscitore secondo quanto detto.

```

ENTITY riconoscitore IS
    PORT (
        i : IN STD_LOGIC;
        A : IN STD_LOGIC;

```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
RST : IN STD_LOGIC;  
M : IN STD_LOGIC;  
Y : OUT STD_LOGIC  
);  
END riconoscitore;  
  
ARCHITECTURE behavioral OF riconoscitore IS  
    TYPE stato IS (S0, S1, S2, S3, S4);  
  
    SIGNAL stato_attuale : stato := S0;  
    SIGNAL stato_prossimo : stato;  
  
    SIGNAL Y_temp : STD_LOGIC;  
  
BEGIN  
    f_stato_uscita : PROCESS (i, stato_attuale, M)  
    BEGIN  
        CASE M IS  
            WHEN '0' =>  
                CASE stato_attuale IS  
                    WHEN S0 =>  
                        IF (i = '0') THEN  
                            stato_prossimo <= S1;  
                            Y_temp <= '0';  
                        ELSE  
                            stato_prossimo <= S3;  
                            Y_temp <= '0';  
                        END IF;  
                END CASE;  
            WHEN others =>  
                stato_prossimo <= S4;  
                Y_temp <= '1';  
            END CASE;  
        END IF;  
    END;  
END;  
END riconoscitore;
```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
WHEN S1 =>
    stato_prossimo <= S2;
    Y_temp <= '0';

WHEN S2 =>
    stato_prossimo <= S0;
    Y_temp <= '0';

WHEN S3 =>
    IF (i = '0') THEN
        stato_prossimo <= S4;
        Y_temp <= '0';
    ELSE
        stato_prossimo <= S2;
        Y_temp <= '0';
    END IF;

WHEN S4 =>
    stato_prossimo <= S0;
    Y_temp <= i;

WHEN OTHERS =>
    Y_temp <= '0';

END CASE;

WHEN '1' =>
    CASE stato_attuale IS
        WHEN S0 =>
            IF (i = '0') THEN
                stato_prossimo <= S0;
                Y_temp <= '0';
            ELSE
```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
        stato_prossimo <= S1;
        Y_temp <= '0';
    END IF;

WHEN S1 =>
    IF (i = '0') THEN
        stato_prossimo <= S2;
        Y_temp <= '0';
    ELSE
        stato_prossimo <= S1;
        Y_temp <= '0';
    END IF;

WHEN S2 =>
    stato_prossimo <= S0;
    Y_temp <= i;
WHEN OTHERS =>
    Y_temp <= '0';
END CASE;

WHEN OTHERS =>
    Y_temp <= '0';
END CASE;

END PROCESS;

memoria_stato : PROCESS (A)
BEGIN
    IF (rising_edge(A)) THEN
        IF (RST = '1') THEN
            stato_attuale <= S0;
```

```

        Y <= '0';

    ELSE
        stato_attuale <= stato_prossimo;
        Y <= Y_temp;
    END IF;

END IF;

END PROCESS;

END behavioral;

```

2.1.4 Simulazione

Il testbench da noi realizzato per il riconoscitore di sequenze è stato strutturato in due parti, affinché potessimo verificare il corretto comportamento della macchina in entrambe le modalità; la sequenza di ingresso è la stessa per entrambi i modi, ciò dunque ci permette anche di visualizzare quali sono le diverse uscite a seconda dei diversi modi di funzionamento della macchina.

```

ENTITY riconoscitore_tb IS
END riconoscitore_tb;

ARCHITECTURE bench OF riconoscitore_tb IS
CONSTANT clock_period : TIME := 10 ns;

COMPONENT riconoscitore
PORT (

```

```

        i : IN STD_LOGIC;
        A : IN STD_LOGIC;
        RST : IN STD_LOGIC;
        M : IN STD_LOGIC;
        Y : OUT STD_LOGIC
    );
END COMPONENT;

SIGNAL i_tb : STD_LOGIC;
SIGNAL A_tb : STD_LOGIC;
SIGNAL RST_tb : STD_LOGIC := '0';
SIGNAL M_tb : STD_LOGIC;
SIGNAL Y_tb : STD_LOGIC;

SIGNAL stop_the_clock : BOOLEAN := false;
SIGNAL SEQUENCE : STD_LOGIC_VECTOR(0 TO 8) := "101110101";

BEGIN
    uut : riconoscitore
    PORT MAP (
        i => i_tb,
        A => A_tb,
        RST => RST_tb,
        M => M_tb,
        Y => Y_tb
    );
    stimulus : PROCESS

```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
BEGIN

    WAIT FOR 10 ns;

    M_tb <= '0';

    FOR k IN 0 TO 8 LOOP
        i_tb <= SEQUENCE(k);
        WAIT FOR 10 ns;

        IF (k = 2 OR k = 8) THEN
            ASSERT Y_tb = '1'
            REPORT "Valore dell'uscita errato" SEVERITY error;
        ELSE
            ASSERT Y_tb = '0'
            REPORT "Valore dell'uscita errato" SEVERITY error;
        END IF;
    END LOOP;

    M_tb <= '1';

    FOR k IN 0 TO 8 LOOP
        i_tb <= SEQUENCE(k);
        WAIT FOR 10 ns;

        IF (k = 2 OR k = 6) THEN
            ASSERT Y_tb = '1'
            REPORT "Valore dell'uscita errato" SEVERITY error;
        ELSE
```

```

        ASSERT Y_tb = '0'
        REPORT "Valore dell'uscita errato" SEVERITY error;
    END IF;
END LOOP;

RST_tb <= '1';

stop_the_clock <= true;
WAIT;
END PROCESS;

clocking : PROCESS
BEGIN
    WHILE NOT stop_the_clock LOOP
        A_tb <= '0', '1' AFTER clock_period / 2;
        WAIT FOR clock_period;
    END LOOP;
    WAIT;
END PROCESS;

END bench;

```

La sequenza testata è **101110101**; vediamo quali sono gli output attesi a seconda del modo di funzionamento:

- quando **M=0** vengono riconosciute le sequenze non sovrapposte, dunque nel nostro caso **101110101**;

- quando $M=1$ vengono riconosciute le sequenze parzialmente sovrapposte, dunque nel nostro caso **101110101**.

In Figura 2.4 possiamo osservare l'output del riconoscitore di sequenze non sovrapposte, in particolare notiamo come l'uscita diventi alta in corrispondenza dei fronti di salita del clock appena successivi al terzo e al nono bit della sequenza in ingresso.

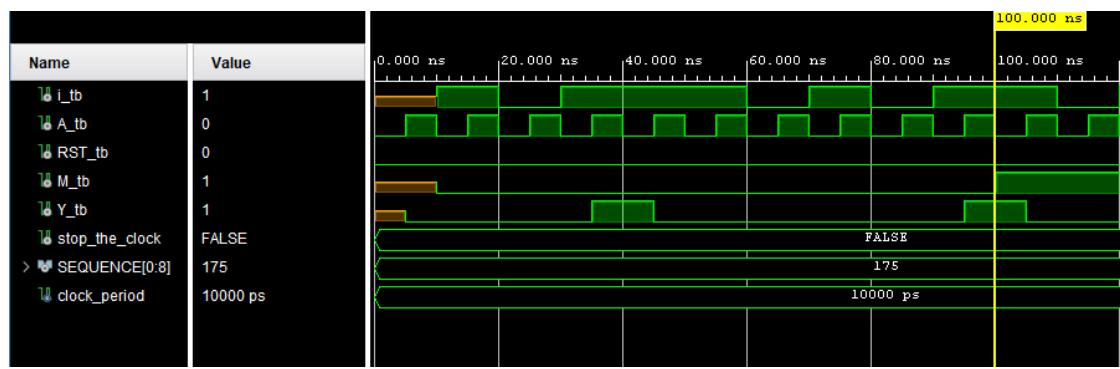


Figura 2.4: Simulazione testbench per il riconoscitore di sequenze non sovrapposte

In Figura 2.4 invece mostriamo l'output del riconoscitore di sequenze parzialmente sovrapposte sottoposto alla stessa sequenza di ingresso: questa volta l'uscita diventa alta in corrispondenza dei fronti di salita del clock successivi al terzo e al settimo bit.

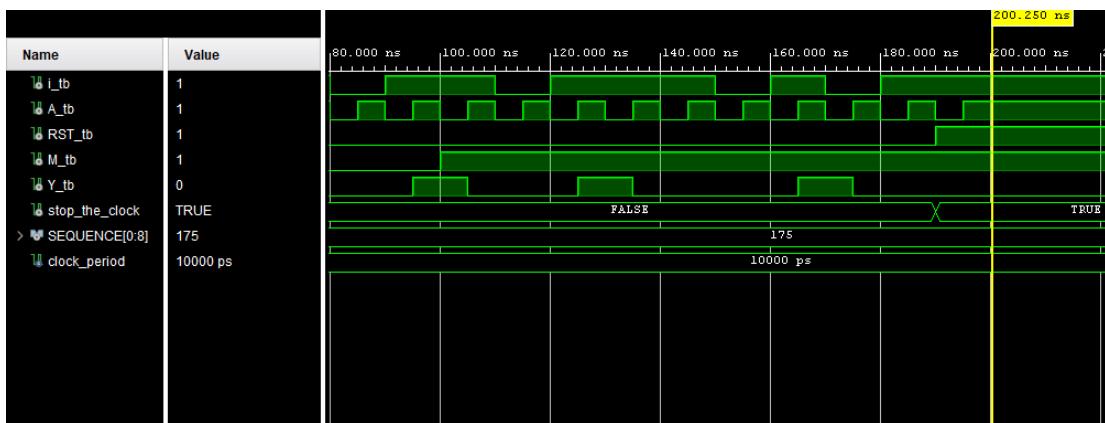


Figura 2.5: Simulazione testbench per il riconoscitore di sequenze parzialmente sovrapposte

2.1.5 Esercizio 3.2

Sintetizzare e implementare su board la rete sviluppata al punto precedente, utilizzando uno switch S_1 per codificare l'input i e uno switch S_2 per codificare il modo M , in combinazione con due bottoni B_1 e B_2 utilizzati rispettivamente per acquisire l'input da S_1 e S_2 in sincronismo con il segnale di temporizzazione A , che deve essere ottenuto a partire dal clock della board. Infine, l'uscita Y può essere codificata utilizzando un led.

2.1.6 Sintesi su board di sviluppo

Basandoci sul componente sviluppato nell'esercizio precedente, abbiamo introdotto una nuova entità denominata CONTROL_UNIT, in cui dichiariamo e istanziamo due DEBOUNCER, uno per il bottone B_1 e uno per il bottone B_2 .

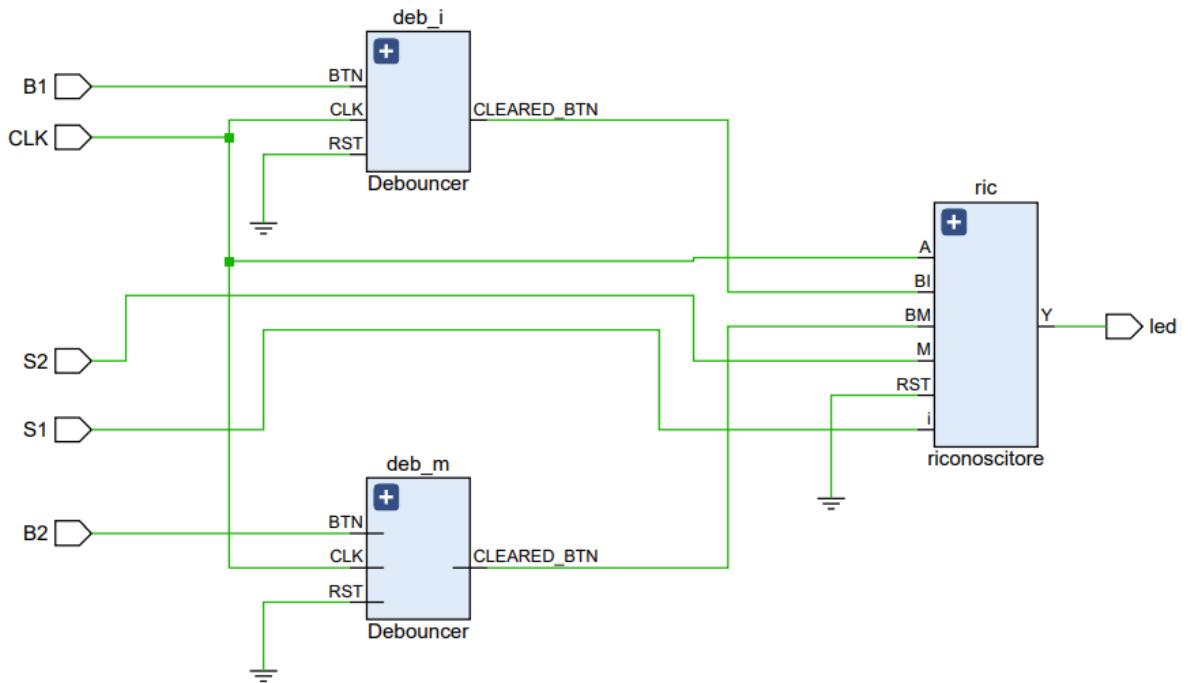


Figura 2.6: Schematic riconoscitore su board

A seguire troviamo l'implementazione della CONTROL_UNIT.

```
ENTITY control_unit IS
  PORT (
    B1 : IN STD_LOGIC;
    B2 : IN STD_LOGIC;
    S1 : IN STD_LOGIC;
    S2 : IN STD_LOGIC;
    CLK : IN STD_LOGIC;
    led: OUT STD_LOGIC
  );
END control_unit;
```

```
ARCHITECTURE Behavioral OF control_unit IS
```

```

COMPONENT Debouncer IS
    GENERIC (
        CLK_period: integer := 10;
        btn_noise_time: integer := 10000000
    );
    PORT ( RST : in STD_LOGIC;
            CLK : in STD_LOGIC;
            BTN : in STD_LOGIC;
            CLEARED_BTN : out STD_LOGIC);
end COMPONENT;

COMPONENT riconoscitore IS
    PORT (
        i : IN STD_LOGIC;
        BI : IN STD_LOGIC;
        BM : IN STD_LOGIC;
        A : IN STD_LOGIC;
        RST : IN STD_LOGIC;
        M : IN STD_LOGIC;
        Y : OUT STD_LOGIC
    );
END COMPONENT;

SIGNAL cleared_i : STD_LOGIC ;
SIGNAL cleared_m : STD_LOGIC;

BEGIN

```

```
deb_i : Debouncer
PORT MAP (
    RST => '0',
    CLK => CLK,
    BTN => B1,
    CLEARED_BTN => cleared_i
);

deb_m : Debouncer
PORT MAP (
    RST => '0',
    CLK => CLK,
    BTN => B2,
    CLEARED_BTN => cleared_m
);

ric: riconoscitore
PORT MAP (
    i => S1,
    BI => cleared_i,
    BM => cleared_m,
    M => S2,
    A => CLK,
    RST => '0',
    Y => led
);
END Behavioral;
```

Le porte utilizzate sono:

- $B1$: con questo bottone carichiamo il valore in input e lo associamo alla porta BTN del componente deb_i la cui uscita $cleared_i$ sarà assegnata alla porta BI del RICONOSCITORE;
- $B2$: con questo bottone carichiamo il valore in input e lo associamo alla porta BTN del componente deb_m la cui uscita $cleared_m$ sarà assegnata alla porta BM del RICONOSCITORE;
- $S1$: con questa porta preleviamo il valore dello switch e lo assegnamo nel segnale i del componente RICONOSCITORE;
- $S2$: con questa porta preleviamo il valore dello switch e lo assegnamo nel segnale M del componente RICONOSCITORE;
- CLK : con questa porta acquisiamo il clock della board;
- led : con questa porta mostriamo l'uscita finale del sistema.

Inoltre, abbiamo apportato una modifica al RICONOSCITORE utilizzato nell'esercizio precedente, introducendo le seguenti porte:

- BI : quando questa porta è alta transitiamo in un nuovo stato dell'automa e assegnamo all'uscita il valore del segnale Y_temp ;
- BM : quando questa porta è alta assegnamo al segnale m_temp il valore del modo M .

Il codice del RICONOSCITORE utilizzato per questo esercizio è il seguente:

```
ENTITY riconoscitore IS
  PORT (
    i : IN STD_LOGIC;
    BI : IN STD_LOGIC;
    BM : IN STD_LOGIC;
    A : IN STD_LOGIC;
    RST : IN STD_LOGIC;
    M : IN STD_LOGIC;
    Y : OUT STD_LOGIC
  );
END riconoscitore;

ARCHITECTURE behavioral OF riconoscitore IS
  TYPE stato IS (S0, S1, S2, S3, S4);

  SIGNAL stato_attuale : stato := S0;
  SIGNAL stato_prossimo : stato;
  SIGNAL m_temp : STD_LOGIC := '0';
  SIGNAL Y_temp : STD_LOGIC;

BEGIN
  f_stato_uscita: process(stato_attuale)
  begin
    --invariato
  end process;
```

```

mem: process (A)
begin
    if (rising_edge(A)) then
        if BI = '1' then
            stato_attuale <= stato_prossimo;
            Y <= Y_temp;
        end if;
        if BM = '1' then
            m_temp <= M;
        end if;
    end if;
end process;
END behavioral;

```

Le porte sono state mappate attraverso i seguenti constraint:

```

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

##Switches
set_property -dict { PACKAGE_PIN J15      IO_STANDARD LVCMOS33 }
[get_ports { S1 }];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16      IO_STANDARD LVCMOS33 }
[get_ports { S2 }];
#IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]

## LEDs

```

```

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

##Buttons

set_property -dict { PACKAGE_PIN P17      IOSTANDARD LVCMOS33 }
[get_ports { B1 }]; #IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17      IOSTANDARD LVCMOS33 }
[get_ports { B2 }]; #IO_L10N_T1_D15_14 Sch=btnr

```

2.1.7 Timing analysis

Come possiamo osservare nel file di constraint:

```

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

```

abbiamo dichiarato un primary clock **CLK** che ha frequenza di 100 MHz in quanto, tramite *period* e *wave-form*, abbiamo dichiarato un clock con periodo di 10 nano-secondi avente fronti d'onda in 0 (salita) e 5 (discesa) nano-secondi; questo clock è collegato al pin E3 della board. Per effettuare la timing analysis abbiamo utilizzato l'apposito tool di report fornito da Vivado; le analisi sono state condotte a valle di sintesi e implementazione per avere delle stime più precise, dato che dopo l'implementazione sono disponibili anche i tempi di routing. Il *path delay type* scelto per il report è *min_max*, in tal modo ci ven-

gono fornite le misurazioni di tre diversi tipi di *slack* (differenza tra required time e arrival time):

- **Worst Negative Slack (WNS):** rientra nell’analisi del ritardo massimo ed è il tempo che impiega un segnale di input a stabilizzarsi prima del fronte successivo del clock, tale che le uscite raggiungano il valore desiderato;
- **Worst Hold Slack (WHS):** rientra nell’analisi del ritardo minimo ed è il tempo per cui un segnale di input deve restare stabile dopo il fronte del clock per consentire all’output di raggiungere il valore desiderato;
- **Worst Pulse Width Slack (WPWS):** ulteriore parametro che fornisce il peggior slack di tutti i controlli temporali visti quando si usano i ritardi min_max.

Abbiamo inoltre impostato al valore 10 i parametri *maximum number of path per clock or path group* e *maximum number of worst paths per endpoints*. I risultati della report per la timing analysis sono mostrati in Figura 2.7; tutti i vincoli temporali specificati sono soddisfatti con il clock utilizzato.

Per capire quale è la frequenza massima di funzionamento possiamo ridurre il periodo del primary clock fino a ottenere un WNS negativo; arrivati a questo punto, se T è il periodo del *clock target* (cioè il periodo per cui abbiamo violato i vincoli temporali) possiamo ottenere la

Design Timing Summary		
Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 5,141 ns	Worst Hold Slack (WHS): 0,193 ns	Worst Pulse Width Slack (WPWS): 4,500 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 200	Total Number of Endpoints: 200	Total Number of Endpoints: 76

All user specified timing constraints are met.

Figura 2.7: Timing analysis riconoscitore di sequenze, periodo del clock di 10 ns

frequenza limite come $FMAX = \frac{1}{T-WNS}$. Nel nostro caso, ciò succede per periodi di clock minori o uguali a 3,7 nano-secondi:

```
create_clock -add -name sys_clk_pin -period 3.70
-waveform {0 1.85} [get_ports { CLK }];
```

L’output dell’analisi di timing è mostrato in Figura 2.8. Il valore della FMAX calcolato secondo la formula vista è circa 264 MHz.

Design Timing Summary		
Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0,087 ns	Worst Hold Slack (WHS): 0,108 ns	Worst Pulse Width Slack (WPWS): 1,350 ns
Total Negative Slack (TNS): -0,102 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 2	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 200	Total Number of Endpoints: 200	Total Number of Endpoints: 76

Timing constraints are not met.

Figura 2.8: Timing analysis riconoscitore di sequenze, periodo del clock di 3,7 ns

2.2 Esercizio 4: Shift register

Lo **shift register** rappresenta un’altra rete sequenziale fondamentale; è composto da una serie di registri (*flip-flop*) che consentono di memo-

rizzare stringhe di bit e, in corrispondenza di un apposito segnale di abilitazione (*shift*), i dati memorizzati al loro interno traslano di una posizione o più, verso sinistra o verso destra a seconda della realizzazione del registro a scorrimento. Questo tipo di rete è alla base di molti dispositivi più complessi, può essere infatti impiegato all'interno di macchine aritmetiche per implementare la moltiplicazione o la divisione per una potenza di due, oppure come contatore con codifica one-hot o, ancora, per la generazione di segnali di controllo periodici nella sua versione circolare, inoltre è alla base di tutte le comunicazioni seriali, in cui è necessario trasmettere un dato, inizialmente memorizzato in parallelo, un bit alla volta.

2.2.1 Esercizio 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. In particolare, i valori possibili di Y sono 1 e 2. L'utente tramite selezione deve scegliere di quante posizioni shiftare.

Il componente deve essere realizzato utilizzando:

- sia un approccio comportamentale;
- sia un approccio strutturale.

Nota: il numero di bit del registro deve essere implementato come un generic, e dall'esterno deve poter essere scelta la modalità di funzionamento mediante opportuni segnali di selezione.

2.2.2 Progetto e architettura

Abbiamo pensato di realizzare lo shift register partendo dall'approccio comportamentale, dunque abbiamo progettato un registro in grado di memorizzare N bit e che preveda in ingresso un segnale di abilitazione, l'ingresso da inserire nello shift register, un ingresso Y, di tipo intero, per indicare se effettuare uno shift di una o due posizioni, un segnale di selezione di un bit (S) e infine il segnale di uscita, anch'esso rappresentato su un bit. Si tratta dunque di un registro di tipo serie-serie. Per rappresentare la memoria abbiamo deciso di utilizzare un segnale di appoggio di N bit. Al fronte di salita del clock, se il segnale di reset è alto, i valori contenuti nello shift register, dunque nella stringa di N bit sudetta, vengono azzerati; quando il segnale di abilitazione si alza, si distinguono quattro casi a seconda della selezione in ingresso e del numero di posizioni da scorrere:

- **S='0', Y=1:** shift verso destra di una posizione, l'ingresso seriale viene posto nella prima posizione della stringa, mentre i valori memorizzati dal primo al penultimo posto avanzano di una posizione, occupando le posizioni dalla seconda all'ultima;

- **S='0', Y=2:** shift verso destra di due posizioni, l'ingresso è memorizzato nella prima e nella seconda posizione della stringa e i bit dalla prima alla terzultima posizione passano nelle posizioni dalla terza all'ultima;
- **S='1', Y=1:** shift verso sinistra di una posizione, l'ingresso viene posto nell'ultima posizione della stringa mentre i valori memorizzati dal secondo all'ultimo posto vanno a occupare le posizioni dalla prima alla penultima;
- **S='1', Y=2:** shift verso sinistra di due posizioni, l'ingresso viene posto nell'ultima e nella penultima posizione della stringa mentre i valori memorizzati dal terzo all'ultimo posto vanno a occupare le posizioni dalla prima alla terzultima.

Successivamente, per realizzare lo stesso registro secondo un approccio strutturale, il nostro primo passo è stato progettare un **flip-flop D**, necessario ai fini della memorizzazione dei bit. Un flip-flop D, schematizzato in Figura 2.9, campiona il valore dell'ingresso e successivamente, in corrispondenza del fronte di salita del clock, lo riproduce in uscita.

Nel nostro shift register vengono istanziati N flip-flop D, uno per ogni bit da memorizzare; inoltre, per realizzare la selezione del comportamento della macchina, che nel caso precedente era regolata da S e Y, abbiamo deciso di utilizzare N multiplexer 4:1 controllati da

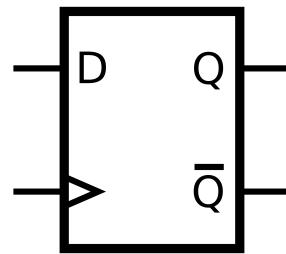


Figura 2.9: Flip-Flop di tipo D

un segnale di selezione S di due bit; tali MUX consentono, oltre l’acquisizione degli ingressi seriali, l’opportuna retroazione delle uscite dei flip-flop, necessaria ai fini dell’implementazione delle diverse modalità di scorrimento del registro, e le loro uscite sono poste in ingresso ai flip-flop. Tenendo presente le quattro modalità di funzionamento, nel caso strutturale il segnale di selezione assume i seguenti valori:

- $S = "00"$: shift a destra di una posizione;
- $S = "01"$: shift a destra di due posizioni;
- $S = "10"$: shift a sinistra di una posizione;
- $S = "11"$: shift a sinistra di due posizioni;

Come uscita del registro osserveremo solo il bit in uscita dall’ultimo flip-flop; anche in tal caso la modalità di realizzazione del registro è serie-serie, in particolare la macchina riceve gli ingressi seriali in corrispondenza del primo e dell’ultimo MUX 4:1.

2.2.3 Implementazione

Per implementare il registro a scorrimento secondo l'approccio behavioral abbiamo dovuto semplicemente realizzare un process, sincronizzato con il fronte di salita del clock, dove abbiamo distinto le quattro casistiche sulla base dei valori di M (segnaletico di modo) e Y (numero di shift da effettuare).

```

ENTITY shift_register_behavioral IS
    GENERIC (
        N : INTEGER := 16
    );
    PORT (
        CLK, RST, SI : IN STD_LOGIC;
        Y : IN INTEGER RANGE 1 TO 2; -- 1 = shift di una posizione;
        DIR : IN STD_LOGIC; -- 0 = shift a destra; 1 = shift a sinistra;
        SO : OUT STD_LOGIC
    );
END shift_register_behavioral;

ARCHITECTURE behavioral OF shift_register_behavioral IS
    SIGNAL tmp : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS => '0');

BEGIN
    PROCESS (CLK)
        BEGIN

```

```

IF (rising_edge(CLK)) THEN
    IF (RST = '1') THEN
        tmp <= (OTHERS => '0');
    ELSE
        CASE DIR IS
            WHEN '0' =>
                IF (Y = 1) THEN
                    tmp(0) <= SI;
                    FOR i IN 1 TO (N - 1) LOOP
                        tmp(i) <= tmp(i - 1);
                    END LOOP;
                ELSIF (Y = 2) THEN
                    tmp(0) <= SI;
                    tmp(1) <= SI;
                    FOR i IN 2 TO (N - 1) LOOP
                        tmp(i) <= tmp(i - 2);
                    END LOOP;
                END IF;

            WHEN '1' =>
                IF (Y = 1) THEN
                    tmp(N - 1) <= SI;
                    FOR i IN 0 TO (N - 2) LOOP
                        tmp(i) <= tmp(i + 1);
                    END LOOP;
                ELSIF (Y = 2) THEN
                    tmp(N - 1) <= SI;
                    tmp(N - 2) <= SI;

```

```

        FOR i IN 0 TO (N - 3) LOOP
            tmp(i) <= tmp(i + 2);
        END LOOP;

        END IF;

WHEN OTHERS =>
    SO <= '0';
END CASE;

END IF;

END IF;

SO <= tmp(N - 1);

END PROCESS;
END behavioral;

```

Nel caso strutturale invece abbiamo innanzitutto realizzato l'architettura del flip-flop D, secondo l'approccio behavioral, in quanto è uno dei due componenti necessari per realizzare il registro; l'altro componente è il MUX 4:1 che abbiamo già realizzato in precedenza (consultabile in Appendice).

```

ENTITY flip_flop_d IS
PORT (
    D, CLK, RST : IN STD_LOGIC;
    Q : OUT STD_LOGIC
);
END flip_flop_d;

```

```
ARCHITECTURE behavioral OF flip_flop_d IS
```

```

BEGIN

    ffd : PROCESS (CLK)
        BEGIN
            IF (rising_edge(CLK)) THEN
                IF (RST = '1') THEN
                    Q <= '0';
                ELSE
                    Q <= D;
                END IF;
            END IF;
        END PROCESS;

END behavioral;

```

Implementato il nostro elemento di memoria, abbiamo realizzato l'architettura di tipo structural dello shift register in modo tale che vengano istanziati $2N$ componenti (N flip-flop, N MUX); l'unica condizione da rispettare è che N sia non minore di 4, dato che i MUX relativi a primo, secondo, penultimo e ultimo flip-flop richiedono degli accorgimenti relativi agli ingressi seriali e dunque sono istanziati a prescindere da N .

```

ENTITY shift_register_structural IS
    GENERIC (
        N : INTEGER := 16
    );

```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
PORT (
    CLK, RST, SI : IN STD_LOGIC;
    SEL : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    SO : OUT STD_LOGIC
);
END shift_register_structural;

ARCHITECTURE structural OF shift_register_structural IS
    SIGNAL temp : STD_LOGIC_VECTOR(0 TO N - 1) := (OTHERS => '0');
    SIGNAL y_temp : STD_LOGIC_VECTOR(0 TO N - 1) := (OTHERS => '0');

COMPONENT mux_4_1 IS
    PORT (
        b : IN STD_LOGIC_VECTOR(0 TO 3);
        r : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        u : OUT STD_LOGIC
    );
END COMPONENT;

COMPONENT flip_flop_d IS
    PORT (
        D, CLK, RST : IN STD_LOGIC;
        Q : OUT STD_LOGIC
    );
END COMPONENT;
```

BEGIN

-- SEL assume i seguenti valori:

-- 00 -> shift right 1
 -- 01 -> shift right 2
 -- 10 -> shift left 1
 -- 11 -> shift left 2

mux_0 : mux_4_1 PORT MAP (

b(0) => SI,
 b(1) => SI,
 b(2) => y_temp(1),
 b(3) => y_temp(2),
 r => SEL,
 u => temp(0)

) ;

mux_1 : mux_4_1 PORT MAP (

b(0) => y_temp(0),
 b(1) => SI,
 b(2) => y_temp(2),
 b(3) => y_temp(3),
 r => SEL,
 u => temp(1)

) ;

mux_N2 : mux_4_1 PORT MAP (

b(0) => y_temp(N - 3),

```

b(1) => y_temp(N - 4),
b(2) => y_temp(N - 1),
b(3) => SI,
r => SEL,
u => temp(N - 2)
);

mux_N1 : mux_4_1 PORT MAP(
b(0) => y_temp(N - 2),
b(1) => y_temp(N - 3),
b(2) => SI,
b(3) => SI,
r => SEL,
u => temp(N - 1)
);

create_mux : FOR i IN 2 TO N - 3 GENERATE
    mux_i : mux_4_1 PORT MAP(
        b(0) => y_temp(i - 1),
        b(1) => y_temp(i - 2),
        b(2) => y_temp(i + 1),
        b(3) => y_temp(i + 2),
        r => SEL,
        u => temp(i)
    );
END GENERATE;

create_ff : FOR i IN 0 TO N - 1 GENERATE

```

```

ff : flip_flop_d PORT MAP (
    D => temp(i),
    CLK => CLK,
    RST => RST,
    Q => y_temp(i)
);

END GENERATE create_ff;

SO <= y_temp(N - 1);

END structural;

```

2.2.4 Simulazione

Le simulazioni sono state effettuate istanziando, sia nel caso behavioral sia nel caso structural, uno shift register di 4 bit, inviando in maniera seriale i bit della stringa 1001101110010001. Di seguito il codice del testbench per lo shift register realizzato secondo l'approccio behavioral, seguito dalla Figura 2.10 e dalla Figura 2.11 dove mostriamo l'output della simulazione.

```

ENTITY shift_register_behavioral_tb IS
END;

ARCHITECTURE bench OF shift_register_behavioral_tb IS

COMPONENT shift_register_behavioral
    GENERIC (

```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
N : INTEGER := 16
);
PORT (
    CLK, RST, SI : IN STD_LOGIC;
    Y : IN INTEGER RANGE 1 TO 2;
    DIR : IN STD_LOGIC;
    SO : OUT STD_LOGIC
);
END COMPONENT;

SIGNAL CLK, RST, SI : STD_LOGIC;
SIGNAL Y : INTEGER RANGE 1 TO 2;
SIGNAL DIR : STD_LOGIC;
SIGNAL SO : STD_LOGIC;

CONSTANT clock_period : TIME := 10 ns;
SIGNAL stop_the_clock : BOOLEAN;

SIGNAL SEQUENCE : STD_LOGIC_VECTOR (15 DOWNTO 0) := "10011011100
BEGIN

uut : shift_register_behavioral GENERIC MAP(N => 4)
PORT MAP(
    CLK => CLK,
    RST => RST,
    SI => SI,
    Y => Y,
```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
DIR => DIR,
SO => SO);

stimulus : PROCESS
BEGIN
    Y <= 1;
    DIR <= '0';

    shift_right_1 : FOR k IN 15 DOWNTO 0 LOOP
        SI <= SEQUENCE(k);
        WAIT FOR 10 ns;
    END LOOP;

    SI <= '-';
    WAIT FOR 30 ns;

    RST <= '1';

    WAIT FOR 20 ns;

    RST <= '0';

    Y <= 2;
    DIR <= '0';

    shift_right_2 : FOR k IN 15 DOWNTO 0 LOOP
        SI <= SEQUENCE(k);
```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
WAIT FOR 10 ns;  
END LOOP;  
  
SI <= '-';  
  
WAIT FOR 30 ns;  
  
RST <= '1';  
  
WAIT FOR 20 ns;  
  
RST <= '0';  
  
Y <= 1;  
DIR <= '1';  
  
shift_left_1 : FOR k IN 15 DOWNTO 0 LOOP  
    SI <= SEQUENCE(k);  
    WAIT FOR 10 ns;  
    END LOOP;  
  
SI <= '-';  
  
WAIT FOR 30 ns;  
  
RST <= '1';  
  
WAIT FOR 20 ns;
```

```
RST <= '0';

Y <= 2;

DIR <= '1';

shift_left_2 : FOR k IN 15 DOWNTO 0 LOOP
    SI <= SEQUENCE(k);
    WAIT FOR 10 ns;
END LOOP;

SI <= '-';

WAIT FOR 30 ns;

RST <= '1';

stop_the_clock <= true;
WAIT;
END PROCESS;

clocking : PROCESS
BEGIN
    WHILE NOT stop_the_clock LOOP
        CLK <= '0', '1' AFTER clock_period / 2;
        WAIT FOR clock_period;
    END LOOP;
    WAIT;
```

```

END PROCESS;

END;

```

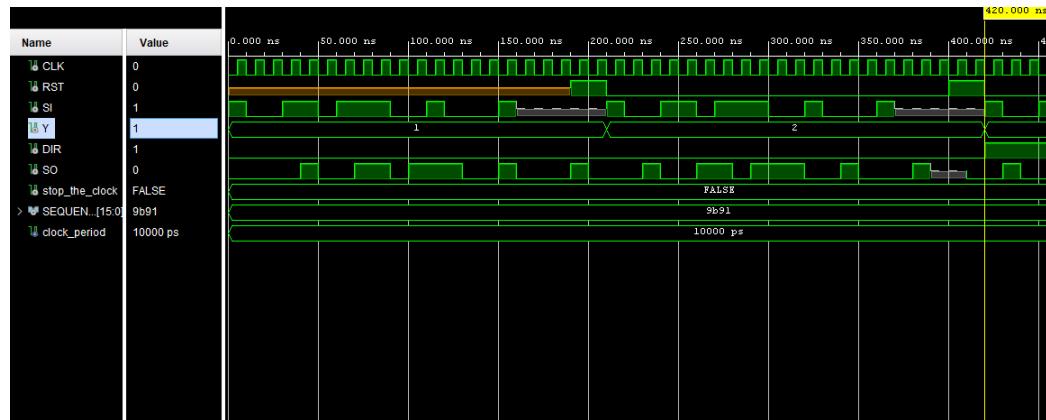


Figura 2.10: Simulazione delle prime due modalità shift register behavioral

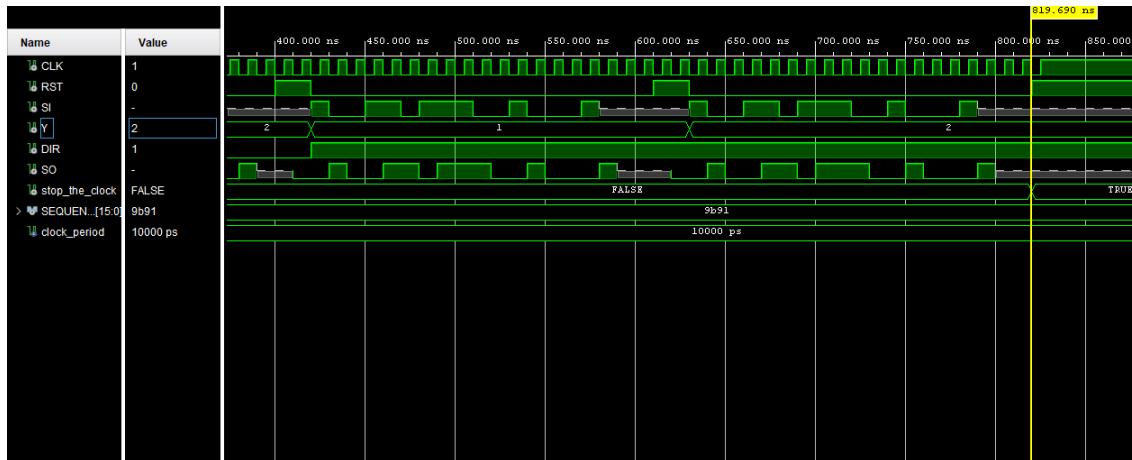


Figura 2.11: Simulazione delle ultime due modalità shift register behavioral

Mostriamo ora il testbench per lo shift register realizzato secondo l'approccio structural e l'output della simulazione in Figura 2.12 e in Figura 2.13.

```

ENTITY shift_register_structural_tb IS
END;

ARCHITECTURE bench OF shift_register_structural_tb IS

COMPONENT shift_register_structural
  GENERIC (
    N : INTEGER := 16
  );
  PORT (
    CLK, RST, SI : IN STD_LOGIC;
    SEL : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    SO : OUT STD_LOGIC
  );
END COMPONENT;

SIGNAL CLK, RST, SI : STD_LOGIC;
SIGNAL SEL : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL SO : STD_LOGIC;

CONSTANT clock_period : TIME := 10 ns;
SIGNAL stop_the_clock : BOOLEAN;

SIGNAL SEQUENCE : STD_LOGIC_VECTOR (15 DOWNTO 0) := "10011011100

```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
BEGIN
```

```
    uut : shift_register_structural GENERIC MAP (N => 4)
```

```
    PORT MAP (
```

```
        CLK => CLK,
```

```
        RST => RST,
```

```
        SI => SI,
```

```
        SEL => SEL,
```

```
        SO => SO);
```

```
stimulus : PROCESS
```

```
BEGIN
```

```
    SEL <= "00";
```

```
    WAIT FOR 10 ns;
```

```
    shift_right_1 : FOR k IN 15 DOWNTO 0 LOOP
```

```
        SI <= SEQUENCE(k);
```

```
        WAIT FOR 10 ns;
```

```
    END LOOP;
```

```
    SI <= '-';
```

```
    WAIT FOR 30 ns;
```

```
    RST <= '1';
```

```
    WAIT FOR 20 ns;
```

```
RST <= '0';

SEL <= "01";

WAIT FOR 10 ns;

shift_right_2 : FOR k IN 15 DOWNTO 0 LOOP
    SI <= SEQUENCE(k);
    WAIT FOR 10 ns;
END LOOP;

SI <= '-';

WAIT FOR 30 ns;

RST <= '1';

WAIT FOR 20 ns;

RST <= '0';

SEL <= "10";

WAIT FOR 10 ns;

shift_left_1 : FOR k IN 15 DOWNTO 0 LOOP
    SI <= SEQUENCE(k);
```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
WAIT FOR 10 ns;  
END LOOP;  
  
SI <= '-';  
  
WAIT FOR 30 ns;  
  
RST <= '1';  
  
WAIT FOR 20 ns;  
  
RST <= '0';  
  
SEL <= "11";  
  
WAIT FOR 10 ns;  
  
shift_left_2 : FOR k IN 15 DOWNTO 0 LOOP  
    SI <= SEQUENCE(k);  
    WAIT FOR 10 ns;  
END LOOP;  
  
SI <= '-';  
  
WAIT FOR 30 ns;  
  
RST <= '1';
```

```

WAIT;

END PROCESS;

clocking : PROCESS
BEGIN
    WHILE NOT stop_the_clock LOOP
        CLK <= '0', '1' AFTER clock_period / 2;
        WAIT FOR clock_period;
    END LOOP;
    WAIT;
END PROCESS;

END;

```

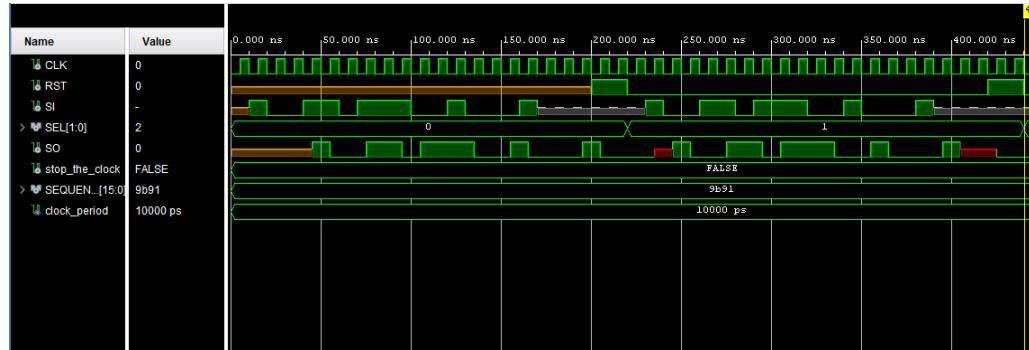


Figura 2.12: Simulazione delle prime due modalità shift register strutturali



Figura 2.13: Simulazione delle ultime due modalità shift register strutturali

2.3 Esercizio 5: Cronometro

2.3.1 Esercizio 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.

2.3.2 Progetto e architettura

Per la progettazione del cronometro abbiamo innanzitutto identificato i componenti fondamentali per realizzare il comportamento desiderato, ovvero i **contatori**. In particolare, sono tre i contatori di cui abbiamo avuto bisogno, uno per i secondi, uno per i minuti e uno per le ore; per i secondi e per i minuti si tratta di contatori modulo 60, mentre per le ore il contatore è modulo 24. Abbiamo deciso di progettare entrambe le tipologie di contatori per riduzione, partendo rispettivamente da contatori modulo 64 e modulo 32, applicando poi opportunamente dei segnali di *reset* quando il conteggio effettuato a 59 (minuti e secondi) o a 23 (ore). Per la realizzazione dei singoli contatori abbiamo deciso di usare un approccio strutturale. Partiremo, quindi, dal componente base dei contatori, ovvero il **flip-flop T**, che si comporta come un contatore modulo 2, per la descrizione e la progettazione dell'oggetto richiesto. Il flip-flop T, schematizzato in Figura 2.14, commuta la sua uscita in maniera sincrona con il fronte di discesa del clock; collegando in serie dei flip-flop T e applicando opportuni segnali di abilitazione e di reset possiamo ottenere contatori di modulo grande a piacere.

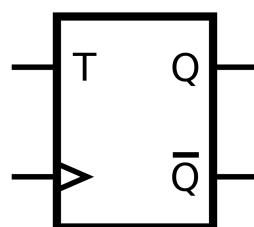


Figura 2.14: Flip-Flop di tipo T

In Figura 2.15 riportiamo lo schema adottato per realizzare un contatore modulo 24 a partire da uno modulo 32. Un generico contatore modulo 32 conta i numeri da 0 a 31; il numero minimo di bit per esprimere i numeri da 0 a 31 è $\log_2 32 = 5$, quindi abbiamo bisogno di 5 flip-flop T posti in cascata. Il nostro contatore è sviluppato secondo un modello parallelo. In questo caso, il primo contatore commuta ogni volta che arriva il clock, il secondo commuta quando arriva il clock e il primo è alto, il terzo invece quando arriva il clock e sia il primo che il secondo sono alti, e così via; a tal fine ci serviamo delle porte AND che prendono in ingresso il segnale di clock e i conteggi in uscita dai flip-flop precedenti. Per ridurre il contatore utilizziamo un apposito segnale di reset che si alza quando il conteggio raggiunge il valore 10111 (23 in binario), in tal modo al successivo colpo di clock si ritorna a 0. Possiamo abilitare il segnale di reset sfruttando una porta AND che raccoglie i valori di conteggio di ogni singolo flip-flop, negando il secondo bit più significativo: in tal modo, raggiunto il valore 10111, la AND in ingresso avrà 11111 e la sua uscita sarà alta.

Analogamente, un contatore modulo 64 conta i numeri da 0 a 63 e per esprimere i numeri da 0 a 63 abbiamo bisogno almeno di $\log_2 64 = 6$ bit; di conseguenza, come possiamo osservare in Figura 2.16, utilizziamo 6 flip-flop T in cascata per realizzare il contatore e un segnale di reset che diventa alto quando il conteggio raggiunge 111011 (59 in binario).

Una volta individuati i contatori, abbiamo dovuto progettare il

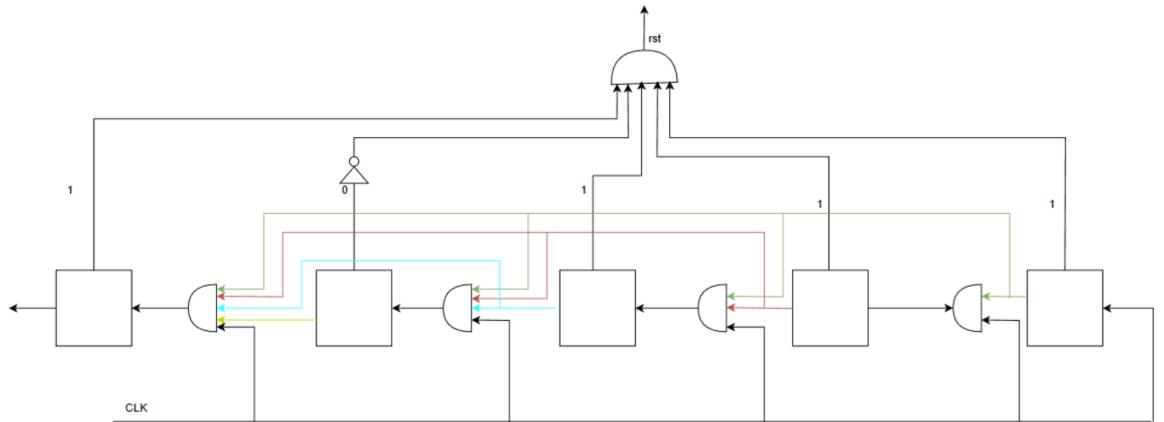


Figura 2.15: Contatore modulo 24

collegamento a cascata tra i tre, in particolare quello dei minuti deve abilitarsi ed effettuare il conteggio in sincronia con il fronte di salita del clock quando termina il conteggio dei secondi (cioè quando il contatore dei secondi si azzerà), analogamente il contatore delle ore deve essere sincrono con il fronte di salita del clock e con l'azzeramento dei contatori relativi a minuti e secondi. I segnali di abilitazione sono dunque realizzati sfruttando delle porte AND che ricevono in ingresso il segnale di clock e le uscite dei contatori precedenti di interesse. Infine, poiché il cronometro deve funzionare a partire da una base dei tempi prefissata, abbiamo scelto quella della board che ha frequenza 100 MHz e abbiamo realizzato un divisore di frequenza che, a partire dal clock della board, fornisce il clock alla frequenza giusta, cioè 1 Hz. Complessivamente, il cronometro progettato può essere schematizzato come in Figura 2.17.

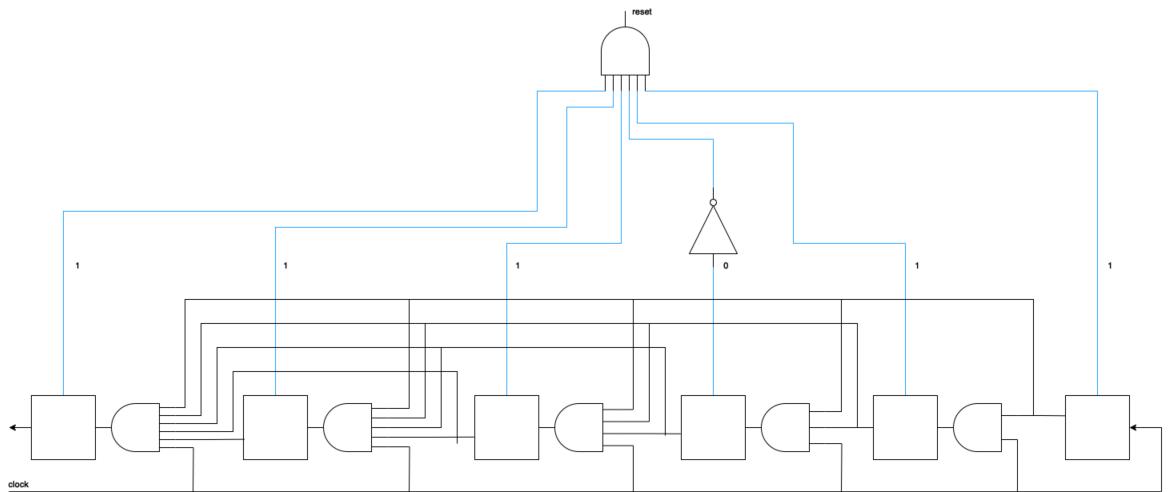


Figura 2.16: Contatore modulo 60

2.3.3 Implementazione

Partiamo dall'implementazione del componente elementare, cioè il Flip-Flop T, che abbiamo realizzato in maniera behavioral.

```

ENTITY flip_flop_t IS
    PORT (
        clk : IN STD_LOGIC;
        rst : IN STD_LOGIC;
        en : IN STD_LOGIC;
        rst_count : IN STD_LOGIC;
        load : IN STD_LOGIC;
        set : IN STD_LOGIC;
        count : OUT STD_LOGIC
    );
END ENTITY flip_flop_t;

```

```

ARCHITECTURE behavioral OF flip_flop_t IS

```

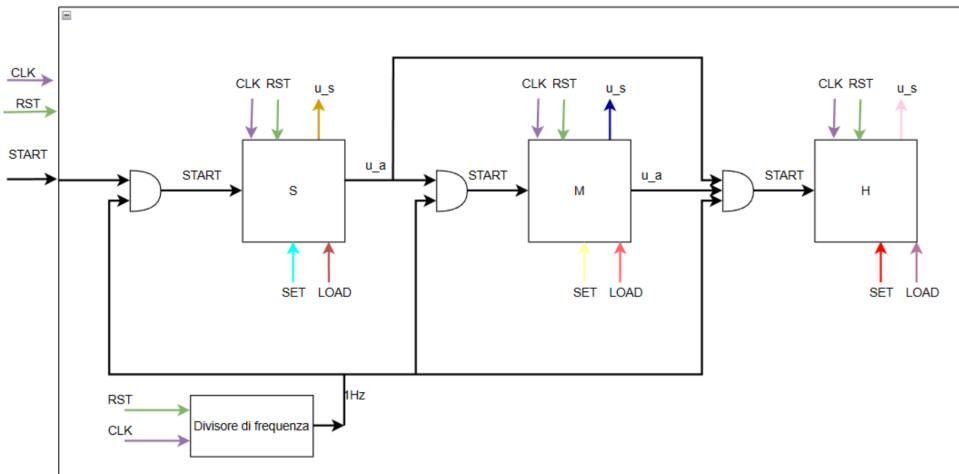


Figura 2.17: Schema del cronometro

```
SIGNAL counter_value : STD_LOGIC := '0';
```

```
BEGIN
    fft : PROCESS (clk)
    BEGIN
        IF (falling_edge(clk)) THEN
            IF (rst = '1') THEN
                counter_value <= '0';
            ELSE
                IF (load = '1') THEN
                    counter_value <= set;
                ELSE
                    IF (rst_count = '1') THEN
                        counter_value <= '0';
                    ELSE
                        IF (en = '1') THEN
                            counter_value <= NOT counter_value;
                        END IF;
                    END IF;
                END IF;
            END IF;
        END IF;
    END PROCESS;
END;
```

```

        END IF;

        END IF;

        END IF;

        END IF;

END PROCESS;

count <= counter_value;
END behavioral;
```

Dopodiché, per composizione, sfruttando un apposito numero di flip-flop e opportuni segnali di reset, abbiamo realizzato il contatore modulo 24

```

ENTITY counter_mod24 IS
PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    en : IN STD_LOGIC;
    load : IN STD_LOGIC;
    set : IN STD_LOGIC_VECTOR(0 TO 4);
    count : OUT STD_LOGIC_VECTOR(0 TO 4)
);
END counter_mod24;
```

```
ARCHITECTURE structural OF counter_mod24 IS
```

```
COMPONENT flip_flop_t IS
PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
```

```

        en : IN STD_LOGIC;
        rst_count : IN STD_LOGIC;
        load : IN STD_LOGIC;
        set : IN STD_LOGIC;
        count : OUT STD_LOGIC
    );
END COMPONENT flip_flop_t;

SIGNAL counter : STD_LOGIC_VECTOR(4 DOWNTO 0) := (OTHERS => '0');
SIGNAL reset : STD_LOGIC := '0';

BEGIN
    -- 23 in binario = 10111
    reset <= counter(4) AND NOT counter(3) AND counter(2) AND counter(1);

    U0 : flip_flop_t PORT MAP (
        clk => clk,
        rst => rst,
        en => en,
        rst_count => reset AND en,
        load => load,
        set => set(4),
        count => counter(0)
    );

    U1 : flip_flop_t PORT MAP (
        clk => clk,
        rst => rst,

```

```

    en => counter(0) AND en,
    rst_count => reset AND en,
    load => load,
    set => set(3),
    count => counter(1)
);

```

```

U2 : flip_flop_t PORT MAP(
    clk => clk,
    rst => rst,
    en => counter(0) AND counter(1) AND en,
    rst_count => reset AND en,
    load => load,
    set => set(2),
    count => counter(2)
);

```

```

U3 : flip_flop_t PORT MAP(
    clk => clk,
    rst => rst,
    en => counter(0) AND counter(1) AND counter(2) AND en,
    rst_count => reset AND en,
    load => load,
    set => set(1),
    count => counter(3)
);

```

```
U4 : flip_flop_t PORT MAP(
```

```

clk => clk,
rst => rst,
en => counter(0) AND counter(1) AND counter(2) AND counter(3)
rst_count => reset AND en,
load => load,
set => set(0),
count => counter(4)
);

count <= counter;
END structural;

```

e il contatore modulo 60.

```

ENTITY counter_mod60 IS
PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    en : IN STD_LOGIC;
    load : IN STD_LOGIC;
    set : IN STD_LOGIC_VECTOR(0 TO 5);
    count : OUT STD_LOGIC_VECTOR(0 TO 5);
    en_next : OUT STD_LOGIC
);
END counter_mod60;

```

```
ARCHITECTURE structural OF counter_mod60 IS
```

```
COMPONENT flip_flop_t IS
PORT (
```

```

        clk : IN STD_LOGIC;
        rst : IN STD_LOGIC;
        en : IN STD_LOGIC;
        rst_count : IN STD_LOGIC;
        load : IN STD_LOGIC;
        set : IN STD_LOGIC;
        count : OUT STD_LOGIC
    );
END COMPONENT flip_flop_t;

SIGNAL counter : STD_LOGIC_VECTOR(5 DOWNTO 0) := (OTHERS => '0');
SIGNAL reset : STD_LOGIC := '0';

BEGIN
    -- 59 in binario = 111011
    reset <= counter(5) AND counter(4) AND counter(3) AND NOT counter(2);
    en_next <= reset;
    U0 : flip_flop_t PORT MAP (
        clk => clk,
        rst => rst,
        en => en,
        rst_count => reset AND en,
        load => load,
        set => set(5),
        count => counter(0)
    );
    U1 : flip_flop_t PORT MAP (

```

```

clk => clk,
rst => rst,
en => counter(0) AND en,
rst_count => reset AND en,
load => load,
set => set(4),
count => counter(1)

);

```

```

U2 : flip_flop_t PORT MAP (
    clk => clk,
    rst => rst,
    en => counter(0) AND counter(1) AND en,
    rst_count => reset AND en,
    load => load,
    set => set(3),
    count => counter(2)

);

```

```

U3 : flip_flop_t PORT MAP (
    clk => clk,
    rst => rst,
    en => counter(0) AND counter(1) AND counter(2) AND en,
    rst_count => reset AND en,
    load => load,
    set => set(2),
    count => counter(3)

);

```

```

U4 : flip_flop_t PORT MAP (
    clk => clk,
    rst => rst,
    en => counter(0) AND counter(1) AND counter(2) AND counter(3),
    rst_count => reset AND en,
    load => load,
    set => set(1),
    count => counter(4)
);

U5 : flip_flop_t PORT MAP (
    clk => clk,
    rst => rst,
    en => counter(0) AND counter(1) AND counter(2) AND counter(3),
    rst_count => reset AND en,
    load => load,
    set => set(0),
    count => counter(5)
);

count <= counter;
END structural;

```

Inoltre, per ottenere la frequenza di conteggio desiderata, cioè 1 Hz (dunque un conteggio ogni secondo), abbiamo realizzato un divisore di frequenza che può essere regolato a piacere conoscendo la frequenza della base dei tempi prefissata.

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
ENTITY divisore_frequenza IS
  PORT (
    clock_in : IN STD_LOGIC;
    reset : IN STD_LOGIC;
    clock_out : OUT STD_LOGIC
  );
END divisore_frequenza;

ARCHITECTURE Behavioral OF divisore_frequenza IS
  SIGNAL count : unsigned(27 DOWNTO 0) := (OTHERS => '0');
  SIGNAL no_division : BOOLEAN := true;

  -- inserire 99999999 al posto di 9 per passare da 100 MHz a 1 Hz
  CONSTANT divider : unsigned(27 DOWNTO 0) := to_unsigned(9, 28);

BEGIN
  PROCESS (clock_in, reset, no_division)
  BEGIN
    IF no_division = true THEN
      clock_out <= clock_in;
    ELSE
      IF reset = '1' THEN
        count <= (OTHERS => '0');
        clock_out <= '0';
      ELSIF rising_edge(clock_in) THEN
        IF count = divider THEN
          count <= (OTHERS => '0');
          clock_out <= '1';
        END IF;
      END IF;
    END IF;
  END PROCESS;
END;
```

```

        ELSE
            clock_out <= '0';
            count <= count + 1;
        END IF;
    END IF;
END IF;

END PROCESS;

END Behavioral;

```

Una volta realizzati tutti i moduli necessari, abbiamo proceduto con la realizzazione di tipo structural del cronometro. Anche in tal caso sono utilizzati appositi segnali di appoggio per effettuare opportunamente il reset dei singoli contatori istanziati e dunque effettuare il conteggio di ore minuti e secondi in maniera corretta.

```

ENTITY cronometro IS
PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    start_stop : IN STD_LOGIC;
    load_sec : IN STD_LOGIC;
    load_min : IN STD_LOGIC;
    load_hours : IN STD_LOGIC;
    set_sec : IN STD_LOGIC_VECTOR(0 TO 5);
    set_min : IN STD_LOGIC_VECTOR(0 TO 5);
    set_hours : IN STD_LOGIC_VECTOR(0 TO 4);
    seconds : OUT STD_LOGIC_VECTOR(0 TO 5);

```

```

        minutes : OUT STD_LOGIC_VECTOR(0 TO 5);

        hours : OUT STD_LOGIC_VECTOR(0 TO 4)

    );

END ENTITY cronometro;

ARCHITECTURE structural OF cronometro IS

COMPONENT counter_mod24 IS

PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    en : IN STD_LOGIC;
    load : IN STD_LOGIC;
    set : IN STD_LOGIC_VECTOR(0 TO 4);
    count : OUT STD_LOGIC_VECTOR(0 TO 4)
);

END COMPONENT counter_mod24;

COMPONENT counter_mod60 IS

PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    en : IN STD_LOGIC;
    load : IN STD_LOGIC;
    set : IN STD_LOGIC_VECTOR(0 TO 5);
    count : OUT STD_LOGIC_VECTOR(0 TO 5);
    en_next : OUT STD_LOGIC
);

END COMPONENT counter_mod60;

```

```

COMPONENT divisore_frequenza IS
    --GENERIC (
        --freq_in : INTEGER := 10000000;
        --freq_out : INTEGER := 1000000
    --);
    PORT (
        clock_in : IN STD_LOGIC;
        reset : IN STD_LOGIC;
        clock_out : OUT STD_LOGIC
    );
END COMPONENT divisore_frequenza;

SIGNAL new_clock : STD_LOGIC;
SIGNAL en_min : STD_LOGIC := '0';
SIGNAL en_hours : STD_LOGIC := '0';

SIGNAL a1 : STD_LOGIC := '0';
SIGNAL a2 : STD_LOGIC := '0';
SIGNAL a3 : STD_LOGIC := '0';

BEGIN
    divisore_freq : divisore_frequenza
    PORT MAP (
        clock_in => clk,
        reset => rst,
        clock_out => new_clock
    );

```

```

) ;

a1 <= start_stop and new_clock;

counter_sec : counter_mod60
PORT MAP (
    clk => clk,
    rst => rst,
    en => a1,
    load => load_sec,
    set => set_sec,
    count => seconds,
    en_next => en_min
) ;

a2 <= en_min and start_stop and new_clock;

counter_min : counter_mod60
PORT MAP (
    clk => clk,
    rst => rst,
    en => a2,
    load => load_min,
    set => set_min,
    count => minutes,
    en_next => en_hours
) ;

```

```

        a3 <= en_hours and en_min and start_stop and new_clock;

        counter_hours : counter_mod24
PORT MAP (
        clk => clk,
        rst => rst,
        en => a3,
        load => load_hours,
        set => set_hours,
        count => hours
) ;

END structural;

```

2.3.4 Simulazione

Per la simulazione abbiamo disattivato il divisore di frequenza per assicurarci che il conteggio di secondi, minuti e ore avvenisse correttamente, usando un clock di 10 nano-secondi; solo successivamente ci siamo accertati del corretto funzionamento del divisore di frequenza. Abbiamo dunque dichiarato un process per simulare il clock con frequenza 10 ns e, in un altro process, facciamo partire il contatore alzando il segnale di ingresso *start_stop*, osservando poi come variano le uscite per 800 ns. Di seguito il testbench da noi realizzato per simulare il cronometro.

```
ENTITY cronometro_tb IS
```

```

END cronometro_tb;

ARCHITECTURE bench OF cronometro_tb IS

COMPONENT cronometro
PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    start_stop : IN STD_LOGIC;
    load_sec : IN STD_LOGIC;
    load_min : IN STD_LOGIC;
    load_hours : IN STD_LOGIC;
    set_sec : IN STD_LOGIC_VECTOR(0 TO 5);
    set_min : IN STD_LOGIC_VECTOR(0 TO 5);
    set_hours : IN STD_LOGIC_VECTOR(0 TO 4);
    seconds : OUT STD_LOGIC_VECTOR(0 TO 5);
    minutes : OUT STD_LOGIC_VECTOR(0 TO 5);
    hours : OUT STD_LOGIC_VECTOR(0 TO 4)
);
END COMPONENT;

SIGNAL clk : STD_LOGIC;
SIGNAL rst : STD_LOGIC := '0';
SIGNAL start_stop : STD_LOGIC := '0';
SIGNAL load_sec : STD_LOGIC := '0';
SIGNAL load_min : STD_LOGIC := '0';
SIGNAL load_hours : STD_LOGIC := '0';
SIGNAL set_sec : STD_LOGIC_VECTOR(0 TO 5) := (OTHERS => '0');

```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
SIGNAL set_min : STD_LOGIC_VECTOR(0 TO 5) := (OTHERS => '0');

SIGNAL set_hours : STD_LOGIC_VECTOR(0 TO 4) := (OTHERS => '0');

SIGNAL seconds : STD_LOGIC_VECTOR(0 TO 5);

SIGNAL minutes : STD_LOGIC_VECTOR(0 TO 5);

SIGNAL hours : STD_LOGIC_VECTOR(0 TO 4);

CONSTANT clock_period : TIME := 10 ns;

SIGNAL stop_the_clock : BOOLEAN := FALSE;

BEGIN

    uut : cronometro PORT MAP (
        clk => clk,
        rst => rst,
        start_stop => start_stop,
        load_sec => load_sec,
        load_min => load_min,
        load_hours => load_hours,
        set_sec => set_sec,
        set_min => set_min,
        set_hours => set_hours,
        seconds => seconds,
        minutes => minutes,
        hours => hours
    );

stimulus : PROCESS
BEGIN
```

```

        wait for 10 ns;

start_stop <= '1';

wait for 800 ns;

start_stop <= '0';

stop_the_clock <= true;
WAIT;
END PROCESS;

clocking : PROCESS
BEGIN
    WHILE NOT stop_the_clock LOOP
        clk <= '0', '1' AFTER clock_period / 2;
        WAIT FOR clock_period;
    END LOOP;
    WAIT;
END PROCESS;
END;

```

In Figura 2.18 l'output della simulazione effettuata su Vivado.

2.3.5 Esercizio 5.2

Sintetizzare e implementare su board il componente sviluppato al punto precedente, utilizzando i display a 7 segmenti per la visualizzazione

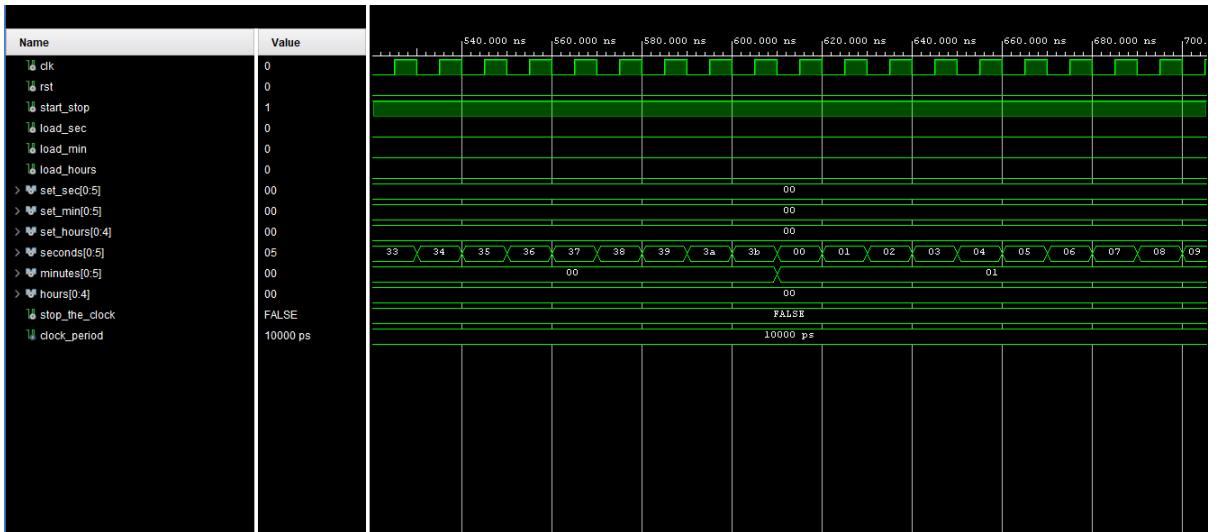


Figura 2.18: Simulazione del cronometro

dell'orario (o una combinazione di display e led nel caso in cui i display a disposizione siano in numero inferiore a quello necessario), gli switch per l'immissione dell'orario iniziale e due pulsanti, uno per il set dell'orario e uno per il reset. Si utilizzi una codifica a scelta dello studente per la visualizzazione dell'orario sui display (esadecimale o decimale).

2.3.6 Sintesi su board di sviluppo

Per visualizzare le uscite del cronometro sul display a 7 segmenti della board, in aggiunta all'implementazione del cronometro dell'esercizio precedente, abbiamo inserito 4 DEBOUNCER per i pulsanti *start_stop*, *load_sec*, *load_min* e *load_hours*, 6 switch per prelevare gli input *set_smh*, due ENCODER che codificano in decimale i valori di ore, minuti e secondi. Un ENCODER è stato utilizzato per codi-

ficare le uscite del CRONOMETRO in modo tale che possano essere mostrate sul display mentre l'altro codifica i valori che l'utente vuole inserire manualmente attraverso gli switch.

```

entity converti_count is
    Port (
        secondi : in std_logic_vector(0 to 5);
        minuti : in std_logic_vector(0 to 5);
        ore : in std_logic_vector(0 to 4);
        outp : out std_logic_vector(23 downto 0)
    );
end converti_count;

architecture beh of converti_count is
    signal secondi_u : integer;
    signal secondi_d : integer;
    signal minuti_u : integer;
    signal minuti_d : integer;
    signal ore_u : integer;
    signal ore_d : integer;

    signal secondi_tot : std_logic_vector(7 downto 0);
    signal minuti_tot : std_logic_vector(7 downto 0);
    signal ore_tot : std_logic_vector(7 downto 0);
    signal uscita_temp : std_logic_vector(23 downto 0);

begin
    secondi_u <= to_integer(unsigned(secondi))mod 10;

```

```

secondi_d <= to_integer(unsigned(secondi)) / 10;
minuti_u <= to_integer(unsigned(minuti)) mod 10;
minuti_d <= to_integer(unsigned(minuti)) / 10;
ore_u <= to_integer(unsigned(ore)) mod 10;
ore_d <= to_integer(unsigned(ore)) / 10;

secondi_tot (3 downto 0) <= std_logic_vector
(to_unsigned(secondi_u, 4));
secondi_tot (7 downto 4) <= std_logic_vector
(to_unsigned(secondi_d, 4));

minuti_tot (3 downto 0) <= std_logic_vector
(to_unsigned(minuti_u, 4));
minuti_tot (7 downto 4) <= std_logic_vector
(to_unsigned(minuti_d, 4));

ore_tot (3 downto 0) <= std_logic_vector
(to_unsigned(ore_u, 4));
ore_tot (7 downto 4) <= std_logic_vector
(to_unsigned(ore_d, 4));

uscita_temp(7 downto 0) <= secondi_tot;
uscita_temp(15 downto 8) <= minuti_tot;
uscita_temp(23 downto 16) <= ore_tot;

outp <= uscita_temp;
end beh;

```

Queste modifiche sono state inserite all'interno della nuova entità

SISTEMA con cui ci occupiamo di effettuare i collegamenti tra tutti i componenti presenti e di mostrare il valore corretto sul display a seconda dei bottoni di cui il processo sequenziale è sensibile. Di seguito è riportato il codice di SISTEMA con i collegamenti tra le componenti del circuito finale:

```
entity sistema is
Port(    clk : IN STD_LOGIC;
          rst : IN STD_LOGIC;
          start_stop : IN STD_LOGIC;
          load_sec : IN STD_LOGIC;
          load_min : IN STD_LOGIC;
          load_hours : IN STD_LOGIC;
          set_smh : IN STD_LOGIC_VECTOR(0 TO 5);
          anodes_out : out STD_LOGIC_VECTOR (7 downto 0);
          cathodes_out : out STD_LOGIC_VECTOR(7 downto 0)
);
end sistema;
```

```
architecture Behavioral of sistema is
```

```
COMPONENT cronometro IS
PORT (
          clk : IN STD_LOGIC;
          rst : IN STD_LOGIC;
          start_stop : IN STD_LOGIC;
          load_sec : IN STD_LOGIC;
          load_min : IN STD_LOGIC;
```

```

        load_hours : IN STD_LOGIC;
        set_sec : IN STD_LOGIC_VECTOR(0 TO 5) := "000000";
        set_min : IN STD_LOGIC_VECTOR(0 TO 5) := "000000";
        set_hours : IN STD_LOGIC_VECTOR(0 TO 4) := "00000";
        seconds : OUT STD_LOGIC_VECTOR(0 TO 5);
        minutes : OUT STD_LOGIC_VECTOR(0 TO 5);
        hours : OUT STD_LOGIC_VECTOR(0 TO 4)
    );
END COMPONENT;

```

```

COMPONENT ButtonDebouncer
    GENERIC (
        CLK_period: integer := 10;
        btn_noise_time: integer := 10000000
    );
    PORT ( RST : in STD_LOGIC;
            CLK : in STD_LOGIC;
            BTN : in STD_LOGIC;
            CLEARED_BTN : 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;

```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
        VALUE : in STD_LOGIC_VECTOR (23 downto 0);
        ENABLE : in STD_LOGIC_VECTOR (7 downto 0);
        DOTS : in STD_LOGIC_VECTOR (7 downto 0);
        ANODES : out STD_LOGIC_VECTOR (7 downto 0);
        CATHODES : out STD_LOGIC_VECTOR (7 downto 0));
END COMPONENT;
```

```
COMPONENT converti_count is
```

```
    Port (
        secondi : in std_logic_vector(5 downto 0);
        minuti : in std_logic_vector(5 downto 0);
        ore : in std_logic_vector(4 downto 0);
        outp : out std_logic_vector(23 downto 0)
    );

```

```
END COMPONENT;
```

```
SIGNAL temp_second : STD_LOGIC_VECTOR(0 TO 5);
```

```
SIGNAL temp_min : STD_LOGIC_VECTOR(0 TO 5);
```

```
SIGNAL temp_hours : STD_LOGIC_VECTOR(0 TO 4);
```

```
SIGNAL temp_value : STD_LOGIC_VECTOR(23 DOWNTO 0);
```

```
SIGNAL var_second : STD_LOGIC_VECTOR(0 TO 5) := "000000";
```

```
SIGNAL var_min : STD_LOGIC_VECTOR(0 TO 5) := "000000";
```

```
SIGNAL var_hours : STD_LOGIC_VECTOR(0 TO 4) := "00000";
```

```
SIGNAL var_value : STD_LOGIC_VECTOR(23 DOWNTO 0);
```

```
SIGNAL in_dss : STD_LOGIC_VECTOR(23 DOWNTO 0);
```

```

        SIGNAL cleared_reset : STD_LOGIC;
        SIGNAL cleared_load_sec : STD_LOGIC;
        SIGNAL cleared_load_min : STD_LOGIC;
        SIGNAL cleared_load_hours : STD_LOGIC;

begin
    cro: cronometro
    PORT MAP (
        clk => clk,
        rst => cleared_reset,
        start_stop => start_stop,
        load_sec => cleared_load_sec,
        load_min => cleared_load_min,
        load_hours => cleared_load_hours,
        set_sec => set_smh(0 TO 5),
        set_min => set_smh(0 TO 5),
        set_hours => set_smh(1 TO 5),
        seconds => temp_second,
        minutes => temp_min,
        hours => temp_hours
    );
    dss : display_seven_segments
    PORT MAP( CLK => clk,
        RST => cleared_reset,
        VALUE => in_dss,
        ENABLE => "11111111",

```

```

        DOTS => "00010100",
        ANODES => anodes_out,
        CATHODES => cathodes_out);

encoder_cron : converti_count
Port map(
    secondi => temp_second,
    minuti => temp_min,
    ore => temp_hours,
    outp => temp_value
) ;

encoder_switch : converti_count
Port map(
    secondi => var_second,
    minuti => var_min,
    ore => var_hours,
    outp => var_value
) ;

deb_reset : ButtonDebouncer
PORT MAP ( RST => '0',
            CLK => clk,
            BTN => rst,
            CLEARED_BTN => cleared_reset);

deb_load_sec : ButtonDebouncer
PORT MAP ( RST => '0',

```

```

CLK => clk,
BTN => load_sec,
CLEARED_BTN => cleared_load_sec);

deb_load_min : ButtonDebouncer
PORT MAP ( RST => '0',
CLK => CLK,
BTN => load_min,
CLEARED_BTN => cleared_load_min);

deb_load_hours : ButtonDebouncer
PORT MAP ( RST => '0',
CLK => CLK,
BTN => load_hours,
CLEARED_BTN => cleared_load_hours);

selezione : process (CLK, start_stop, cleared_load_sec,
cleared_load_min, cleared_load_hours)
begin
    if rising_edge(CLK) then
        if start_stop = '0' then
            if cleared_load_sec = '1' then
                var_second <= set_smh;
                in_dss <= var_value;
            elsif cleared_load_min = '1' then
                var_min <= set_smh;
                in_dss <= var_value;
            elsif cleared_load_hours = '1' then

```

```

        var_hours <= set_smh(1 TO 5);

        in_dss <= var_value;

        else

            in_dss <= temp_value;

        end if;

        else

            in_dss <= temp_value;

        end if;

        end if;

    end process selezione;
end Behavioral;
```

In particolare, l'ingresso del DSS è *in_dss* vale *var_value* o *temp_value* a seconda se vogliamo inserire manualmente i valori sul display oppure lasciare che vengano visualizzate le uscite del CRONOMETRO. Riportiamo ora i constraint attivi sulla board:

```

## Clock signal

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

##Switches

set_property -dict { PACKAGE_PIN J15      IO_STANDARD LVCMOS33 }
[get_ports { set_smh[5] }];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16      IO_STANDARD LVCMOS33 }
[get_ports { set_smh[4] }];
#IO_L3N_T0_DQS_EMCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13      IO_STANDARD LVCMOS33 }
[get_ports { set_smh[3] }];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]
```

```

set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 }
[get_ports { set_smh[2] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 }
[get_ports { set_smh[1] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMOS33 }
[get_ports { set_smh[0] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN V10      IOSTANDARD LVCMOS33 }
[get_ports { start_stop }]; #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 N17      IOSTANDARD LVCMOS33 }
[get_ports { load_min }]; #IO_L9P_T1_DQS_14 Sch=btnc

set_property -dict { PACKAGE_PIN P17      IOSTANDARD LVCMOS33 }
[get_ports { load_hours }]; #IO_L12P_T1_MRCC_14 Sch=btnl

set_property -dict { PACKAGE_PIN M17      IOSTANDARD LVCMOS33 }
[get_ports { load_sec }]; #IO_L10N_T1_D15_14 Sch=btnr

set_property -dict { PACKAGE_PIN P18      IOSTANDARD LVCMOS33 }
[get_ports { rst }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd

```

2.3.7 Timing analysis

Nel file di constraint troviamo:

```
create_clock -add -name sys_clk_pin -period 10.00
-waveform {0 5} [get_ports { clk }];
```

Il report riguardante la timing analysis ci dice che il nostro primary clock impostato con periodo di 10 ns soddisfa i timing constraints; possiamo visualizzare gli esiti in Figura 2.19.

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 1,476 ns	Worst Hold Slack (WHS): 0,196 ns	Worst Pulse Width Slack (WPWS):	4,500 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints:	0
Total Number of Endpoints: 561	Total Number of Endpoints: 561	Total Number of Endpoints:	246

All user specified timing constraints are met.

Figura 2.19: Timing analysis cronometro semplice, periodo del clock di 10 ns

Effettuando le diverse prove abbiamo rilevato la massima frequenza di lavoro in corrispondenza di un periodo T pari a 4,5 ns:

```
create_clock -add -name sys_clk_pin -period 4.50
-waveform {0 2.25} [get_ports { clk }];
```

Utilizzando la formula apposita troviamo che FMAX è pari a circa 222 MHz. L'output del report ottenuto per tale periodo di clock è visualizzabile in Figura 2.20.

Design Timing Summary		
Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0,005 ns	Worst Hold Slack (WHS): 0,208 ns	Worst Pulse Width Slack (WPWS): 1,750 ns
Total Negative Slack (TNS): -0,005 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 1	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 561	Total Number of Endpoints: 561	Total Number of Endpoints: 246

Timing constraints are not met.

Figura 2.20: Timing analysis cronometro semplice, periodo del clock di 4,5 ns

2.3.8 Esercizio 5.3

Estendere il componente sviluppato ai punti precedenti in modo che sia in grado di acquisire e memorizzare internamente fino a 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).

2.3.9 Sintesi su board di sviluppo

Per rispondere alle esigenze di questo esercizio abbiamo aggiunto un’ultima entità che è GESTORE_MEMORY in cui abbiamo inserito una MEMORIA modulo N, due CONTATORI per scorrere le locazioni della memoria in scrittura e lettura, un ENCODER per codificare le uscite della MEMORIA in modo tale da poterle mostrare sul display. Quest’entità implementa il seguente ASF:

Sulla base dei valori assunti da *start_gestore* e *en_gestore* si ha una transizione verso gli stati di scrittura (con *start_gestore*='1' e

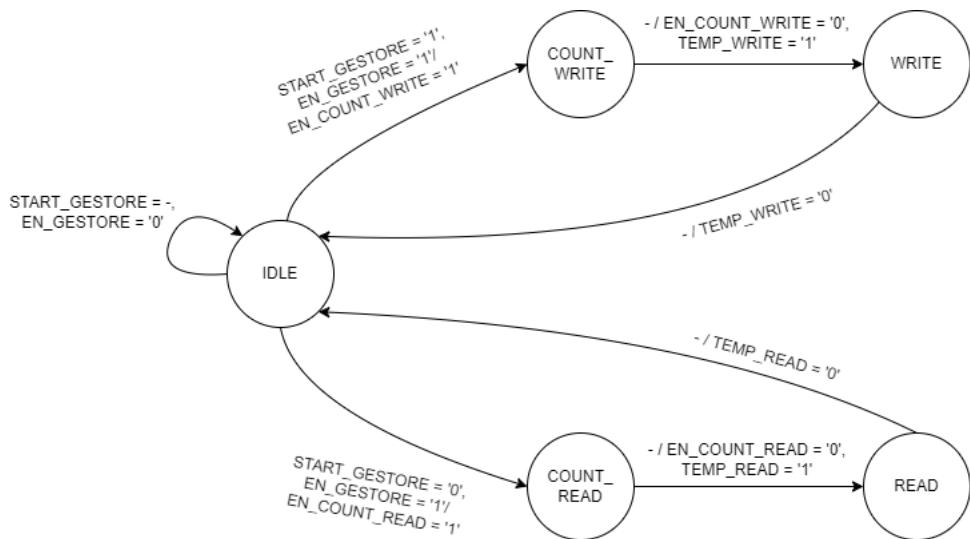


Figura 2.21: ASF del GESTORE_MEMORIA

$en_gestore='1'$) o verso stati di lettura (con $start_gestore='0'$ e $en_gestore='1'$) a partire dallo stato di partenza. Di seguito l'implementazione di GESTORE_MEMORIA:

```

entity gestore_memoria is
    Port (
        clk_gestore : in std_logic;
        start_gestore : in std_logic;
        en_gestore : in std_logic;
        reset_gestore: in std_logic;
        input_gestore : in std_logic_vector(16 downto 0);
        output_gestore : out std_logic_vector(23 downto 0)
    );
end gestore_memoria;

```

```

architecture Behavioral of gestore_memoria is
component memoria is

```

```

Generic(
    len_add : positive := 2
);

Port (
    CLK_mem : in std_logic;
    write : in std_logic;
    read : in std_logic;
    address : in std_logic_vector (len_add-1 downto 0);
    inp_val : in std_logic_vector(16 downto 0);
    out_val : out std_logic_vector(16 downto 0)
);

end component;

component contatore is
generic( N: positive := 2);
port (
    CLK_cont: in std_logic;
    RESET: in std_logic := '0';
    EN_COUNT : in std_logic;
    Y: out std_logic_vector(N-1 downto 0)
);
end component;

COMPONENT converti_count is
Port (
    secondi : in std_logic_vector(5 downto 0);
    minuti : in std_logic_vector(5 downto 0);
    ore : in std_logic_vector(4 downto 0);

```

```

        outp : out std_logic_vector(23 downto 0)
    );
END COMPONENT;

signal temp_out_counter_write : std_logic_vector(1 downto 0);
signal temp_out_mem : std_logic_vector(16 downto 0);
signal temp_address : std_logic_vector(1 downto 0);

signal temp_out_counter_read : std_logic_vector(1 downto 0);

signal temp_write : std_logic;
signal temp_read : std_logic;
signal en_count_read: std_logic;
signal en_count_write: std_logic;

type STATO is (IDLE, COUNT_WRITE, COUNT_READ, WRITE, READ);
signal stato_corrente : STATO := IDLE;
signal stato_prossimo : STATO;

begin

    counter_mem_write : contatore
    port map(
        CLK_cont => clk_gestore,
        RESET => reset_gestore,
        EN_COUNT => en_count_write,
        Y => temp_out_counter_write
    );

```

```

counter_mem_read : contatore
port map(
    CLK_cont => clk_gestore,
    RESET => reset_gestore,
    EN_COUNT => en_count_read,
    Y => temp_out_counter_read
);

MEM : memoria
Port map(
    CLK_mem => clk_gestore,
    write => temp_write,
    read => temp_read,
    address => temp_address,
    inp_val => input_gestore,
    out_val => temp_out_mem
);

encoder_gestore : converti_count
Port map(
    secondi => temp_out_mem(5 downto 0),
    minuti => temp_out_mem(11 downto 6),
    ore => temp_out_mem(16 downto 12),
    outp => output_gestore
);

```

```

comb_gestore : process(start_gestore, en_gestore)
begin
    case stato_corrente is
        when IDLE =>
            if start_gestore = '1' and en_gestore = '1' then
                en_count_write <= '1';
                stato_prossimo <= COUNT_WRITE;
            elsif start_gestore = '0' and en_gestore = '1' then
                en_count_read <= '1';
                stato_prossimo <= COUNT_READ;
            else
                stato_prossimo <= IDLE;
            end if;

        when COUNT_WRITE =>
            en_count_write <= '0';
            temp_write <= '1';
            temp_address <= temp_out_counter_write;
            stato_prossimo <= WRITE;

        when WRITE =>
            temp_write <= '0';
            stato_prossimo <= IDLE;

        when COUNT_READ =>
            en_count_read <= '0';
            temp_address <= temp_out_counter_read;
            temp_read <= '1';
    end case;
end process;

```

```

        stato_prossimo <= READ;

when READ =>

    temp_read <= '0';

    stato_prossimo <= IDLE;

end case;

end process comb_gestore;

seq_gestore: process(clk_gestore)
begin

if rising_edge(clk_gestore) then

    stato_corrente <= stato_prossimo;

end if;

end process seq_gestore;
end Behavioral;

```

Il SISTEMA finale con le ultime modifiche da noi effettuate è il seguente:

```

entity sistema is

Port(    clk : IN STD_LOGIC;
         rst : IN STD_LOGIC;
         wr_mem : IN STD_LOGIC;
         start_stop : IN STD_LOGIC;
         load_sec : IN STD_LOGIC;
         load_min : IN STD_LOGIC;
         load_hours : IN STD_LOGIC;
         set_smh : IN STD_LOGIC_VECTOR(0 TO 5);
         anodes_out : out STD_LOGIC_VECTOR (7 downto 0);
cathodes_out : out STD_LOGIC_VECTOR (7 downto 0)

```

```

);

end sistema;

architecture Behavioral of sistema is

COMPONENT cronometro IS

PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    start_stop : IN STD_LOGIC;
    load_sec : IN STD_LOGIC;
    load_min : IN STD_LOGIC;
    load_hours : IN STD_LOGIC;
    set_sec : IN STD_LOGIC_VECTOR(0 TO 5) := "000000";
    set_min : IN STD_LOGIC_VECTOR(0 TO 5) := "000000";
    set_hours : IN STD_LOGIC_VECTOR(0 TO 4) := "00000";
    seconds : OUT STD_LOGIC_VECTOR(0 TO 5);
    minutes : OUT STD_LOGIC_VECTOR(0 TO 5);
    hours : OUT STD_LOGIC_VECTOR(0 TO 4)
);

END COMPONENT;

COMPONENT ButtonDebouncer
GENERIC (
    CLK_period: integer := 10;
    btn_noise_time: integer := 10000000
);
PORT ( RST : in STD_LOGIC;

```

```

        CLK : in STD_LOGIC;
        BTN : in STD_LOGIC;
        CLEARED_BTN : 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 (23 downto 0);
        ENABLE : in STD_LOGIC_VECTOR (7 downto 0);
        DOTS : in STD_LOGIC_VECTOR (7 downto 0);
        ANODES : out STD_LOGIC_VECTOR (7 downto 0);
        CATHODES : out STD_LOGIC_VECTOR (7 downto 0));
END COMPONENT;

COMPONENT converti_count is
Port (
    secondi : in std_logic_vector(5 downto 0);
    minuti : in std_logic_vector(5 downto 0);
    ore : in std_logic_vector(4 downto 0);
    outp : out std_logic_vector(23 downto 0)
);
END COMPONENT;

```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
COMPONENT gestore_memoria IS
  PORT (
    clk_gestore : in std_logic;
    start_gestore : in std_logic;
    en_gestore : in std_logic;
    reset_gestore : in std_logic;
    input_gestore : in std_logic_vector(16 downto 0);
    output_gestore : out std_logic_vector(23 downto 0)
  );
END COMPONENT;

SIGNAL temp_second : STD_LOGIC_VECTOR(0 TO 5);
SIGNAL temp_min : STD_LOGIC_VECTOR(0 TO 5);
SIGNAL temp_hours : STD_LOGIC_VECTOR(0 TO 4);
SIGNAL temp_value : STD_LOGIC_VECTOR(23 DOWNTO 0);
SIGNAL temp_out_gestore : STD_LOGIC_VECTOR(23 DOWNTO 0);
SIGNAL temp_in_gestore : STD_LOGIC_VECTOR(16 DOWNTO 0);

SIGNAL var_second : STD_LOGIC_VECTOR(0 TO 5) := "000000";
SIGNAL var_min : STD_LOGIC_VECTOR(0 TO 5) := "000000";
SIGNAL var_hours : STD_LOGIC_VECTOR(0 TO 4) := "00000";
SIGNAL var_value : STD_LOGIC_VECTOR(23 DOWNTO 0);

SIGNAL in_dss : STD_LOGIC_VECTOR(23 DOWNTO 0);

SIGNAL cleared_reset : STD_LOGIC;
SIGNAL cleared_load_sec : STD_LOGIC;
```

```

SIGNAL cleared_load_min : STD_LOGIC;
SIGNAL cleared_load_hours : STD_LOGIC;
SIGNAL cleared_wr_mem : STD_LOGIC;

begin
    cro: cronometro
        PORT MAP (
            clk => clk,
            rst => cleared_reset,
            start_stop => start_stop,
            load_sec => cleared_load_sec,
            load_min => cleared_load_min,
            load_hours => cleared_load_hours,
            set_sec => set_smh(0 TO 5),
            set_min => set_smh(0 TO 5),
            set_hours => set_smh(1 TO 5),
            seconds => temp_second,
            minutes => temp_min,
            hours => temp_hours
        );
    end;

dss : display_seven_segments
PORT MAP( CLK => clk,
          RST => cleared_reset,
          VALUE => in_dss,
          ENABLE => "11111111",
          DOTS => "00010100",

```

```

ANODES => anodes_out,
CATHODES => cathodes_out);

encoder_cron : converti_count
Port map(
    secondi => temp_second,
    minuti => temp_min,
    ore => temp_hours,
    outp => temp_value
);

encoder_switch : converti_count
Port map(
    secondi => var_second,
    minuti => var_min,
    ore => var_hours,
    outp => var_value
);

gestore_mem : gestore_memoria
PORT MAP (
    clk_gestore => clk,
    start_gestore => start_stop,
    en_gestore => cleared_wr_mem,
    reset_gestore => rst,
    input_gestore => temp_in_gestore,
    output_gestore => temp_out_gestore
);

```

```
deb_reset : ButtonDebouncer
PORT MAP ( RST => '0',
            CLK => clk,
            BTN => rst,
            CLEARED_BTN => cleared_reset);

deb_load_sec : ButtonDebouncer
PORT MAP ( RST => '0',
            CLK => clk,
            BTN => load_sec,
            CLEARED_BTN => cleared_load_sec);

deb_load_min : ButtonDebouncer
PORT MAP ( RST => '0',
            CLK => CLK,
            BTN => load_min,
            CLEARED_BTN => cleared_load_min);

deb_load_hours : ButtonDebouncer
PORT MAP ( RST => '0',
            CLK => CLK,
            BTN => load_hours,
            CLEARED_BTN => cleared_load_hours);

deb_wr_mem : ButtonDebouncer
PORT MAP ( RST => '0',
            CLK => CLK,
```

```

        BTN => wr_mem,
        CLEARED_BTN => cleared_wr_mem);

selezione : process (CLK, start_stop, cleared_load_sec,
cleared_load_min, cleared_load_hours, cleared_wr_mem)
begin
    if rising_edge(CLK) then
        if start_stop = '0' then
            if cleared_load_sec = '1' then
                var_second <= set_smh;
                in_dss <= var_value;
            elsif cleared_load_min = '1' then
                var_min <= set_smh;
                in_dss <= var_value;
            elsif cleared_load_hours = '1' then
                var_hours <= set_smh(1 TO 5);
                in_dss <= var_value;
            elsif cleared_wr_mem = '1' then
                in_dss <= temp_out_gestore;
            end if;
        else
            if cleared_wr_mem = '1' then
                temp_in_gestore <= temp_hours & temp_min & temp_
            end if;
            in_dss <= temp_value;
        end if;
    end if;
end process selezione;

```

```
end Behavioral;
```

Riportiamo ora i constraint attivi sulla board:

```
## Clock signal

set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 }
[get_ports { clk }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5}
[get_ports { clk }];

##Switches

set_property -dict { PACKAGE_PIN J15     IOSTANDARD LVCMOS33 }
[get_ports { set_smh[5] }];
#IO_L24N_T3_RS0_15 Sch=sw[0]

set_property -dict { PACKAGE_PIN L16     IOSTANDARD LVCMOS33 }
[get_ports { set_smh[4] }];
#IO_L3N_T0_DQS_EMCLK_14 Sch=sw[1]

set_property -dict { PACKAGE_PIN M13     IOSTANDARD LVCMOS33 }
[get_ports { set_smh[3] }];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]

set_property -dict { PACKAGE_PIN R15     IOSTANDARD LVCMOS33 }
[get_ports { set_smh[2] }];
#IO_L13N_T2_MRCC_14 Sch=sw[3]

set_property -dict { PACKAGE_PIN R17     IOSTANDARD LVCMOS33 }
[get_ports { set_smh[1] }];
#IO_L12N_T1_MRCC_14 Sch=sw[4]

set_property -dict { PACKAGE_PIN T18     IOSTANDARD LVCMOS33 }
[get_ports { set_smh[0] }];
#IO_L7N_T1_D10_14 Sch=sw[5]

set_property -dict { PACKAGE_PIN V10    IOSTANDARD LVCMOS33 }
[get_ports { start_stop }];
#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 }
```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

```
[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 N17      IOSTANDARD LVCMOS33 }
[get_ports { load_min }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18      IOSTANDARD LVCMOS33 }
[get_ports { wr_mem }]; #IO_L4N_T0_D05_14 Sch=btnu
set_property -dict { PACKAGE_PIN P17      IOSTANDARD LVCMOS33 }
[get_ports { load_hours }]; #IO_L12P_T1_MRCC_14 Sch=btndl
set_property -dict { PACKAGE_PIN M17      IOSTANDARD LVCMOS33 }
[get_ports { load_sec }]; #IO_L10N_T1_D15_14 Sch=btnr
set_property -dict { PACKAGE_PIN P18      IOSTANDARD LVCMOS33 }
[get_ports { rst }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

2.3.10 Timing analysis

Nel file di constraint troviamo:

```
create_clock -add -name sys_clk_pin -period 10.00
-waveform {0 5} [get_ports { clk }];
```

Il report riguardante la timing analisys ci dice che il nostro primary clock impostato con periodo di 10 ns soddisfa i timing constraints; possiamo visualizzare gli esiti in Figura 2.22.

Effettuando le diverse prove abbiamo rilevato la massima frequenza di lavoro in corrispondenza di un periodo T pari a 4,5 ns:

```
create_clock -add -name sys_clk_pin -period 4.50
-waveform {0 2.25} [get_ports { clk }];
```

Design Timing Summary		
Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 1,483 ns	Worst Hold Slack (WHS): 0,101 ns	Worst Pulse Width Slack (WPWS): 3,750 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 750	Total Number of Endpoints: 750	Total Number of Endpoints: 339

All user specified timing constraints are met.

Figura 2.22: Timing analysis cronometro con memorizzazione intertempi, periodo del clock di 10 ns

Utilizzando la formula apposita troviamo che FMAX è pari a circa 222 MHz. L'output del report ottenuto per tale periodo di clock è visualizzabile in Figura 2.23.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0,004 ns	Worst Hold Slack (WHS): 0,104 ns	Worst Pulse Width Slack (WPWS): 1,000 ns
Total Negative Slack (TNS): -0,004 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 1	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 752	Total Number of Endpoints: 752	Total Number of Endpoints: 340

Timing constraints are not met.

Figura 2.23: Timing analysis cronometro con memorizzazione intertempi, periodo del clock di 4,5 ns

2.4 Esercizio 6: Sistema di lettura-elaborazione-scrittura PO_PC

2.4.1 Esercizio 6.1

Progettare, implementare in VHDL e verificare mediante simulazione un sistema dotato di una memoria ROM di N locazioni da 8 bit ciascu-

na, una macchina combinatoria M in grado di trasformare (secondo una funzione a scelta dello studente) la stringa di 8 bit letta dalla ROM in una stringa di 4 bit, e una memoria MEM di N locazioni che memorizza la stringa in output da M . Il sistema si avvia in corrispondenza di un segnale di START che viene fornito esternamente. Una volta avviato, tramite un'apposita unità di controllo che gestisce la tempificazione del sistema, viene scandita una locazione alla volta della ROM e viene scritta la corrispondente locazione di MEM. Gli indirizzi di memoria sono forniti da un contatore. Le memorie ROM e MEM hanno rispettivamente un read e un write sincrono.

2.4.2 Progetto e architettura

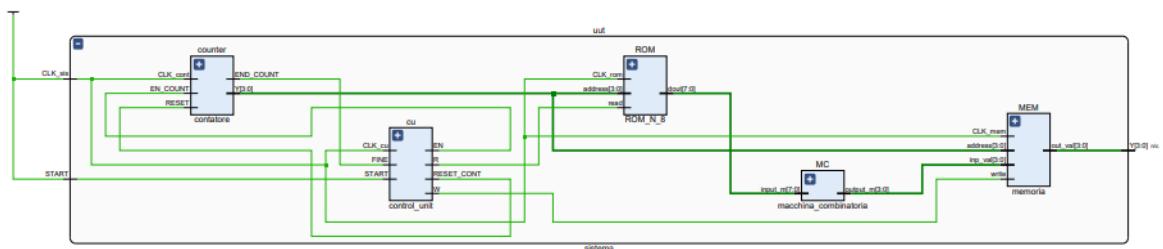


Figura 2.24: Sistema di lettura-elaborazione-scrittura PO _ PC

Il sistema da noi ideato prevede in ingresso un segnale di **START** e un segnale di tempificazione **CLK_sis** ed è composto dalle seguenti parti:

- **COUNTER**: contatore modulo N che scandisce le locazioni della **ROM** e della **MEMORIA**. Questo componente è stato idea-

to in modo tale che al termine del conteggio si resetti e alzi un flag *END_COUNT*;

- **ROM:** memoria di sola lettura costituita da N locazioni di 8 bit ciascuna;
- **MC:** questa macchina combinatoria prende in ingresso il contenuto letto in una delle locazioni della ROM ed esegue delle operazioni logiche di OR tra coppie di bit concatenandone i risultati;
- **MEM:** prende in ingresso l’uscita del componente MC e lo scrive all’indirizzo indicato dal COUNTER;
- **CU:** la control unit CU è un *ASF di Mealy* (Figura 2.25) che coordina le operazioni di lettura su ROM e conseguente scrittura su MEM in seguito all’elaborazione effettuata dalla MC.

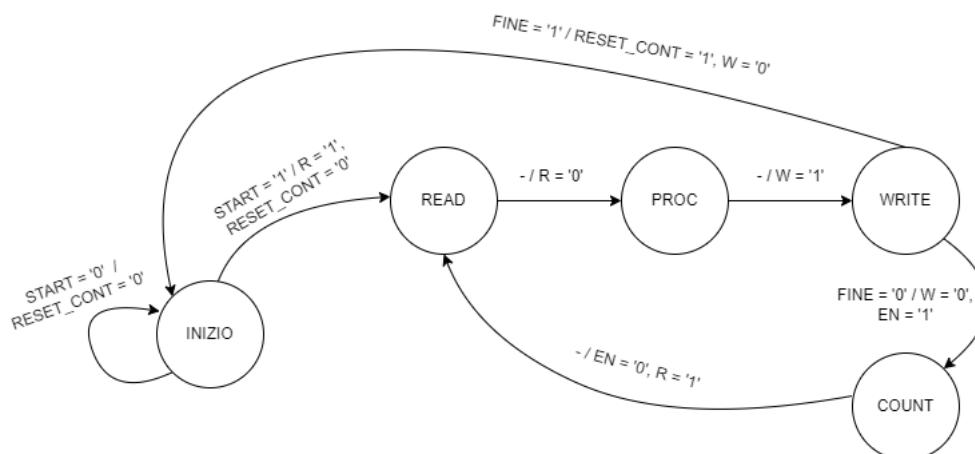


Figura 2.25: ASF realizzato dalla CU

2.4.3 Implementazione

```

entity control_unit is
    Generic( len_add : positive := 4);
    Port(
        CLK_cu : in std_logic;
        START : in std_logic;
        FINE : in std_logic;
        R : out std_logic;
        W : out std_logic;
        RESET_CONT : out std_logic;
        EN : out std_logic
    );
end control_unit;

architecture Behavioral of control_unit is
constant N : positive := 2**len_add;

type STATO is (INIZIO, READ, WRITE, PROC, COUNT);
signal stato_corrente : STATO := INIZIO;
signal stato_prossimo : STATO;

begin
    comb : process(START, FINE, stato_corrente)
begin
    case stato_corrente is

```

```

when INIZIO =>

    RESET_CONT <= '0';

    if START = '1' then
        stato_prossimo <= READ;
        R <= '1';
    end if;

when COUNT =>

    EN <= '0';

    R <= '1';

    stato_prossimo <= READ;

when READ =>

    R <= '0';

    stato_prossimo <= PROC;

when PROC =>

    W <= '1';

    stato_prossimo <= WRITE;

when WRITE =>

    W <= '0';

    if FINE = '1' then
        stato_prossimo <= INIZIO;
        RESET_CONT <= '1';
    else
        EN <= '1';
        stato_prossimo <= COUNT;
    end if;

end case;

end process comb;

```

```

seq: process(CLK_cu)
begin
    if rising_edge(CLK_cu) then
        stato_corrente <= stato_prossimo;
    end if;
end process seq;

end Behavioral;

```

La CU gestisce le transizioni del sistema attraverso i seguenti stati:

- *INIZIO*: è lo stato iniziale del nostro sistema in cui anzitutto resettiamo il conteggio del contatore ponendo il segnale *RESET_CONT* a 0. Se il segnale di inizio elaborazione *START* è alto, allora passiamo nello stato *READ* e alziamo il segnale di abilitazione alla lettura *R*;
- *READ*: in questo stato, effettuata la lettura, abbassiamo il segnale *R* e passiamo nello stato *PROC*
- *PROC*: dopo l'elaborazione della MC abilitiamo la scrittura in *MEM* ponendo ad 1 il segnale *W* e passiamo nello stato di *WRITE*;
- *WRITE*: qui disabilitiamo la scrittura e se il segnale di fine conteggio *FINE* è alto allora ritorniamo nello stato *INIZIO* e reset-

tiamo il contatore alzando *RESET_CONT* altrimenti abilitiamo il segnale *EN* e passiamo nello stato COUNT;

- *COUNT*: disabilitiamo il contatore, poniamo *R* ad 1 e passiamo nello stato READ.

Tutti i componenti finora citati, sono stati dichiarati e istanziati all'interno del componente SISTEMA che li racchiude e li collega tra di loro, come mostrato di seguito nel codice:

```
entity sistema is
    generic( len_add : positive := 4);
    Port (
        CLK_sis : in std_logic;
        START : in std_logic;
        Y : out std_logic_vector(3 downto 0)
    );
end sistema;

architecture Behavioral of sistema is

COMPONENT memoria is
    Generic(
        len_add : positive := 4
    );
    Port (
        CLK_mem : in std_logic;
        write : in std_logic;
        address : in std_logic_vector (len_add-1 downto 0);
    );
end component;
```

```

        inp_val : in std_logic_vector(3 downto 0);
        out_val : out std_logic_vector(3 downto 0)
    );
END COMPONENT;

COMPONENT ROM_N_8 IS
    GENERIC(
        len_add : positive := 4
    );
    PORT (
        CLK_rom : in std_logic;
        address : in std_logic_vector(len_add - 1 downto 0);
        read : in std_logic;
        dout : out std_logic_vector(7 downto 0)
    );
END COMPONENT;

component contatore is
    generic( N: positive := 4);
    port(
        CLK_cont: in std_logic;
        RESET: in std_logic;
        EN_COUNT : in std_logic;
        END_COUNT : out std_logic;
        Y: out std_logic_vector(N-1 downto 0)
    );
end component;

```

```

component macchina_combinatoria is
    Port (
        input_m : in std_logic_vector(7 downto 0);
        output_m : out std_logic_vector(3 downto 0)
    );
end component;

component control_unit is
    Generic( len_add : positive := 4);
    Port (
        CLK_cu : in std_logic;
        START : in std_logic;
        FINE : in std_logic;
        R : out std_logic;
        W : out std_logic;
        RESET_CONT : out std_logic;
        EN : out std_logic
    );
end component;

signal temp_R : std_logic;
signal temp_W : std_logic;
signal temp_RESET_CONT : std_logic;
signal temp_EN : std_logic;
signal temp_END_COUNT : std_logic;
signal temp_address : std_logic_vector(len_add -1 downto 0);
signal temp_input_mc : std_logic_vector(7 downto 0);

```

```

signal temp_out_mc : std_logic_vector(3 downto 0);

begin

cu: control_unit
PORT MAP (
    CLK_cu => CLK_sis,
    START => START,
    FINE => temp_END_COUNT,
    R => temp_R,
    W => temp_W,
    RESET_CONT => temp_RESET_CONT,
    EN => temp_EN
);

counter: contatore
port map(
    CLK_cont => CLK_sis,
    EN_COUNT => temp_EN,
    RESET => temp_RESET_CONT,
    END_COUNT => temp_END_COUNT,
    Y => temp_address
);

ROM : ROM_N_8
PORT MAP (
    CLK_rom => CLK_sis,
    address => temp_address,

```

```

        read => temp_R,
        dout => temp_input_mc
    );
MC : macchina_combinatoria
Port map(
    input_m => temp_input_mc,
    output_m => temp_out_mc
);
MEM : memoria
Port map(
    CLK_mem => CLK_sis,
    write => temp_W,
    address => temp_address,
    inp_val => temp_out_mc,
    out_val => Y
);
end Behavioral;

```

2.4.4 Simulazione

Per la simulazione abbiamo scritto il testbench in cui abbiamo implementato il processo di simulazione del clock e abbiamo alzato il segnale di *START*. Nell'istanziare il sistema abbiamo considerato il generic *len_add* pari a 4 specificando quindi che sia ROM che MEM hanno $N = 2^{len_add}$ locazioni.

```
entity sistema_tb is
```

```

end;

architecture bench of sistema_tb is

component sistema
    generic( len_add : positive := 4);
    Port (
        CLK_sis : in std_logic;
        START : in std_logic;
        Y : out std_logic_vector(3 downto 0)
    );
end component;

signal CLK_sis: std_logic;
signal START: std_logic;
signal Y: std_logic_vector(3 downto 0) ;

constant CLK_period : time := 5 ns;

begin

    uut: sistema generic map ( len_add => 4 )
        port map ( CLK_sis => CLK_sis,
                    START      => START,
                    Y          => Y ) ;

    CLK_process :process
        begin
            CLK_sis <= '0';

```

```

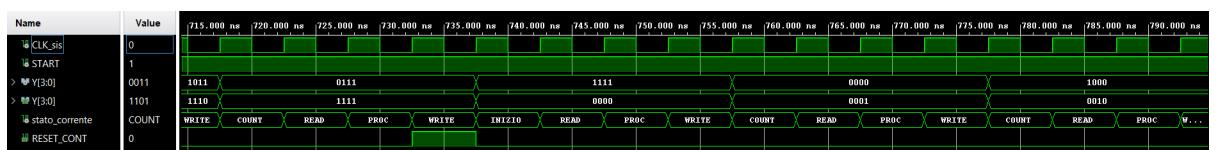
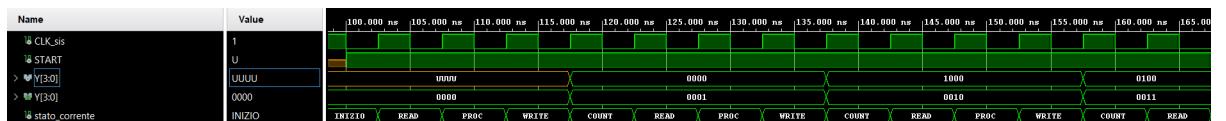
        wait for CLK_period/2;
        CLK_sis <= '1';
        wait for CLK_period/2;
        end process;

stimulus: process
begin
    wait for 100ns;

    START <= '1';

    wait;
end process;
end;
    
```

I risultati ottenuti dalla simulazione del testbench sono i seguenti:



2.4.5 Esercizio 6.2

Sintetizzare e implementare su board il componente sviluppato al punto precedente, utilizzando due bottoni per i segnali di read e reset rispet-

tivamente e i led per la visualizzazione delle uscite della macchina istante per istante.

2.4.6 Sintesi su board di sviluppo

Per rispettare le specifiche richieste per la sintesi su board è stato necessario apportare delle modifiche rispetto al sistema implementato nell'esercizio precedente al SISTEMA e alla CONTROL_UNIT.

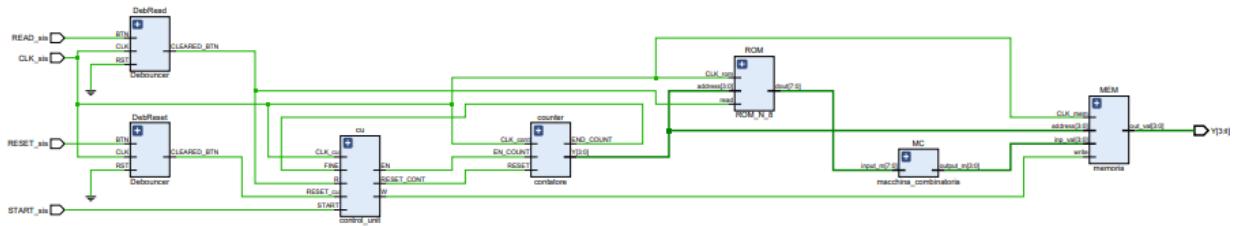


Figura 2.26: Sistema di lettura-elaborazione-scrittura PO_PC su board

Nella CONTROL_UNIT R è stato reso un segnale di input collegato a un bottone attraverso SISTEMA. Inoltre, abbiamo aggiunto un nuovo segnale di ingresso $RESET_{cu}$ che resetta il contatore alla pressione di un bottone della board a cui è collegato. Con queste modifiche abbiamo reso il processo combinatorio sensibile anche ad R e $RESET_{cu}$ e l'abbiamo leggermente adattato. Di seguito il codice corretto:

```

entity control_unit is
    Generic( len_add : positive := 4 );
    Port(

```

```

        CLK_cu : in std_logic;
        START : in std_logic;
        FINE : in std_logic;
        RESET_cu : in std_logic;
        R : in std_logic;
        W : out std_logic;
        RESET_CONT : out std_logic;
        EN : out std_logic
    );
end control_unit;

architecture Behavioral of control_unit is
constant N : positive := 2**len_add;

type STATO is (INIZIO, READ, WRITE, PROC, COUNT);
signal stato_corrente : STATO := INIZIO;
signal stato_prossimo : STATO;

begin

comb : process(START, FINE, stato_corrente, RESET_cu, R)
begin
    if RESET_cu = '1' then
        stato_prossimo <= INIZIO;
        RESET_CONT <= '1';
    else

```

```

case stato_corrente is
    when INIZIO =>
        RESET_CONT <= '0';
        if START = '1' then
            if R = '1' then
                stato_prossimo <= READ;
            else
                stato_prossimo <= INIZIO;
            end if;
        else
            stato_prossimo <= INIZIO;
        end if;

    when COUNT =>
        if FINE = '1' then
            stato_prossimo <= INIZIO;
            RESET_CONT <= '1';
        else
            EN <= '0';
            stato_prossimo <= INIZIO;
        end if;

    when READ =>
        stato_prossimo <= PROC;

    when PROC =>
        W <= '1';
        stato_prossimo <= WRITE;

    when WRITE =>

```

```

        W <= '0';
        EN <= '1';
        stato_prossimo <= COUNT;
    end case;
end if;
end process comb;

seq: process(CLK_cu)
begin
    if rising_edge(CLK_cu) then
        stato_corrente <= stato_prossimo;
    end if;
end process seq;
end Behavioral;

```

In SISTEMA:

- il segnale di *START_sis* adesso è acquisito da uno switch della board e in seguito assegnato allo *START* della CONTROL_UNIT;
- è stata aggiunta la porta *READ_sis* collegata a un bottone della board, che ci consente di leggere una locazione della ROM;
- è stata aggiunta la porta *RESET_sis* collegata a un bottone della board, che ci consente di resettare il contatore;
- è stato aggiunto un DEBOUNCER per *READ_sis* la cui uscita *cleared_read* viene assegnata a *R* di CONTROL_UNIT e a *read* di ROM_N_8;

- è stato aggiunto un DEBOUNCER per *RESET_sis* la cui uscita *cleared_reset* viene assegnata a *RESET_cu* di CONTROL_UNIT.

```

entity sistema is
    generic( len_add : positive := 4);
    Port (
        CLK_sis : in std_logic;
        START_sis : in std_logic;
        READ_sis : in std_logic;
        RESET_sis : in std_logic;
        Y : out std_logic_vector(3 downto 0)
    );
end sistema;

architecture Behavioral of sistema is

COMPONENT memoria is
    Generic(
        len_add : positive := 4
    );
    Port (
        CLK_mem : in std_logic;
        write : in std_logic;
        address : in std_logic_vector (len_add-1 downto 0);
        inp_val : in std_logic_vector(3 downto 0);
        out_val : out std_logic_vector(3 downto 0)
    );
END COMPONENT;

```

```

COMPONENT ROM_N_8 IS
  GENERIC (
    len_add : positive := 4
  );
  PORT (
    CLK_rom : in std_logic;
    address : in std_logic_vector(len_add - 1 downto 0);
    read : in std_logic;
    dout : out std_logic_vector(7 downto 0)
  );
END COMPONENT;

component contatore is
  generic( N: positive := 4);
  port(
    CLK_cont: in std_logic;
    RESET: in std_logic;
    EN_COUNT : in std_logic;
    END_COUNT : out std_logic;
    Y: out std_logic_vector(N-1 downto 0)
  );
end component;

component macchina_combinatoria is
  Port(
    input_m : in std_logic_vector(7 downto 0);
    output_m : out std_logic_vector(3 downto 0)
  );

```

```

    );
end component;

component control_unit is
  Generic( len_add : positive := 4);
  Port(
    CLK_cu : in std_logic;
    START : in std_logic;
    FINE : in std_logic;
    RESET_cu : in std_logic;
    R : in std_logic;
    W : out std_logic;
    RESET_CONT : out std_logic;
    EN : out std_logic
  );
end component;

component Debouncer
  GENERIC (
    CLK_period: integer := 10;
    btn_noise_time: integer := 10000000
  );
  PORT ( RST : in STD_LOGIC;
    CLK : in STD_LOGIC;
    BTN : in STD_LOGIC;
    CLEARED_BTN : out STD_LOGIC);
end component;

```

```

signal temp_W : std_logic;
signal temp_RESET_CONT : std_logic;
signal temp_EN : std_logic;
signal temp_END_COUNT : std_logic;
signal temp_address : std_logic_vector(len_add -1 downto 0);
signal temp_input_mc : std_logic_vector(7 downto 0);
signal temp_out_mc : std_logic_vector(3 downto 0);

SIGNAL cleared_read : STD_LOGIC ;
SIGNAL cleared_reset : STD_LOGIC;

begin

cu: control_unit
PORT MAP (
    CLK_cu => CLK_sis,
    START => START_sis,
    FINE => temp_END_COUNT,
    RESET_cu => cleared_reset,
    R => cleared_read,
    W => temp_W,
    RESET_CONT => temp_RESET_CONT,
    EN => temp_EN
);

counter: contatore
port map(
    CLK_cont => CLK_sis,

```

```

EN_COUNT => temp_EN,
RESET => temp_RESET_CONT,
END_COUNT => temp_END_COUNT,
Y => temp_address

);

ROM : ROM_N_8

PORT MAP (
    CLK_rom => CLK_sis,
    address => temp_address,
    read => cleared_read,
    dout => temp_input_mc
);

MC : macchina_combinatoria

Port map(
    input_m => temp_input_mc,
    output_m => temp_out_mc
);

MEM : memoria

Port map(
    CLK_mem => CLK_sis,
    write => temp_W,
    address => temp_address,
    inp_val => temp_out_mc,
    out_val => Y
);

```

```

DebRead : Debouncer
Port map(
    RST => '0',
    CLK => CLK_sis,
    BTN => READ_sis,
    CLEARED_BTN => cleared_read
);

DebReset : Debouncer
Port map(
    RST => '0',
    CLK => CLK_sis,
    BTN => RESET_sis,
    CLEARED_BTN => cleared_reset
);
end Behavioral;

```

Le porte sono state mappate attraverso i seguenti constraint:

```

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

##Switches
set_property -dict { PACKAGE_PIN J15      IO_STANDARD LVCMOS33 }
[get_ports { START_sis }]; #IO_L24N_T3_RS0_15 Sch=sw[0]

```

```

## LEDs

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

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

set_property -dict { PACKAGE_PIN J13      IOSTANDARD LVCMOS33 }
[get_ports { Y[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]

set_property -dict { PACKAGE_PIN N14      IOSTANDARD LVCMOS33 }
[get_ports { Y[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]

##Buttons

set_property -dict { PACKAGE_PIN P17      IOSTANDARD LVCMOS33 }
[get_ports { READ_sis }]; #IO_L12P_T1_MRCC_14 Sch=btnl

set_property -dict { PACKAGE_PIN M17      IOSTANDARD LVCMOS33 }
[get_ports { RESET_sis }]; #IO_L10N_T1_D15_14 Sch=btnr

```

2.4.7 Timing analysis

Nel file di constraint troviamo:

```

create_clock -add -name sys_clk_pin -period 10.00
-waveform {0 5} [get_ports { CLK_sis }];

```

Il report riguardante la timing analisys ci dice che il nostro primary clock impostato con periodo di 10 ns soddisfa i timing constraints; possiamo visualizzare gli esiti in Figura 2.27.

Effettuando le diverse prove abbiamo rilevato la massima frequenza di lavoro in corrispondenza di un periodo T pari a 3,5 ns:

```

create_clock -add -name sys_clk_pin -period 3.50

```

CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 5,296 ns	Worst Hold Slack (WHS): 0,076 ns	Worst Pulse Width Slack (WPWS): 3,750 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 260	Total Number of Endpoints: 260	Total Number of Endpoints: 100

All user specified timing constraints are met.

Figura 2.27: Timing analysis sistema lettura-scrittura-elaborazione, periodo del clock di 10 ns

```
-waveform {0 1.75} [get_ports { CLK_sis }];
```

Utilizzando la formula apposita troviamo che FMAX è pari a circa 262 MHz. L'output del report ottenuto per tale periodo di clock è visualizzabile in Figura 2.28.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0,318 ns	Worst Hold Slack (WHS): 0,092 ns	Worst Pulse Width Slack (WPWS): 0,500 ns
Total Negative Slack (TNS): -4,755 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 38	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 260	Total Number of Endpoints: 260	Total Number of Endpoints: 100

Timing constraints are not met.

Figura 2.28: Timing analysis sistema lettura-scrittura-elaborazione, periodo del clock di 3,5 ns

Capitolo 3

Macchine aritmetiche

3.1 Esercizio 7: Moltiplicatore di Booth

Un moltiplicatore è una macchina aritmetica che esegue l'operazione di moltiplicazione tra gli operandi che riceve in ingresso. Esistono diverse tipologie di moltiplicatori, a seconda dell'architettura e degli algoritmi implementati; inoltre la moltiplicazione tra numeri positivi (cioè senza segno) differisce dalla stessa operazione effettuata su una coppia di operandi di cui almeno uno ha segno negativo. A tal proposito, il moltiplicatore di Booth, sfruttando la codifica omonima, è in grado di effettuare l'operazione tra un moltiplicando e un moltiplicatore appartenenti ai numeri relativi ed espressi binario in complemento a 2.

3.1.1 Esercizio 7.1

Progettare, implementare in VHDL e simulare una macchina moltiplicatore di Booth in grado di effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna.

3.1.2 Progetto e architettura

La realizzazione del moltiplicatore ha come base il moltiplicatore di Robertson già implementato che ci è stato fornito il materiale didattico. In effetti, l'architettura di tale macchina ha diversi punti in comune con il moltiplicatore di Booth. Di maggiore interesse per noi sono state però le differenze; dati due operandi di n bit ciascuno:

- il registro AQ complessivamente ha un bit in più nel caso di Booth, in quanto è sempre presente uno 0 finale che consente di effettuare tutti i confronti delle coppie di bit necessari, per un totale di $2n+1$ bit;
- non è necessario il latch F che interviene nel caso di uno dei due operandi negativi, in quanto Booth è in grado di riconoscere, sulla base della configurazione della coppia di bit, quando effettuare lo shift semplice (00 o 11), l'addizione (01) o la sottrazione (11);
- non è necessario il MUX 2:1 che interviene a trasmettere l'operando M oppure 0 al sommatore parallelo.

Sulla base di queste differenze architetturali siamo intervenuti nella modifica dell'unità operativa di Robertson. L'architettura del moltiplicatore di Booth è schematizzata in Figura 3.1.

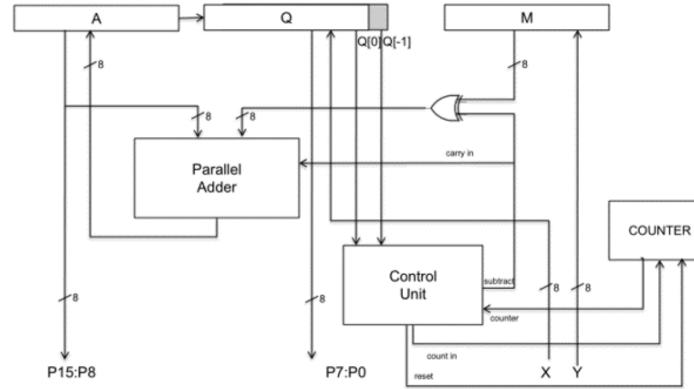


Figura 3.1: Architettura del moltiplicatore di Booth

Anche dal punto di vista comportamentale ci sono differenze in quanto cambia l'algoritmo utilizzato, che per esempio analizza il moltiplicatore a coppie di bit adiacenti. Di conseguenza, siamo intervenuti sull'unità di controllo per implementare l'ASF che modella il moltiplicatore di Booth, visualizzabile in Figura 3.2.

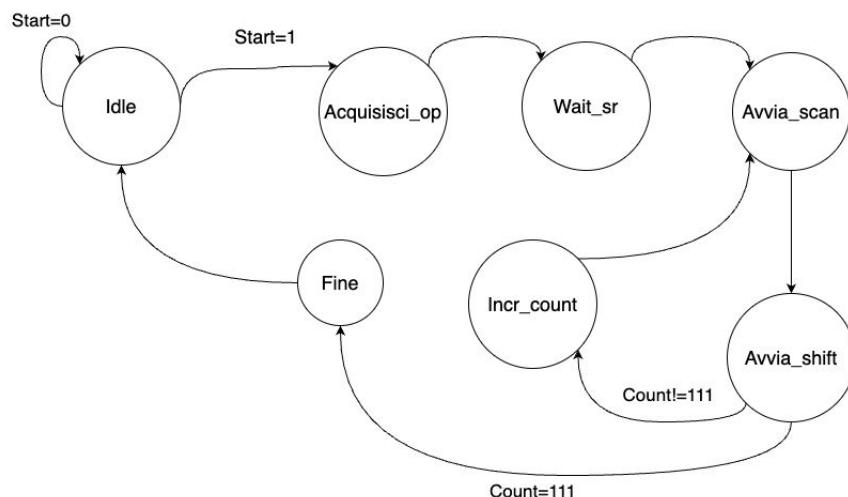


Figura 3.2: Automa a stati finiti per il moltiplicatore di Booth

Vediamo più nel dettaglio stati e transizione della macchina a stati finiti così modellata:

- **idle**: stato iniziale, dove si resta fintanto che il segnale di *start* resta basso; quanto tale segnale si alza si transita in *acquisisci_op*;
- **acquisisci_op**: qui si abilita il caricamento del moltiplicando nel registro M ($\text{loadM} = '1'$), poi il caricamento del moltiplicatore e degli 8 zeri in testa nel registro A.Q ($\text{loadAQ} = '1'$), infine si effettua la transizione in *wait_sr*;
- **wait_sr**: qui si resta in attesa per il completamento del caricamento degli operandi, dopodiché si effettua la transizione in *avvia_scan*;
- **avvia_scan**: in base al valore di q10, si determinano le operazioni successive da effettuare e dunque i segnali giusti da abilitare. Se $q10 = "01"$, si abilita il caricamento in A del risultato della somma; se $q10 = "10"$, si abilita invece la sottrazione e il caricamento in A del risultato della somma; infine, se $q10 = "00"$ o $"11"$, non c'è bisogno di abilitare alcun segnale poiché bisognerà effettuare unicamente lo shift; in tutti e tre i casi verrà effettuata la transizione in *avvia_shift*,

- **avvia_shift:** si abilita lo shift verso destra (`en_shift = '1'`); se `count = "111"`, non bisogna più scorrere il moltiplicatore e si effettua la transizione in *fine*, altrimenti si transita in *incr_count*;
- **incr_count:** si abilita l'incremento del conteggio (`count_in = '1'`), per poi transitare in *avvia_scan*;
- **fine:** si segnala la fine delle operazioni (`stop_cu = '1'`), per poi ritornare in *idle*.

Anche l'unità operativa è stata adattata per effettuare le operazioni necessarie; in particolare, considerando il moltiplicatore X e il moltiplicando Y:

- il registro **M** riceve in ingresso il moltiplicando Y e può caricarlo quando il segnale `loadM` è attivo; il contenuto del registro M è utilizzato come uno dei due operandi per l'adder-subtractor;
- il multiplexer **MUX_SR_parallel_in** seleziona l'ingresso parallelo per lo shift register A.Q in base al segnale `selAQ`; se `selAQ` è attivo, l'ingresso parallelo è dato da `AQ_sum_in` (risultato della somma), altrimenti è dato da `AQ_init` (valore iniziale);
- lo shift register A.Q, denominato **SR**, riceve in ingresso un valore iniziale (`AQ_init`) o un risultato di somma (`AQ_sum_in`) a seconda del segnale `selAQ`. La modalità di shift è controllata dal segnale `shift`, mentre il segnale `loadAQ` abilita il caricamento

nel registro; l'uscita del shift register è rappresentata dal segnale AQ_{out} ;

- l'adder-subtractor **ADD_SUB** riceve in ingresso il valore corrispondente ai bit più significativi di AQ_{out} e il moltiplicando ($Mreg$). Il segnale sub controlla se deve effettuare una sottrazione al posto di un'addizione; il risultato dell'operazione è rappresentato dal segnale sum ;
- il contatore modulo 8 **CONT** è responsabile di incrementare il conteggio ogni volta che il segnale $count_in$ è attivo; il risultato del conteggio è rappresentato dal segnale $count$.

Tra i segnali di appoggio utilizzati troviamo $Mreg$ per trasmettere Y da M ad **ADD_SUB**; AQ_{init} , AQ_{in} , AQ_{out} , che trasmettono input e output di SR; $SRserialIn$, che viene utilizzato come ingresso seriale di SR ed è determinato dal MSB di AQ_{out} . L'uscita dell'unità operativa è rappresentata dal segnale P , che è collegato all'uscita di SR e contiene il risultato della moltiplicazione su 17 bit; il bit in più è dovuto allo 0 che viene aggiunto come LSB; l'uscita effettiva del moltiplicatore sarà invece su 16 bit, dunque i primi 16 bit di P .

3.1.3 Implementazione

Nel materiale didattico sono state fornite le implementazioni di alcune macchine aritmetiche più semplici, in particolare i sommatori. Il com-

ponente più semplice che troviamo è il **Full Adder**, realizzato secondo un approccio dataflow; riceve in ingresso i due addendi e un riporto, producendo in uscita la somma e il riporto dell'operazione.

```

ENTITY full_adder IS
PORT (
    a, b : IN STD_LOGIC;
    cin : IN STD_LOGIC;
    cout, s : OUT STD_LOGIC
);
END full_adder;

ARCHITECTURE rtl OF full_adder IS

BEGIN

    s <= a XOR b XOR cin;
    cout <= (a AND b) OR (cin AND (a XOR b));

END rtl;

```

Il **Ripple Carry Adder** è stato realizzato per composizione dei Full Adder, in particolare è stato realizzato un sommatore a propagazione del riporto su 8 bit.

```

ENTITY ripple_carry IS
PORT (
    X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    c_in : IN STD_LOGIC;
    c_out : OUT STD_LOGIC;

```

```

Z : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
);

END ripple_carry;

ARCHITECTURE structural OF ripple_carry IS
COMPONENT full_adder IS
PORT (
a, b : IN STD_LOGIC;
cin : IN STD_LOGIC;
cout, s : OUT STD_LOGIC);
END COMPONENT;

SIGNAL temp : STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN

RA0 : full_adder PORT MAP(X(0), Y(0), c_in, temp(0), Z(0));

RA1to6 : FOR i IN 1 TO 6 GENERATE
RA : full_adder PORT MAP(X(i), Y(i), temp(i - 1), temp(i), Z(i));
END GENERATE;

RA7 : full_adder PORT MAP(X(7), Y(7), temp(6), c_out, Z(7));

END structural;

```

Ancora per composizione, sfruttando il Ripple Carry Adder visto, è stato realizzato un **Adder-Subber**, che si comporta come addizio-

natore o sottrattore sfruttando la rappresentazione dei numeri in complemento a due e trattando la differenza $X-Y$ come somma $X+(-Y)$.

Anche in tal caso, il sommatore è su 8 bit.

```
ENTITY 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 adder_sub;
```

```
ARCHITECTURE structural OF adder_sub IS
COMPONENT ripple_carry IS
PORT (
X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
c_in : IN STD_LOGIC;
c_out : OUT STD_LOGIC;
Z : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;
```

```
SIGNAL complementoy : STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```
BEGIN
complemento_y : FOR i IN 0 TO 7 GENERATE
complementoy(i) <= Y(i) XOR cin;
END GENERATE;
```

```
RA : ripple_carry PORT MAP(X, complementoy, cin, cout, Z);
END structural;
```

Oltre ai sommatori, sono state fornite le implementazioni di un MUX 2:1 e di alcune macchine sequenziali notevoli. Il **MUX 2:1** fornito è stato realizzato secondo un approccio dataflow.

```
ENTITY mux_21 IS
  GENERIC (
    width : INTEGER RANGE 0 TO 17 := 8
  );
  PORT (
    x0, x1 : IN STD_LOGIC_VECTOR(width - 1 DOWNTO 0);
    s : IN STD_LOGIC;
    y : OUT STD_LOGIC_VECTOR(width - 1 DOWNTO 0)
  );
END mux_21;
```

```
ARCHITECTURE rtl OF mux_21 IS
```

```
BEGIN
  y <= x0 WHEN s = '0' ELSE
  x1 WHEN s = '1' ELSE
  (OTHERS => '0');
END rtl;
```

Per quanto riguarda le macchine sequenziali, realizzate invece secondo un approccio behavioral, troviamo innanzitutto un **registro di 8 bit**, utilizzato per mantenere il moltiplicando M.

```
--registro parallelo-parallelo che mantiene il valore del moltiplicando
ENTITY registro8 IS
PORT (
A : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
clk, res, load : IN STD_LOGIC;
B : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
);
END registro8;

ARCHITECTURE behavioural OF registro8 IS
SIGNAL temp_b : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN

R_PP : PROCESS (clk)
BEGIN
IF (clk'event AND clk = '1') THEN
IF (res = '1') THEN
temp_b <= (OTHERS => '0');
ELSE
IF (load = '1') THEN
temp_b <= A;
END IF;
END IF;
END IF;
END PROCESS;
B <= temp_b;
END behavioural;
```

Troviamo poi uno **shift register di 17 bit**, utilizzato per il registro

A.Q, che conterrà la somma parziale e che bisogna scorrere verso destra per andare avanti con la moltiplicazione.

```
-- shift register che contiene inizialmente una stringa di 8 zeri e
-- al termine dell'operazione di moltiplicazione conterrà il risultato
ENTITY shift_register IS
PORT (
    parallel_in : IN STD_LOGIC_VECTOR(16 DOWNTO 0);
    serial_in : IN STD_LOGIC;
    clock, reset, load, shift : IN STD_LOGIC;
    parallel_out : OUT STD_LOGIC_VECTOR(16 DOWNTO 0)
);
END shift_register;

ARCHITECTURE behavioural OF shift_register IS

SIGNAL temp : STD_LOGIC_VECTOR(16 DOWNTO 0);

BEGIN

SR : PROCESS (clock)
BEGIN
    IF (clock'event AND clock = '1') THEN
        IF (reset = '1') THEN
            temp <= (OTHERS => '0');
        ELSE
            IF (load = '1') THEN --caricamento iniziale del moltiplicatore
                temp <= parallel_in;
            ELSIF (shift = '1') THEN

```

```

temp(15 DOWNTO 0) <= temp(16 DOWNTO 1);

temp(16) <= serial_in;

END IF;

END IF;

END IF;

END PROCESS;

parallel_out <= temp;

END behavioural;

```

Infine, troviamo un **contatore modulo 8**, utilizzato per conteggiare le operazioni effettuate e capire dunque quando il moltiplicatore dovrà fermarsi.

```

ENTITY cont_mod8 IS
PORT (
    clock, reset : IN STD_LOGIC;
    count_in : IN STD_LOGIC;
    count : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
);
END cont_mod8;

ARCHITECTURE behavioural OF cont_mod8 IS
SIGNAL c : STD_LOGIC_VECTOR(2 DOWNTO 0) := (OTHERS => '0');

BEGIN
CM8 : PROCESS (clock)
BEGIN

```

```

IF (clock'event AND clock = '1') THEN

IF (reset = '1') THEN
  c <= (OTHERS => '0');

ELSE
  IF (count_in = '1') THEN
    c <= STD_LOGIC_VECTOR(unsigned(c) + 1);
    -- c <= c + "111"; posso fare direttamente questa somma se importo I
    -- preferibile non farlo perché sono package non standard
  END IF;
END IF;
END IF;

END PROCESS;

count <= c;

END behavioural;

```

Complessivamente, questi componenti vengono collegati tra loro, attraverso opportuni segnali di appoggio, all'interno dell'**unità operativa**.

```

ENTITY unita_operativa IS
  PORT (
    X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- moltiplicatore
    clock, reset : IN STD_LOGIC;
    loadAQ, shift, loadM, sub, selAQ, count_in : IN STD_LOGIC;
    count : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
    P : OUT STD_LOGIC_VECTOR(16 DOWNTO 0)
  );

```

CAPITOLO 3. MACCHINE ARITMETICHE

```
) ;  
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, load : IN STD_LOGIC;  
    B : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));  
END COMPONENT;  
  
COMPONENT mux_21 IS  
GENERIC (width : INTEGER RANGE 0 TO 17 := 8);  
PORT (  
    x0, x1 : IN STD_LOGIC_VECTOR(width - 1 DOWNTO 0);  
    s : IN STD_LOGIC;  
    y : OUT STD_LOGIC_VECTOR(width - 1 DOWNTO 0));  
END COMPONENT;
```

CAPITOLO 3. MACCHINE ARITMETICHE

```
COMPONENT shift_register IS
    PORT (
        parallel_in : IN STD_LOGIC_VECTOR(16 DOWNTO 0);
        serial_in : IN STD_LOGIC;
        clock, reset, load, shift : IN STD_LOGIC;
        parallel_out : OUT STD_LOGIC_VECTOR(16 DOWNTO 0));
    END COMPONENT;

--component FFD is
--port( clock, reset, d: 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 : OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
    END COMPONENT;

SIGNAL Mreg : STD_LOGIC_VECTOR(7 DOWNTO 0); --segnalet temporaneo
--signal op2: std_logic_vector(7 downto 0); --segnalet temporaneo
SIGNAL AQ_init : STD_LOGIC_VECTOR(16 DOWNTO 0); --segnalet iniziale
SIGNAL AQ_in : STD_LOGIC_VECTOR(16 DOWNTO 0); --segnalet in ingresso
SIGNAL AQ_out : STD_LOGIC_VECTOR(16 DOWNTO 0); --segnalet temporaneo
--signal temp_d: std_logic :='0';
--signal temp_F: std_logic :='0';
SIGNAL sum : STD_LOGIC_VECTOR(7 DOWNTO 0); --uscita del parallelo
```

CAPITOLO 3. MACCHINE ARITMETICHE

```
SIGNAL AQ_sum_in : STD_LOGIC_VECTOR(16 DOWNTO 0);
SIGNAL riporto : STD_LOGIC; -- riporto in uscita dell'adder che
SIGNAL SRserialIn : STD_LOGIC;

BEGIN

-- 1) predisposizione del secondo operando della somma:

-- registro moltiplicando
M : registro8 PORT MAP(Y, clock, reset, loadM, Mreg);

-- mux per selezionare moltiplicando oppure 0 come secondo operando
--MUX_molt: mux_21 generic map (width => 8) port map("00000000", 0, M);
-- 2) predisposizione del primo operando della somma:

--stringa da 16 bit da inserire nello shift register A.Q durante la somma
--? ottenuta concatenando 00000000 con il moltiplicatore X
AQ_init <= "00000000" & X & "0"; --valore da inserire all'inizio della somma

--stringa di 17 bit da inserire nello shift register A.Q durante la somma
AQ_sum_in <= sum & AQ_out(8 DOWNTO 0);

-- mux per selezionare l'ingresso parallelo dello shift register
-- oppure uscita dell'adder AQ_sum_in
MUX_SR_parallel_in : mux_21 GENERIC MAP(width => 17) PORT MAP(AQ_sum_in, 0, AQ_out(8 DOWNTO 0));
-- 3) predisposizione dell'ingresso seriale dello shift register
-- final shift, quando bisogna mantenere il bit pi? significativo
```

```

--segnale che rappresenta il contenuto del latch D
--temp_d <= (Mreg(7) and AQ_out(0)) or temp_F;
--D: FFD port map(clock, reset, temp_d, temp_F);

--SRserialIn <= temp_F when self = '0' else AQ_out(15);
SRserialIn <= AQ_out(16);

-- 4) shift register A.Q
SR : shift_register PORT MAP(AQ_in, SRserialIn, clock, reset,
-- 5) sommatore
ADD_SUB : adder_sub PORT MAP(AQ_out(16 DOWNTO 9), Mreg, sub, s

-- 6) contatore
CONT : cont_mod8 PORT MAP(clock, reset, count_in, count);

--7) uscita del moltiplicatore, corrispondente al valore conte
P <= AQ_out;

END structural;

```

Il comportamento del Moltiplicatore di Booth, che abbiamo già modellato come ASF in precedenza (Figura 3.2), è stato definito invece nell'**unità di controllo**, che quindi regola le transizioni tra i possibili stati della macchina a seconda degli ingressi, fornendo di volta in volta in uscita i segnali di controllo necessari.

```
ENTITY unita_controllo IS
```

CAPITOLO 3. MACCHINE ARITMETICHE

```
PORT (
    q10 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    clock, reset, start : IN STD_LOGIC; -- clock = il clock dell'elaborazione;
    count : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
    loadM, count_in, loadAQ, en_shift : OUT STD_LOGIC;
    selAQ, subtract, stop_cu : OUT STD_LOGIC
);

END unita_controllo;

ARCHITECTURE structural OF unita_controllo IS
    TYPE state IS (idle, acquisisci_op, wait_sr, avvia_scan, avvia_stop);
    SIGNAL current_state, next_state : state;

BEGIN
    --selM <= q0;-- in ogni istante la selezione del mux è data dal
    -- parte dell'automa che rappresenta la memoria
    reg_stato : PROCESS (clock)
    BEGIN
        IF (rising_edge(clock)) THEN
            IF (reset = '1') THEN
                current_state <= idle;
            ELSE
                current_state <= next_state;
            END IF;
        END IF;
    END PROCESS;
```

CAPITOLO 3. MACCHINE ARITMETICHE

```
-- parte dell'automa che rappresenta la logica combinatoria
comb : PROCESS (current_state, start, count)
BEGIN

    -- Attenzione! questo process si attiva ogni volta che c'è un
    -- current_state e count per loro natura variano sempre in corrispondenza
    -- start viene dall'esterno: se non varia (sale e scende) come
    -- in cui il next_state varia ma non ha modo da stabilizzarsi
    -- quando il moltiplicatore sarà messo su board, START dovrà essere
    -- impostato a 1

    count_in <= '0';
    subtract <= '0';
    selAQ <= '0';
    --selF <= '0';
    loadAQ <= '0'; --carica nello shift register
    loadM <= '0'; --carica il moltiplicando nel registro M
    stop_cu <= '0';
    en_shift <= '0'; --segnale che abilita lo shift durante le
    -- operazioni di somma e sottrazione

CASE current_state IS

    WHEN idle =>
        IF (start = '1') THEN
            next_state <= acquisisci_op;
        ELSE
            next_state <= idle;
        END IF;
```

```

--fornisce i segnali di caricamento operandi
WHEN acquisisci_op =>
    loadM <= '1'; --abilita il caricamento del moltiplicando
    loadAQ <= '1'; --abilita il caricamento del moltiplicatore
    --nello shift register A.Q (perché selAQ=0)

next_state <= wait_sr;

--acquisisce gli operandi, su cui il sommatore inizia a sommare
WHEN wait_sr =>
    next_state <= avvia_scan;
    --selAQ <= '1';
    --loadAQ <= '1'; --fornisce il segnale di caricamento del moltiplicatore
    --if(count="111") then
    --if(q0='1') then
    --subtract <='1'; --se questa ? l'ultima iterazione
    --end if;
    --next_state <= avvia_fshift;

--else
--next_state <= avvia_shift;
--end if;

WHEN avvia_scan =>

```

```

IF  (q10 = "01")  THEN
    selAQ <= '1';
    loadAQ <= '1'; --fornisce il segnale di caricame
    next_state <= avvia_shift;

ELSIF (q10 = "10")  THEN
    subtract <= '1';
    selAQ <= '1';
    loadAQ <= '1'; --fornisce il segnale di caricame
    next_state <= avvia_shift;

ELSIF (q10 = "00" OR q10 = "11") THEN
    next_state <= avvia_shift;

END IF;

--carica il risultato della somma in A e da fornisce
WHEN avvia_shift =>

en_shift <= '1';

IF (count = "111")  THEN
    next_state <= fine;
ELSE
    next_state <= incr_count;
END IF;

--esegue lo shift, abilita incremento conteggio e pr
WHEN incr_count =>

```

```

        count_in <= '1';

        next_state <= avvia_scan;

--in correzione fa la sottrazione e predisponde per l
-- WHEN avvia_fshift =>

--          en_shift <= '1';
--          self <= '1';

--          next_state <= fine;

WHEN fine =>

stop_cu <= '1';

next_state <= idle;

END CASE;

END PROCESS;
END structural;

```

Complessivamente, il **Moltiplicatore di Booth** si ottiene collegando unità operativa e unità di controllo, usando dunque un approccio structural.

```

ENTITY molt_booth IS
PORT (
    clock, reset, start : IN STD_LOGIC;

```

```

X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
--stop: out std_logic; --a che serve?
P : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
stop_cu : OUT STD_LOGIC
);

END molt_booth;

```

ARCHITECTURE structural OF molt_booth IS

```

COMPONENT unita_controllo IS
PORT (
q10 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
clock, reset, start : IN STD_LOGIC;
count : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
loadM, count_in, loadAQ, en_shift : OUT STD_LOGIC;
selAQ, subtract, stop_cu : OUT STD_LOGIC);
END COMPONENT;

```

COMPONENT unita_operativa IS

```

PORT (
X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- moltiplicatore
clock, reset : IN STD_LOGIC;
loadAQ, shift, loadM, sub, selAQ, count_in : IN STD_LOGIC;
count : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
P : OUT STD_LOGIC_VECTOR(16 DOWNTO 0));
END COMPONENT;

```

SIGNAL tempq10 : STD_LOGIC_VECTOR(1 DOWNTO 0);

SIGNAL temp_selAQ, temp_clock, temp_sub, temp_loadAQ : STD_LOGIC;

CAPITOLO 3. MACCHINE ARITMETICHE

```
SIGNAL temp_count : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL temp_p : STD_LOGIC_VECTOR(16 DOWNTO 0);
SIGNAL temp_count_in, t_load_add : STD_LOGIC;
SIGNAL fine conteggio : STD_LOGIC;
SIGNAL temp_shift : STD_LOGIC;
SIGNAL temp_loadM : STD_LOGIC;
SIGNAL temp_stop_cu : STD_LOGIC; -- segnale di reset generato da
SIGNAL temp_reset_in : STD_LOGIC; -- segnale di reset in ingresso
--signal temp_self: std_logic;

BEGIN

    UC : unita_controllo PORT MAP
    (
        tempq10, clock, reset, start,
        temp_count,
        temp_loadM, temp_count_in, temp_loadAQ, temp_shift,
        temp_selAQ, temp_sub, temp_stop_cu
    );

    UO : unita_operativa PORT MAP
    (
        X, Y, clock, reset, temp_loadAQ, temp_shift, temp_loadM,
        temp_sub, temp_selAQ, temp_count_in, temp_count, temp_p
    );

    tempq10 <= temp_p(1 DOWNTO 0); --invio all'unità di controllo i
    P <= temp_p(16 DOWNTO 1);
```

```
-- la UO viene resettata sia se arriva un reset dall'esterno sia
-- temp_reset_in <= reset or temp_stop_cu;

stop_cu <= temp_stop_cu;
END structural;
```

3.1.4 Simulazione

Il testbench da noi preparato prevede l'esecuzione di due prodotti, di cui la prima con operandi entrambi positivi, mentre la seconda con operandi di segno discorde.

```
ENTITY mbooth_tb IS
END mbooth_tb;

ARCHITECTURE behavioural OF mbooth_tb IS
COMPONENT molt_booth 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;

CONSTANT clk_period : TIME := 20 ns;

SIGNAL inputx, inputy : STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```

SIGNAL prod : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL clk, res, start : STD_LOGIC;
SIGNAL t_stop_cu : STD_LOGIC;
SIGNAL end_sim : STD_LOGIC := '0';

BEGIN
    uut : molt_booth PORT MAP(clk, res, start, inputx, inputy, prod,
                                t_stop_cu);

    clk_process : PROCESS
    BEGIN
        WHILE (end_sim = '0') LOOP
            clk <= '1';
            WAIT FOR clk_period/2;
            clk <= '0';
            WAIT FOR clk_period/2;
        END LOOP;
        WAIT;
    END PROCESS;

    -- SIMULARE PER 9000 NS

    sim : PROCESS
    BEGIN
        WAIT FOR 100 ns;
        res <= '1';
    END;

```

```
WAIT FOR 20 ns;

res <= '0';

-- ----- operazione numero

-- 15*3=45 (002D)

inputx <= "00001111";

inputy <= "00000011";

-- start deve essere visto da clk_div: poiché sarà generato

-- al button debouncer e il segnale di start deve durare qua

WAIT FOR 40 ns;

start <= '1';

WAIT FOR 20 ns;

start <= '0';

WAIT FOR 600 ns;

res <= '1';
```

```

WAIT FOR 20 ns;

res <= '0';

-- ----- operazione numero

-- 15*(-3)=-45 (0053)

inputx <= "00001111";

inputy <= "11111101";

WAIT FOR 40 ns;

start <= '1';

WAIT FOR 20 ns;

start <= '0';

WAIT;

END PROCESS;

END behavioural;

```

Come possiamo osservare in Figura 3.3 e in Figura 3.4, gli output della simulazione per entrambe le operazioni corrispondono ai risultati

corretti.



Figura 3.3: Simulazione del prodotto 15×3

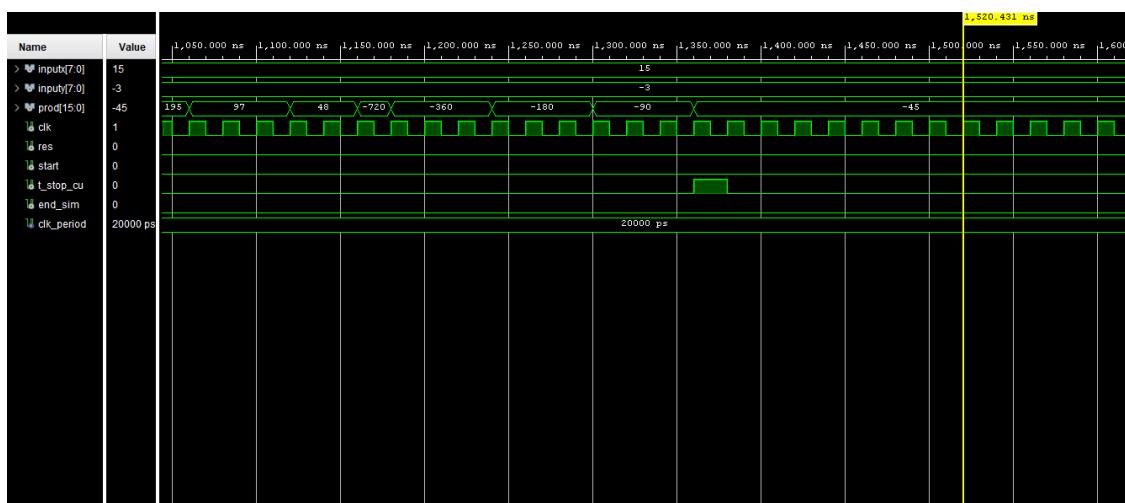


Figura 3.4: Simulazione del prodotto $15 \times (-3)$

3.1.5 Esercizio 7.2

Sintetizzare il moltiplicatore implementato al punto 7.1 su FPGA e testarlo mediante l'utilizzo dei dispositivi di input/output (switch, bot-

(toni, led, display) presenti sulla board di sviluppo in dotazione. La modalità di utilizzo degli stessi è a completa discrezione degli studenti.

3.1.6 Sintesi su board di sviluppo

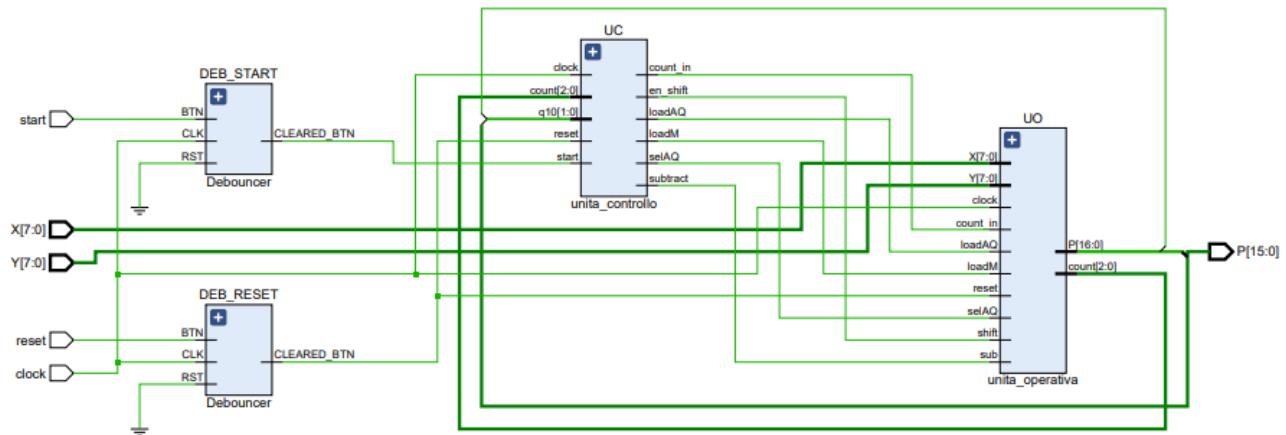


Figura 3.5: Schematic Moltiplicatore di Booth su board

Per l'implementazione del moltiplicatore su board rispetto all'esercizio precedente abbiamo:

- aggiunto i DEBOUNCER per i bottoni di *start* e *reset*;
- aggiunto 16 switch per il caricamento degli operandi (8 switch per operando);
- attivato i 16 led della board per mostrare il risultato della moltiplicazione.

```
entity molt_booth is
    port( clock, reset, start: in std_logic;
```

```

        X, Y: in std_logic_vector(7 downto 0);
        --stop: out std_logic;    --a che serve?
        P: out std_logic_vector(15 downto 0)
        --stop_cu: out std_logic
    );
end molt_booth;

```

```
architecture structural of molt_booth is
```

```

component unita_controllo is
port(
    q10 : in std_logic_vector(1 downto 0);
    clock, reset, start: in std_logic;
    count: in std_logic_vector(2 downto 0);
    loadM, count_in, loadAQ, en_shift: out std_logic;
    selAQ, subtract: out std_logic);
end component;
```

```
component unita_operativa is
```

```

port( X, Y: in std_logic_vector(7 downto 0);--moltiplicatore e m
      clock, reset: in std_logic;
      loadAQ, shift, loadM, sub, selAQ, count_in: in std_logic;
      count: out std_logic_vector(2 downto 0);
      P: out std_logic_vector(16 downto 0));
end component;
```

```
component Debouncer
```

```
generic (
```

```
    CLK_period: integer := 10;  -- periodo del clock (della boar
```

```

btn_noise_time: integer := 10000000 -- durata stimata dell'onda
-- il valore di default
);

port ( RST : in STD_LOGIC;
       CLK : in STD_LOGIC;
       BTN : in STD_LOGIC;
       CLEARED_BTN : out STD_LOGIC);
end component;

signal tempq10: std_logic_vector(1 downto 0);
signal temp_selAQ, temp_clock, temp_sub, temp_loadAQ: std_logic;
signal temp_count: std_logic_vector(2 downto 0);
signal temp_p: std_logic_vector(16 downto 0);
signal temp_count_in, t_load_add: std_logic;
signal fine conteggio: std_logic;
signal temp_shift: std_logic;
signal temp_loadM: std_logic;
signal temp_reset_in: std_logic; -- segnale di reset in ingresso
signal temp_start : std_logic;
signal temp_reset : std_logic;

begin

UC: unita_controllo port map
(tempq10, clock, temp_reset, temp_start,
temp_count,
temp_loadM, temp_count_in, temp_loadAQ, temp_shift,
temp_selAQ, temp_sub);

```

```

UO: unita_operativa port map
(X, Y, clock, temp_reset, temp_loadAQ, temp_shift, temp_loadM,
temp_sub, temp_selAQ, temp_count_in, temp_count, temp_p);

DEB_START : Debouncer port map
('0', clock, start,temp_start);

DEB_RESET : Debouncer port map
('0', clock, reset, temp_reset);

tempq10<=temp_p(1 downto 0); --invio all'unit? di controllo i du
P<=temp_p(16 downto 1);

end structural;

```

I constraint attivi sulla board sono i seguenti:

```

## Clock signal

set_property -dict { PACKAGE_PIN E3      IO_STANDARD 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 }];

##Switches

set_property -dict { PACKAGE_PIN J15      IO_STANDARD LVCMOS33 }
[get_ports { X[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16      IO_STANDARD LVCMOS33 }

```

CAPITOLO 3. MACCHINE ARITMETICHE

```
[get_ports { X[1] }]; #IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 }
[get_ports { X[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 }
[get_ports { X[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 }
[get_ports { X[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMOS33 }
[get_ports { X[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMOS33 }
[get_ports { X[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMOS33 }
[get_ports { X[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8       IOSTANDARD LVCMOS18 }
[get_ports { 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 }
```

CAPITOLO 3. MACCHINE ARITMETICHE

```
[get_ports { Y[7] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

## LEDs

set_property -dict { PACKAGE_PIN H17      IOSTANDARD LVCMOS33 }
[get_ports { P[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15      IOSTANDARD LVCMOS33 }
[get_ports { P[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13      IOSTANDARD LVCMOS33 }
[get_ports { P[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14      IOSTANDARD LVCMOS33 }
[get_ports { P[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
set_property -dict { PACKAGE_PIN R18      IOSTANDARD LVCMOS33 }
[get_ports { P[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
set_property -dict { PACKAGE_PIN V17      IOSTANDARD LVCMOS33 }
[get_ports { P[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17      IOSTANDARD LVCMOS33 }
[get_ports { P[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
set_property -dict { PACKAGE_PIN U16      IOSTANDARD LVCMOS33 }
[get_ports { P[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16      IOSTANDARD LVCMOS33 }
[get_ports { P[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
set_property -dict { PACKAGE_PIN T15      IOSTANDARD LVCMOS33 }
[get_ports { P[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14      IOSTANDARD LVCMOS33 }
[get_ports { P[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
set_property -dict { PACKAGE_PIN T16      IOSTANDARD LVCMOS33 }
[get_ports { P[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15      IOSTANDARD LVCMOS33 }
```

```
[get_ports { P[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
set_property -dict { PACKAGE_PIN V14      IOSTANDARD LVCMOS33 }
[get_ports { P[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12      IOSTANDARD LVCMOS33 }
[get_ports { P[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
set_property -dict { PACKAGE_PIN V11      IOSTANDARD LVCMOS33 }
[get_ports { P[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

##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=btnr
```

3.1.7 Timing analysis

Nel file di constraint troviamo:

```
create_clock -add -name sys_clk_pin -period 10.00
-waveform {0 5} [get_ports { clock }];
```

Il report riguardante la timing analisys ci dice che il nostro primary clock impostato con periodo di 10 ns soddisfa i timing constraints; possiamo visualizzare gli esiti in Figura 3.6.

Effettuando le diverse prove abbiamo rilevato la massima frequenza di lavoro in corrispondenza di un periodo T pari a 3,5 ns:

```
create_clock -add -name sys_clk_pin -period 3.50
-waveform {0 1.75} [get_ports { clock }];
```

CAPITOLO 3. MACCHINE ARITMETICHE

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 4,809 ns	Worst Hold Slack (WHS): 0,205 ns	Worst Pulse Width Slack (WPWS): 4,500 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 267	Total Number of Endpoints: 267	Total Number of Endpoints: 102
All user specified timing constraints are met.		

Figura 3.6: Timing analysis moltiplicatore di Booth, periodo del clock di 10 ns

Utilizzando la formula apposita troviamo che FMAX è pari a circa 273 MHz. L'output del report ottenuto per tale periodo di clock è visualizzabile in Figura 3.7.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0,169 ns	Worst Hold Slack (WHS): 0,237 ns	Worst Pulse Width Slack (WPWS): 1,250 ns
Total Negative Slack (TNS): -3,070 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 44	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 268	Total Number of Endpoints: 268	Total Number of Endpoints: 103
Timing constraints are not met.		

Figura 3.7: Timing analysis moltiplicatore di Booth, periodo del clock di 3,5 ns

Capitolo 4

Comunicazione con handshaking

4.1 Esercizio 8: Comunicazione con handshaking

4.1.1 Esercizio 8.1

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.

4.1.2 Progetto e architettura

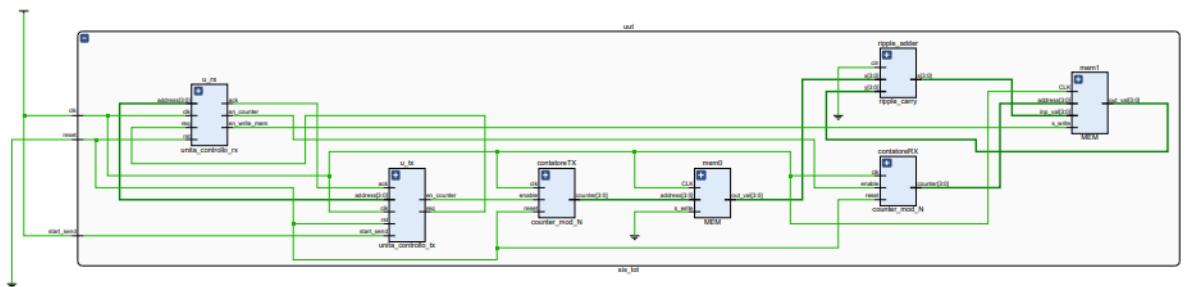


Figura 4.1: Schematic comunicazione handshaking

Per la realizzazione del progetto abbiamo usato come protocollo di comunicazione l’ *handshaking completo*. Abbiamo provveduto all’istanziazione di due componenti: il componente *A* ha la funzione di trasmittitore, mentre il componente *B* ha la funzione di ricevitore. La gestione dei singoli componenti è affidata a due unità di controllo differenti, realizzate come due macchine sequenziali.

Nel caso del trasmittitore, come si può osservare in figura, l’automa passa dallo stato di *IDLE* allo stato di *DATA_LOADED* nel momento in cui riceve il segnale di *start_send*, abilitando così la lettura dalla

memoria per passare successivamente allo stato di *WAITING _ ACK*, nel quale l'unità A alza il segnale di *req*. Se il ricevitore ha inviato il messaggio di *ACK*, possiamo passare allo stato di *WATING _ DONE* nel quale viene abbassato il segnale di *req*. Quando l'unità B termina le operazioni da effettuare, quest'ultima invia il messaggio "1111", permettendo all'unità trasmettente di tornare allo stato di *IDLE* nel caso in cui non ci siano ulteriori messaggi da inviare, altrimenti si passa allo stato di *INCR_ COUNT*.

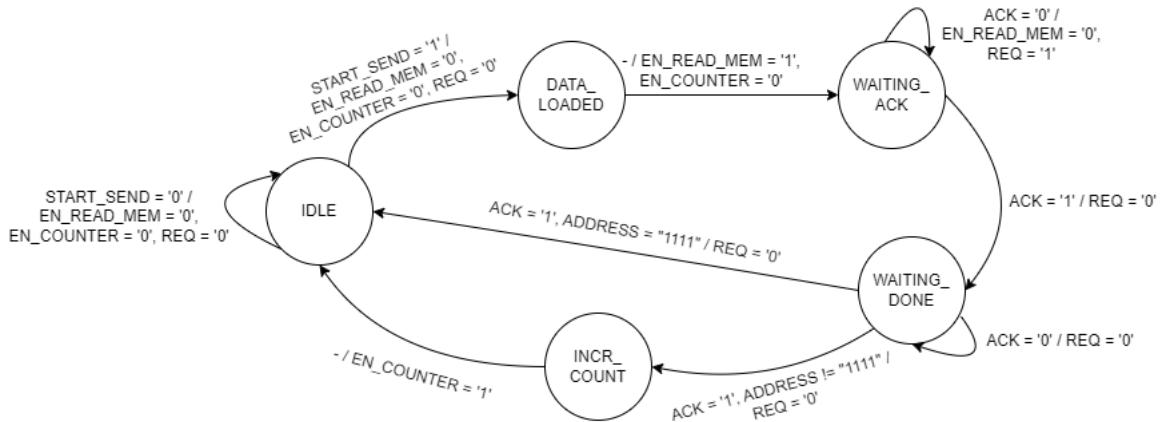


Figura 4.2: ASF del TX

Nel caso dell'unità ricevitore come si può vedere in figura l'automma esce dallo stato di *IDLE* nel momento in cui riceve una richiesta, passando poi allo stato di *READ_IN _ AND _ MEM* il quale legge i dati in ingresso e a abilita la lettura della memoria, passa poi allo stato di *WRITE _ MEM* che permetterà di salvare il risultato dell'operazione eseguita all'interno della memoria. Si passa poi allo stato di

WAIT_END dove se l'unità ricevitore non ha avuto alcuna richiesta invia all'unità A l'indirizzo 1111 per indicare che le operazioni sono state concluse.

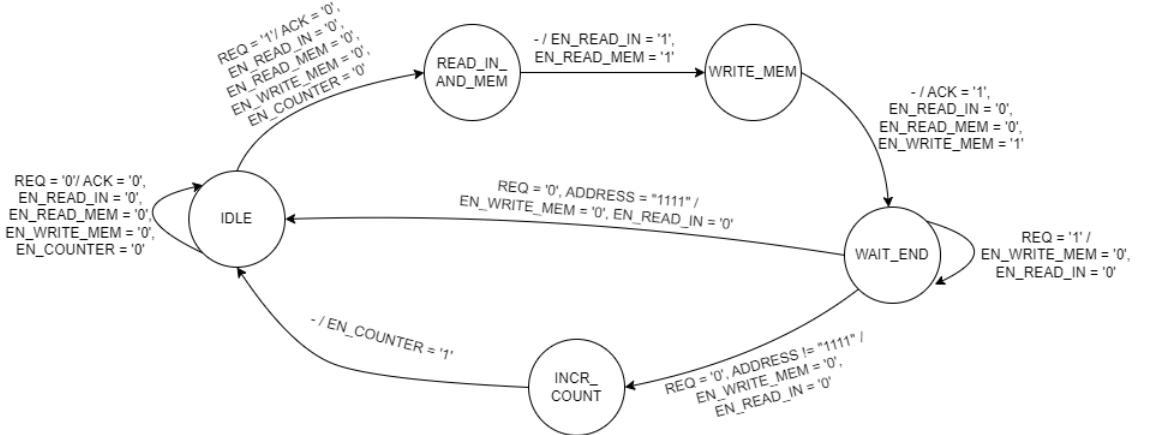


Figura 4.3: ASF del RX

Per compiere le operazioni entrambe le unità A e B necessitano di un contatore modulo 8 che sarà utilizzato per conteggiare l'invio dei messaggi e per accedere alle opportune locazioni di memoria. L'operazione di somma sarà effettuata solo dall'unità B attraverso l'utilizzo di un *full adder*, la quale preleverà l'elemento dalla memoria e lo sommerà con l'elemento ricevuto dall'unità A. Entrambe le unità utilizzeranno poi due componenti *MEM* che serviranno per la lettura dei dati e il salvataggio delle operazioni.

4.1.3 Implementazione

Per l'implementazione del protocollo *headshaking completo* abbiamo realizzato due unità di controllo, di seguito è riportata l'implementazione dell'unità di controllo per A.

Dal punto di vista dell'interfaccia dell'unità di controllo abbiamo dichiarato i seguenti segnali:

- *clk*: utilizzato per permettere di effettuare il passaggio di stato in maniera sincrona con le altre operazioni.
- *rst* : utilizzato per effettuare il reset dell'unità di controllo e impostare nell'automa lo stato di *IDLE*.
- *start_send*: utilizzato per indicare al ricevitore che può iniziare la procedura di invio.
- *ack* : utilizzato dall'unità A per comprendere il l'unità B ha ricevuto correttamente i dati
- *address* : utilizzato per accedere alla memoria e per indicare se l'operazione è terminata.
- *req* : utilizzato per indicare se è presente la richiesta di un'operazione.
- *en_read_mem* : utilizzato per abilitare la lettura dalla memoria
- *en_counter*: utilizzato per abilitare il contatore.

Dopodiché all'interno del blocco *architecture* che essendo un automa è di tipo *behavioral* abbiamo definito il comportamento della macchina indicando prima gli stati di funzionamento dell'automa, precedentemente indicati, poi i segnali di appoggio per gli stati prossimi e correnti. I due process hanno un funzionamento indicato come segue:

- *f_stato_uscita*: sensibile ai seguenti segnali: *stato_attuale*, *start_send*, *address*, *ack*. questo process permette all'automa di spostarsi tra gli stati in base ai segnali che vengono ricevuti dall'unità B per poi generare i dovuti stati prossimi e uscite.
- *memoria*: sensibile al solo *clock*, questo process permette di far variare lo stato prossimo sul fronte di salita del *clock* e di resettare la macchina qualora fosse presente un segnale di *reset*.

```

ENTITY unita_controllo_tx IS
  GENERIC (
    N : INTEGER := 4
  );
  PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    start_send : IN STD_LOGIC;
    ack : IN STD_LOGIC;
    address : IN STD_LOGIC_VECTOR (N - 1 DOWNTO 0);
    req : OUT STD_LOGIC;
  );
END;
  
```

CAPITOLO 4. COMUNICAZIONE CON HANDSHAKING

```

en_read_mem : OUT STD_LOGIC;
en_counter : OUT STD_LOGIC
);

END unita_controllo_tx;

ARCHITECTURE behavioral OF unita_controllo_tx IS

TYPE stato IS (IDLE, DATA_LOADED, WAITING_ACK,
               WAITING_DONE, INCR_COUNT);

SIGNAL stato_attuale : stato := IDLE;
SIGNAL stato_prossimo : stato;

BEGIN

f_stato_uscita : PROCESS (stato_attuale, start_send,
                           address, ack)
BEGIN

CASE stato_attuale IS

WHEN IDLE =>
      req <= '0';
      en_read_mem <= '0';
      en_counter <= '0';

IF (start_send = '0') THEN
      stato_prossimo <= IDLE;
ELSE
      stato_prossimo <= DATA_LOADED;
END IF;

```

```
WHEN DATA_LOADED =>
    en_read_mem <= '1';
    en_counter <='0';
    stato_prossimo <= WAITING_ACK;

WHEN WAITING_ACK =>
    en_read_mem <= '0';
    req <= '1';

IF (ack = '1') THEN
    stato_prossimo <= WAITING_DONE;
ELSE
    stato_prossimo <= WAITING_ACK;
END IF;

WHEN WAITING_DONE =>
    req <= '0';
    IF (ack = '1') THEN
        IF (address = "1111") THEN
            stato_prossimo <= IDLE;
        ELSE
            stato_prossimo <= INCR_COUNT;
        END IF;
    ELSE
        stato_prossimo <= WAITING_DONE;
    END IF;
```

```

        WHEN INCR_COUNT =>
            en_counter <= '1';
            stato_prossimo <= IDLE;

        END CASE;
    END PROCESS;

memoria : PROCESS (clk)
BEGIN
    IF (rising_edge(clk)) THEN
        IF (rst = '1') THEN
            stato_attuale <= IDLE;
        ELSE
            stato_attuale <= stato_prossimo;
        END IF;
    END IF;
END PROCESS;
END behavioral;

```

L'interfaccia dell'unità di controllo prevede i seguenti segnali:

- *clk*: utilizzato per permettere di effettuare il passaggio di stato in maniera sincrona con le altre operazioni;
- *rst*: utilizzato per effettuare il reset dell'unità di controllo e impostare nell'automa lo stato di *IDLE*;
- *ack*: utilizzato per trasmettere all'unità A l'avvenuta ricezione dei dati;

- *address*: utilizzato per accedere alla memoria e per indicare se l'operazione è terminata;
- *req*: utilizzato per indicare se è presente la richiesta di un'operazione;
- *en_read_mem*: utilizzato per abilitare la lettura dalla memoria;
- *en_counter*: utilizzato per abilitare il contatore;
- *en_write_mem*: utilizzato per permettere la scrittura delle operazioni effettuate all'interno della memoria;
- *en_read_in*: utilizzato per effettuare la lettura dei dati dalla memoria.

Di seguito è riportata l'implementazione dell'unità di controllo di B.

```
ENTITY unita_controllo_rx IS
  GENERIC (
    N : INTEGER := 4
  );
  PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    req : IN STD_LOGIC;
    address : IN STD_LOGIC_VECTOR (N - 1 DOWNTO 0);
    ack : OUT STD_LOGIC;
```

```

        en_read_in : OUT STD_LOGIC;
        en_read_mem : OUT STD_LOGIC;
        en_write_mem : OUT STD_LOGIC;
        en_counter : OUT STD_LOGIC
    );
END unita_controllo_rx;

ARCHITECTURE behavioral OF unita_controllo_rx IS
    TYPE stato IS (IDLE, READ_IN_AND_MEM, WRITE_MEM,
                   WAIT_END, INCR_COUNT);

    SIGNAL stato_attuale : stato := IDLE;
    SIGNAL stato_prossimo : stato;
BEGIN

    memoria : PROCESS (clk)
    BEGIN
        IF (rising_edge(clk)) THEN
            IF (rst = '1') THEN
                stato_attuale <= IDLE;
            ELSE
                stato_attuale <= stato_prossimo;
            END IF;
        END IF;
    END PROCESS memoria;

    f_stato_uscita : PROCESS (stato_attuale, req, address)
    BEGIN

```

CAPITOLO 4. COMUNICAZIONE CON HANDSHAKING

```
CASE stato_attuale IS
    WHEN IDLE =>
        ack <= '0';
        en_read_in <= '0';
        en_read_mem <= '0';
        en_write_mem <= '0';
        en_counter <= '0';

        IF (req = '1') THEN
            stato_prossimo <= READ_IN_AND_MEM;
        ELSE
            stato_prossimo <= IDLE;
        END IF;

    WHEN READ_IN_AND_MEM =>
        en_read_in <= '1';
        en_read_mem <= '1';
        stato_prossimo <= WRITE_MEM;

    WHEN WRITE_MEM =>
        ack <= '1';
        en_read_in <= '0';
        en_read_mem <= '0';
        en_write_mem <= '1';
        stato_prossimo <= WAIT_END;

    WHEN WAIT_END =>
        en_write_mem <= '0';
```

```

        if (req = '0') then
            if (address = "1111") then
                stato_prossimo <= IDLE;
            else
                stato_prossimo <= INCR_COUNT;
            end if;
        else
            stato_prossimo <= WAIT_END;
        end if;

WHEN INCR_COUNT =>
    en_counter <= '1';
    stato_prossimo <= IDLE;
END CASE;
END PROCESS;
END behavioral;

```

Di seguito riportiamo L'Implementazione del sistema totale che prevede l'istanziazione di due memorie mem0 e mem1, l'istanziazione di un *full adder* e di due contatori utilizzati per tenere traccia dei messaggi inviati e per accedere alle opportune locazioni di memoria.

```

entity sis_tot is
port (
    clk : in std_logic;
    reset : in std_logic;
    start_send : in std_logic
);

```

```
end entity sis_tot;

architecture rtl of sis_tot is

component MEM is
    port (
        CLK : in std_logic;
        s_write : in std_logic;
        address : in std_logic_vector (3 downto 0);
        out_val : out std_logic_vector(3 downto 0);
        inp_val : in std_logic_vector(3 downto 0)
    );
end component MEM;

component counter_mod_N is
    port (
        clk : in std_logic;
        reset : in std_logic;
        enable : in std_logic;
        counter : out std_logic_vector(3 downto 0)
    );
end component counter_mod_N;

component ripple_carry is
    port (
        x : in std_logic_vector(3 downto 0);
        y : in std_logic_vector(3 downto 0);
        cin : in std_logic;

```

CAPITOLO 4. COMUNICAZIONE CON HANDSHAKING

```
        s : out std_logic_vector(3 downto 0);
        cout : out std_logic
    );
end component ripple_carry;

component unita_controllo_tx is
GENERIC (
    N : INTEGER := 4
);
port (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    start_send : IN STD_LOGIC;
    ack : IN STD_LOGIC;
    address : IN STD_LOGIC_VECTOR (N - 1 DOWNTO 0);

    req : OUT STD_LOGIC;
    en_read_mem : OUT STD_LOGIC;
    en_counter : OUT STD_LOGIC
);
end component unita_controllo_tx;

component unita_controllo_rx is
GENERIC (
    N : INTEGER := 4
);
PORT (
    clk : IN STD_LOGIC;
```

```

        rst : IN STD_LOGIC;
        req : IN STD_LOGIC;
        address : IN STD_LOGIC_VECTOR (N - 1 DOWNTO 0);

        ack : OUT STD_LOGIC;
        en_read_in : OUT STD_LOGIC;
        en_read_mem : OUT STD_LOGIC;
        en_write_mem : OUT STD_LOGIC;
        en_counter : OUT STD_LOGIC
    );
end component unita_controllo_rx;

signal ack_temp : std_logic;
signal address_temp : std_logic_vector(3 downto 0);
signal req_temp : std_logic;
signal en_read_in_temp : std_logic;
signal en_read_mem_temp : std_logic;
signal en_write_mem_temp : std_logic;
signal en_counter_temp_RX : std_logic;
signal en_counter_temp_TX : std_logic;
signal count_temp_TX : std_logic_vector(3 downto 0);
signal count_temp_RX : std_logic_vector(3 downto 0);
signal input_val_temp : std_logic_vector(3 downto 0);
signal registro1_temp : std_logic_vector(3 downto 0);
signal registro2_temp : std_logic_vector(3 downto 0);
signal ris_temp : std_logic_vector(3 downto 0);
signal cout_temp : std_logic;
begin

```

CAPITOLO 4. COMUNICAZIONE CON HANDSHAKING

```
u_tx : unita_controllo_tx
port map (
    clk => clk,
    rst => reset,
    start_send => start_send,
    ack => ack_temp,
    address => address_temp,
    req => req_temp,
    en_read_mem => en_read_mem_temp,
    en_counter => en_counter_temp_TX
);

u_rx : unita_controllo_rx
port map (
    clk => clk,
    rst => reset,
    req => req_temp,
    address => address_temp,
    ack => ack_temp,
    en_read_in => en_read_in_temp,
    en_read_mem => en_read_mem_temp,
    en_write_mem => en_write_mem_temp,
    en_counter => en_counter_temp_RX
);

mem0 : MEM
port map (
```

```

CLK => clk,
s_write => '0',
address => count_temp_TX,
out_val => registro1_temp,
inp_val => input_val_temp
);

mem1 : MEM
port map (
CLK => clk,
s_write => en_write_mem_temp,
address => count_temp_RX,
out_val => registro2_temp,
inp_val => ris_temp
);

contatoreTX : counter_mod_N
port map (
clk => clk,
reset => reset,
enable => en_counter_temp_TX,
counter => count_temp_TX
);

contatoreRX : counter_mod_N
port map (
clk => clk,
reset => reset,
enable => en_counter_temp_RX,

```

```

        counter => count_temp_RX
    );

ripple_adder : ripple_carry
port map (
    x => registro1_temp,
    y => registro2_temp,
    cin => '0',
    s => ris_temp,
    cout => cout_temp
);
end architecture rtl;

```

4.1.4 Simulazione

Di seguito riportiamo l'implementazione della testbench nella quale istanziamo il sistema totale e alziamo il segnale di *start_send* per iniziare la comunicazione dei dati.

```

entity sis_tot_tb is
end entity sis_tot_tb;

architecture bench of sis_tot_tb is
component sis_tot
port(
    clk : in std_logic;
    reset : in std_logic;
    start_send : in std_logic)
;

```

```
end component;

signal CLK, RST: std_logic := '0';
signal start_temp : std_logic := '0';

constant clock_period: time := 10 ns;

begin
    uut: sis_tot
        port map (
            reset => RST,
            clk => CLK,
            start_send=> start_temp
        );

    stimulus: process
    begin
        RST <= '1';
        wait for 10 ns;
        RST <= '0';
        wait for 10 ns;
        start_temp <= '1';

        wait for 20000 ns;

        wait;
    end process;
```

CAPITOLO 4. COMUNICAZIONE CON HANDSHAKING

```

clocking: process

begin

    CLK <= '0';

    wait for clock_period / 2;

    CLK <= '1';

    wait for clock_period / 2;

end process;

end architecture bench;

```

I risultati ottenuti dalla simulazione del testbench sono i seguenti:

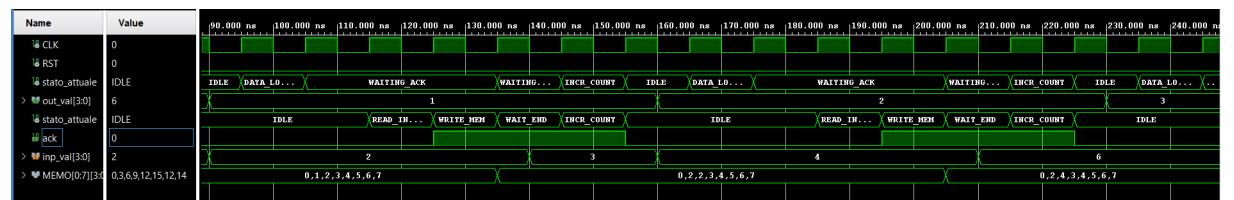


Figura 4.4: Simulazione della comunicazione tramite handshaking

Capitolo 5

Processore

5.1 Esercizio 9: Processore

5.1.1 Esercizio 9.1

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

- *si proceda all'analisi dell'architettura mediante simulazione e si approfondisca lo studio del suo funzionamento per due istruzioni a scelta;*
- *si modifichi un codice operativo a scelta, documentando tutte le modifiche effettuate.*

5.1.2 Analisi architettura e istruzioni

Abbiamo deciso di analizzare le istruzioni di IADD e POP. Iniziamo con l'istruzione di **IADD**, di cui riportiamo il codice a seguire.

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

Vediamo cosa fa nel dettaglio ogni istruzione:

1. La MAR (Memory Address Register) viene impostata uguale allo stack pointer (SP) e poi lo SP viene decrementato di 1. Successivamente, viene eseguita un'operazione di lettura dalla memoria (rd), che porta il valore dalla posizione dello stack pointer (SP) al registro MDR (Memory Data Register);
2. Il valore letto dalla memoria (TOS - Top Of Stack) viene memorizzato in un registro H (holding);
3. Il contenuto del registro MDR viene sommato al valore memorizzato nel registro temporaneo H. Il risultato della somma viene quindi memorizzato nella posizione dello stack pointer (SP) nella memoria. Successivamente, viene eseguita un'operazione di scrittura nella memoria (wr) per memorizzare il risultato della somma.

Dopo aver eseguito con successo l'operazione di IADD, il programma salta alla posizione di memoria "main" per continuare l'esecuzione del programma principale. Questa sequenza di istruzioni esegue dunque l'addizione di due valori interi che si trovano nello stack e memorizza il risultato nello stack stesso. Abbiamo effettuato una simulazione dell'operazione attraverso il seguente program:

```
.main
.var
    a
.endvar

BIPUSH 0xA
BIPUSH 0xE
IADD
ISTORE a

HALT

.endmethod
```

L'output della simulazione, visualizzata tramite **GTKWave**, è mostrato in Figura 5.1. Innanzitutto effettuiamo un *bipush* per inserire sulla punta dello stack il valore 0XA e un secondo *bipush* per il valore 0XE; successivamente viene eseguita l'istruzione di IADD e il suo risultato viene memorizzato nella variabile a, che è visibile sul TOS.

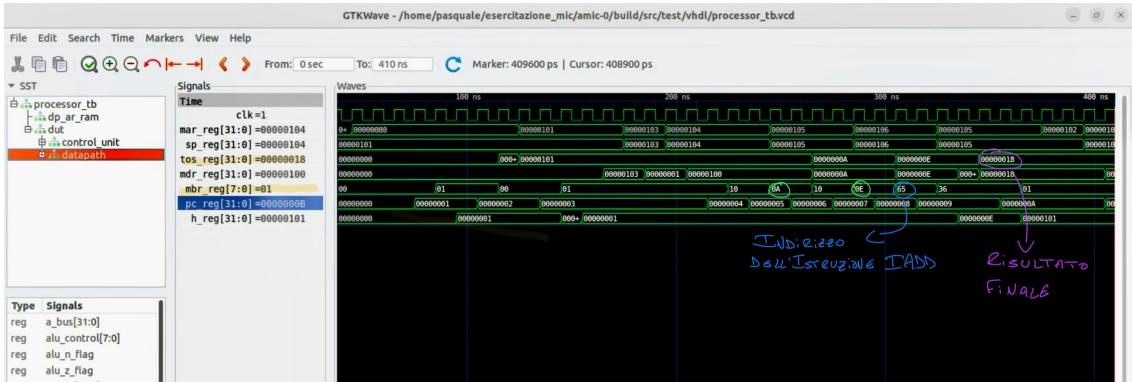


Figura 5.1: Simulazione IADD

La seconda istruzione è la **POP**, che permette di rimuovere l’ultimo elemento inserito nello stack, il codice è come segue:

```
pop = 0x59:
    MAR = SP = SP - 1; rd
    empty
    TOS = MDR; goto main
```

I passaggi sono i seguenti:

1. La MAR (Memory Address Register) viene impostata uguale allo stack pointer (SP) e poi lo SP viene decrementato di 1. Viene eseguita un’operazione di lettura dalla memoria (rd), che porta il valore dalla posizione dello stack pointer (SP) al registro MDR (Memory Data Register);
2. Questa riga indica che lo stack è vuoto (empty), il che potrebbe essere verificato se non ci sono dati da estrarre dallo stack. Nel contesto di un’operazione di pop, questo potrebbe indicare che non c’è nessun dato da estrarre dallo stack;

3. Il valore letto dalla memoria (che è stato precedentemente nello stack) viene memorizzato nel registro TOS (Top of Stack). Quindi, il valore estratto dallo stack (il vecchio TOS) viene memorizzato in TOS per un utilizzo successivo.

Per simulare l'operazione, abbiamo utilizzato il seguente program.

```
.main
    BIPUSH 0xA
    BIPUSH 0xE
    POP
    HALT

.endmethod
```

Come si può osservare dalla simulazione mostrata in Figura 5.2, il valore 0XE viene rimosso dallo stack poiché è quello che è stato inserito per ultimo, sarà dunque presente solo il valore 0XA.

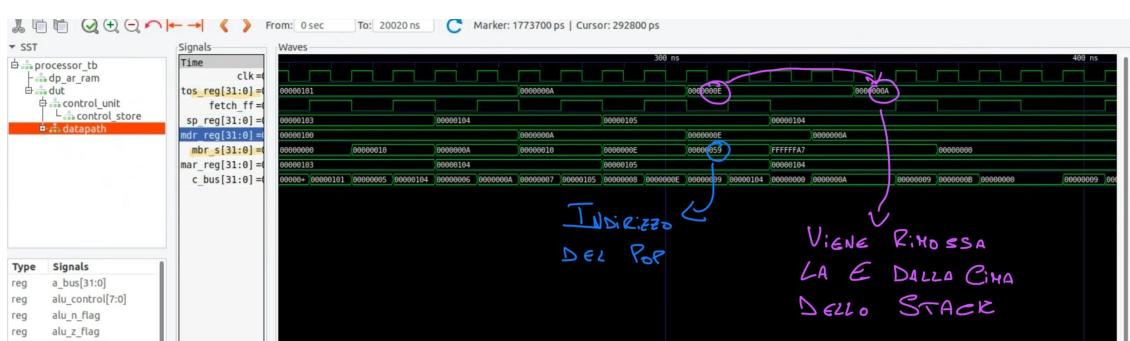


Figura 5.2: Simulazione POP

5.1.3 Modifica del codice operativo

Il codice operativo che abbiamo deciso di modificare è quello dell'operazione **IAND**, che è stata modificata per diventare una **NOTAND**. Il codice modificato è il seguente:

```
iand = 0x7E:  
    MAR = SP = SP - 1; rd  
    H = TOS  
    MDR = TOS = MDR AND H  
    MDR = NOT MDR; wr; goto main
```

Il programma che ci permette di eseguire l'operazione è stato modificato nel seguente modo:

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

L'output della simulazione è visualizzabile in Figura 5.3, dove possiamo osservare che viene effettuata correttamente l'operazione di NOTAND.

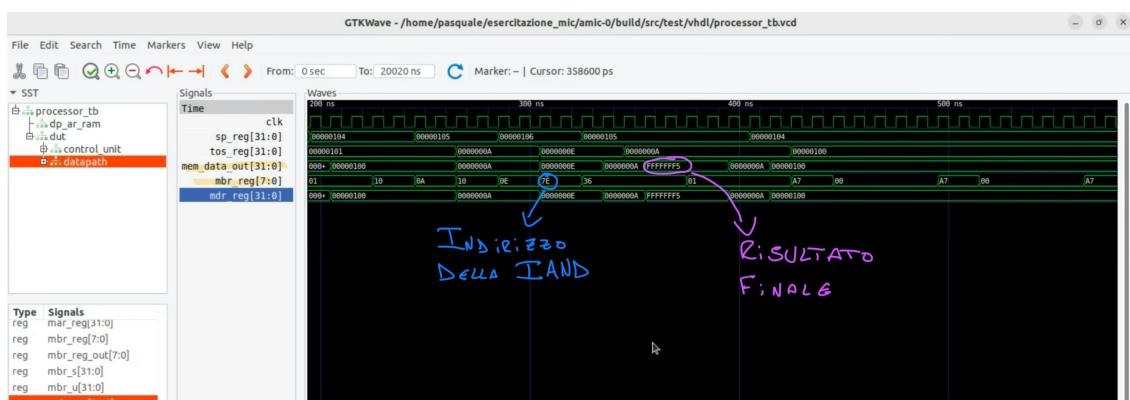


Figura 5.3: Simulazione NOTAND

Capitolo 6

Interfaccia seriale

6.1 Esercizio 10: Interfaccia seriale

6.1.1 Esercizio 10.1

Partendo dall'implementazione fornita dalla Digilent di un dispositivo UART-RS232 (componente RS232RefComp.vhd), progettare, implementare e simulare in VHDL un sistema composto da 2 unità A e B che condividono lo stesso segnale di clock e comunicano tra loro mediante interfaccia seriale. Il sistema A contiene una ROM di 8 locazioni da 1 byte ciascuno, un contatore $CONT_A$ per scandire le locazioni della ROM e una $UART_A$, mentre il sistema B contiene una memoria MEM di 8 locazioni da 1 byte ciascuno, un contatore $CONT_B$ per scandire le locazioni della MEM e una $UART_B$. Quando un segnale WR viene asserito nell'unità A, viene prelevato un byte dalla ROM e inviato all'unità B, che dovrà riceverlo e salvarlo in

MEM.

6.1.2 Progetto e architettura

La UART (Universal Asynchronous Receiver Transmitter) è un dispositivo hardware che supporta la comunicazione seriale asincrona. L'interfaccia UART consiste di 2 segnali TX e RX su cui viaggiano i dati trasmessi/ricevuti, più un segnale di GROUND, e prevede che la comunicazione avvenga secondo uno specifico protocollo che prevede una determinata struttura del frame trasmesso. Nel caso dell'UART, il pacchetto è composto da un bit di START, un byte di dati e un bit di STOP. Per garantire la corretta sincronizzazione, è comune che il ricevitore sovraccampioni la linea, operando a una frequenza 8-16 volte superiore rispetto a quella del trasmettitore. Nel dettaglio, il ricevitore, al rilevamento del bit di START, inizia a campionare i primi 8 campioni, posizionandosi al centro byte, e successivamente campiona ogni 16, mantenendo la sincronizzazione con il flusso di dati. Il trasmettitore, d'altra parte, è più semplice e inizia la trasmissione sotto il controllo dell'unità di controllo. Al termine della comunicazione, alza il flag TBE per indicare che il bus è stato svuotato. Esistono tre modalità di trasmissione:

- Simplex: Comunicazione unidirezionale, in cui solo il trasmettitore può comunicare con il ricevitore;

- Half-Duplex: Trasmettitore e ricevitore possono comunicare sullo stesso canale, ma solo uno alla volta;
- Full-Duplex: Comunicazione bidirezionale permettendo la trasmissione simultanea in entrambe le direzioni.

Il sistema SIS_TOT da noi progettato definisce tre segnali di ingresso *start_tot*, *rst_tot* e *clk_tot* che consentono la comunicazione tra i due nodi A e B. Il NODO_A è costituito da una ROM 8_8, da un contatore modulo 8 che ne scandisce gli indirizzi e dal componente RS232 che implementa l'interfaccia UART. Il comportamento di questo nodo è definito dal seguente ASF:

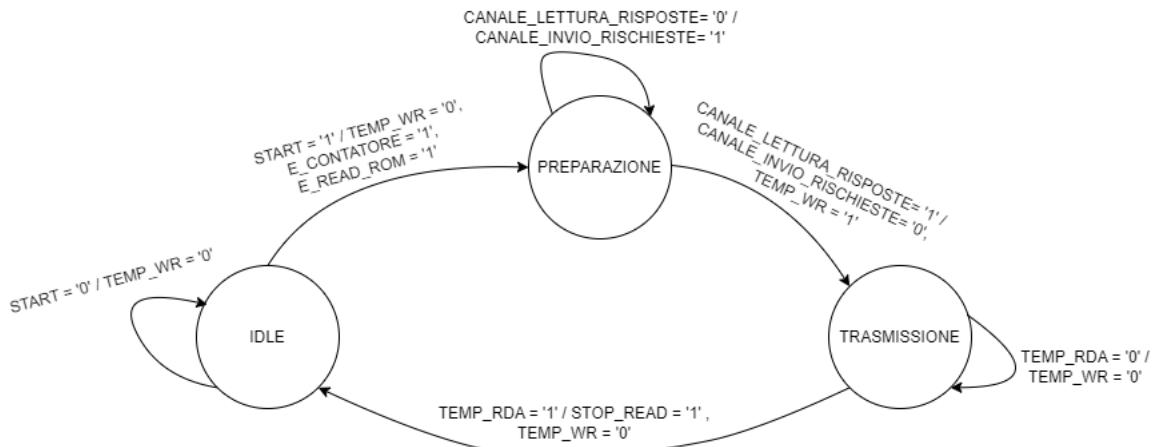


Figura 6.1: ASF del NODO_A

- Stato iniziale (IDLE): il nodo A è IDLE fino a quando non viene richiesta la trasmissione;

- Preparazione dei Dati (PREPARAZIONE): Quando richiesto, il nodo deve preparare i dati da trasmettere leggendoli da una ROM;
- Trasmissione dei Dati (TRASMISSIONE): Dopo la preparazione, i dati vengono trasmessi seguendo il protocollo UART.

Il NODO_B è costituito da una MEM 8_8, da un contatore modulo 8 che ne scandisce gli indirizzi e dal componente RS232 che implementa l'interfaccia UART. Il comportamento di questo nodo è definito dal seguente ASF:

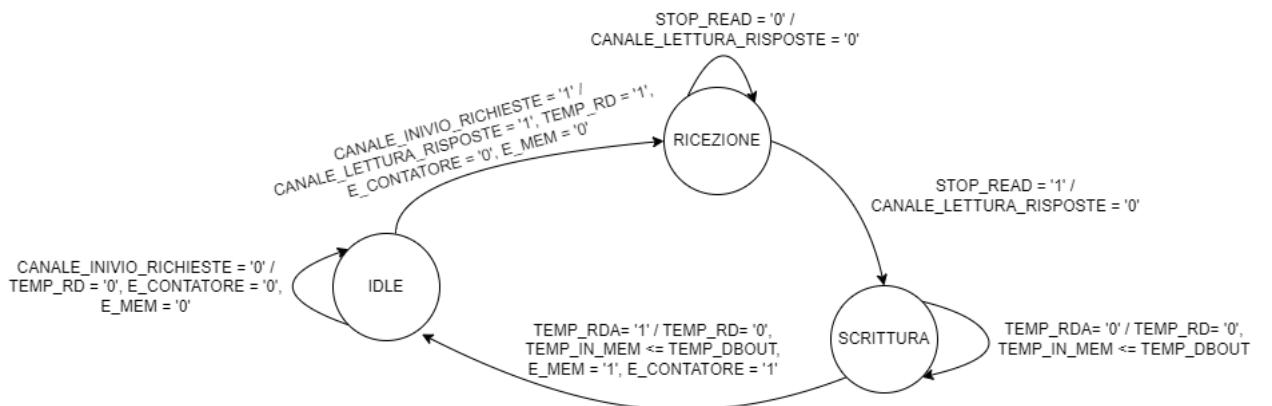


Figura 6.2: ASF del NODO_B

- Stato Iniziale (IDLE): all'avvio del sistema, il nodo B si trova nello stato IDLE, in attesa di richieste di trasmissione dal nodo A. La transizione dallo stato IDLE allo stato RICEZIONE avviene quando il nodo B rileva la richiesta di trasmissione (*RTS*) proveniente da NODO_A.

- RICEZIONE: durante lo stato RICEZIONE, NODO_B attiva il canale di lettura risposte (*CTS*) per indicare al nodo A che è pronta a ricevere. NODO_B rimane in stato RICEZIONE finché il nodo A non comunica la fine della trasmissione attraverso il segnale *stop_read*.
- SCRITTURA: dopo la ricezione completa dei dati, il nodo B passa allo stato SCRITTURA, pronta per scrivere i dati ricevuti in memoria. Durante lo stato SCRITTURA, il nodo B controlla la corretta scrittura dei dati in memoria. Quando la scrittura in memoria è completata, il nodo B ritorna allo stato IDLE, pronta per ricevere nuove richieste di trasmissione.

La struttura del sistema complessivo si presenta come segue:

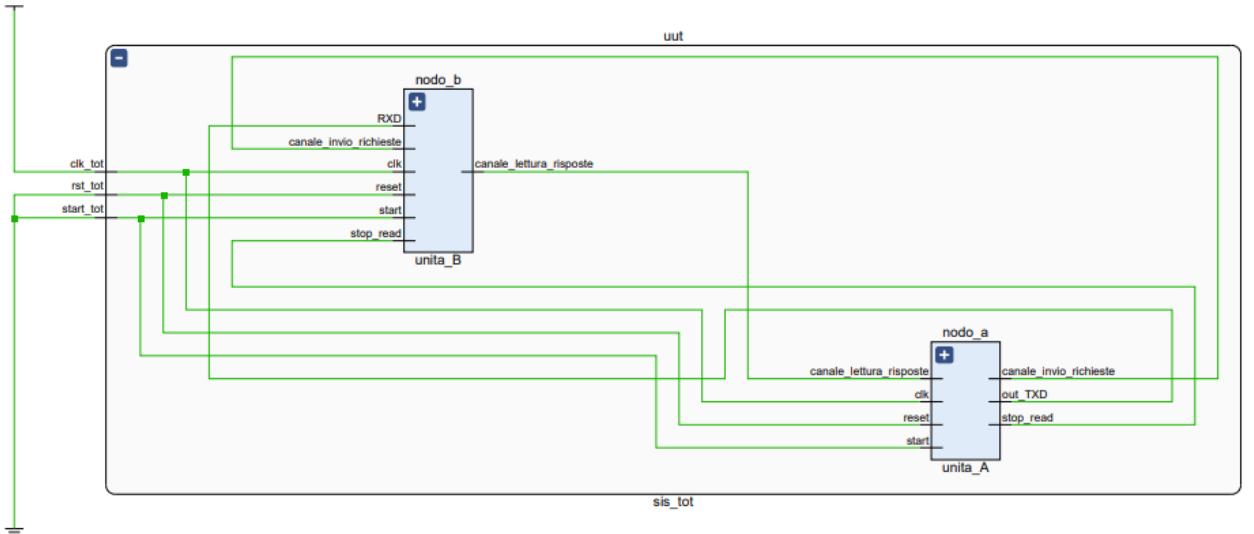


Figura 6.3: Schematic sistema complessivo

6.1.3 Implementazione

Il codice che implementa NODO_A è il seguente:

```
entity unita_A is
    Port (
        clk : in std_logic;
        start: in std_logic;
        reset : in std_logic;
        stop_read : out std_logic;
        out_TXD :out std_logic;
        canale_invio_richieste : out STD_LOGIC := '0';
        canale lettura_risposte : in STD_LOGIC := '0'
    );
end unita_A;

architecture Behavioral of unita_A is

component cont_mod_N is
    generic( N: positive := 3);
    port(
        CLK_cont: in std_logic;
        RESET: in std_logic := '0';
        EN_COUNT : in std_logic;
        Y: out std_logic_vector(N-1 downto 0)
    );
end component;

component ROM_8_8 is
    generic(

```

```
len_add : positive := 3
);

port (
    CLK_rom : in std_logic;
    address : in std_logic_vector(len_add - 1 downto 0);
    read : in std_logic;
    dout : out std_logic_vector(7 downto 0)
);
end component;

component Rs232RefComp is
    Port (
        TXD : out std_logic := '1';
        RXD : in std_logic;
        CLK : in std_logic;
        DBIN : in std_logic_vector (7 downto 0);
        DBOUT : out std_logic_vector (7 downto 0);
        RDA : inout std_logic;
        TBE : inout std_logic := '1';
        RD : in std_logic;
        WR : in std_logic;
        PE : out std_logic;
        FE : out std_logic;
        OE : out std_logic;
        RST : in std_logic := '0');
    end component;
```

```

signal temp_out : std_logic_vector(7 downto 0);
signal temp_addr : std_logic_vector(2 downto 0);

signal temp_DBIN      : std_logic_vector(7 downto 0);
signal temp_DBOUT     : std_logic_vector(7 downto 0);
signal temp_RDA       : std_logic := '0';
signal temp_TBE       : std_logic := '1';
signal temp_PE        : std_logic := '0';
signal temp_FE        : std_logic := '0';
signal temp_OE        : std_logic := '0';
signal temp_RXD       : std_logic := '0';
signal temp_WR        : std_logic := '0';

signal e_contatore   : std_logic := '0';
signal e_read_rom     : std_logic := '0';

type stato is (IDLE, PREPARAZIONE, TRASMISSIONE);
signal stato_attuale : stato := IDLE;
signal stato_prossimo : stato;

begin

CONT_MOD_8 : cont_mod_N
port map(
    CLK_cont => clk,
    RESET => reset,
    EN_COUNT => e_contatore,
    Y => temp_addr
)

```

```

) ;

ROM88 : ROM_8_8
port map (
    CLK_rom => clk,
    address => temp_addr,
    read => '1',
    dout => temp_out
) ;

UART_A : Rs232RefComp
Port map (
    TXD => out_TXD,
    RXD => temp_RXD,
    CLK => clk,
    DBIN => temp_DBIN,
    DBOUT => temp_DBOUT,
    RDA => temp_RDA,
    TBE => temp_TBE,
    RD => '0',
    WR => temp_WR,
    PE => temp_PE,
    FE => temp_FE,
    OE => temp_OE,
    RST => reset
) ;

f_stato_uscita: process(clk)

```

```
begin

if (reset = '1') then
stato_prossimo <= IDLE;
else

case stato_attuale is

when IDLE =>
temp_WR <= '0';
if (start = '1') then
stato_prossimo <= PREPARAZIONE;
e_contatore <= '1';
e_read_rom <= '1';
else
stato_prossimo <= IDLE;
end if;

when PREPARAZIONE =>
if e_read_rom = '1' then
temp_DBIN <= temp_out;
e_contatore <= '0';
e_read_rom <= '0';
end if;

canale_invio_richieste <= '1'; --RTS

if (canale_lettura_risposte = '1') then --SE ARRIVA CTS
canale_invio_richieste <= '0';--OK RTS
```

```

stato_prossimo <= TRASMISSIONE;
temp_WR <= '1';

else
stato_prossimo <= PREPARAZIONE;
end if;

when TRASMISSIONE =>
    if temp_WR = '1' then
        temp_WR <= '0';
    end if;

if (temp_RDA = '1') then
    stop_read <= '1';

stato_prossimo <= IDLE;
else
stato_prossimo <= TRASMISSIONE;

end if;

when others =>
stato_prossimo <= IDLE;
end case;
end if;
end process;

cambio_stato: process (clk)
begin
if (rising_edge(clk) and clk = '1') then
stato_attuale <= stato_prossimo;
end if;

```

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

Il codice che implementa NODO_B è il seguente:

```
entity unita_B is
port(
    clk : in std_logic;
    reset: in std_logic;
    start: in std_logic;
    RXD: in std_logic;
    stop_read : in std_logic;
    canale_invio_richieste : in STD_LOGIC := '0';
    canale_lettura_risposte : out STD_LOGIC := '0';
    uscita : out std_logic_vector(7 downto 0)
);
end entity unita_B;
```

```
architecture behavioral of unita_B is
```

```
component memoria is
Generic(
    len_add : positive := 3
);
Port(
    CLK_mem : in std_logic;
    write : in std_logic;
    address : in std_logic_vector (len_add-1 downto 0);
    inp_val : in std_logic_vector(7 downto 0);
    out_val : out std_logic_vector(7 downto 0)
```

```
) ;  
end component;  
  
component cont_mod_N is  
    generic( N: positive := 3);  
    port (  
        CLK_cont: in std_logic;  
        RESET: in std_logic := '0';  
        EN_COUNT : in std_logic;  
        Y: out std_logic_vector(N-1 downto 0)  
    );  
end component;  
  
component Rs232RefComp is  
    Port (  
        TXD : out std_logic := '1';  
        RXD : in std_logic;  
        CLK : in std_logic;  
        DBIN : in std_logic_vector (7 downto 0);  
        DBOUT : out std_logic_vector (7 downto 0);  
        RDA : inout std_logic ;  
        TBE : inout std_logic := '1';  
        RD : in std_logic ;  
        WR : in std_logic;  
        PE : out std_logic;  
        FE : out std_logic;  
        OE : out std_logic;  
        RST : in std_logic := '0');
```

```
end component;

signal temp_addr : std_logic_vector(2 downto 0);
signal temp_in_mem : std_logic_vector(7 downto 0);

signal temp_DBIN      : std_logic_vector(7 downto 0);
signal temp_DBOUT     : std_logic_vector(7 downto 0);
signal temp_RDA       : std_logic := '0';
signal temp_RD        : std_logic := '0';
signal temp_TBE       : std_logic := '0';
signal temp_TXD       : std_logic := '0';
signal temp_PE         : std_logic := '0';
signal temp_FE         : std_logic := '0';
signal temp_OE         : std_logic := '0';

signal e_contatore   : std_logic := '0';
signal e_mem          : std_logic := '0';

type stato is (IDLE, RICEZIONE, SCRITTURA);
signal stato_attuale : stato := IDLE;
signal stato_prossimo : stato;

begin

CONT_MOD_8 : cont_mod_N
port map(
    CLK_cont => clk,
    RESET => reset,
```

```

EN_COUNT => e_contatore,
Y => temp_addr
);

MEM : memoria
port map(
    CLK_mem => clk,
    write => e_mem,
    address => temp_addr,
    inp_val => temp_in_mem,
    out_val => uscita
);

UART_B : Rs232RefComp
Port map(
    TXD      => temp_TXD,
    RXD      => RXD,
    CLK      => clk,
    DBIN     => temp_DBIN,
    DBOUT    => temp_DBOUT,
    RDA      => temp_RDA,
    TBE      => temp_TBE,
    RD       => temp_RD,
    WR       => '0',
    PE       => temp_PE,
    FE       => temp_FE,
    OE       => temp_OE,
    RST     => reset
);

```

```
) ;  
  
f_stato_uscita: process(clk)  
begin  
  if (reset = '1') then  
    stato_prossimo <= IDLE;  
  else  
  
    case stato_attuale is  
  
      when IDLE =>  
        e_contatore <= '0';  
        e_mem <= '0';  
        if (canale_invio_richieste = '1') then  
          canale lettura_risposte <= '1'; --CTS  
          stato_prossimo <= RICEZIONE;  
          temp_RD <= '1';  
        else  
          temp_RD <= '0';  
        end if;  
  
      when RICEZIONE =>  
        canale lettura_risposte <= '0'; -- OK CTS  
        if (stop_read = '1') then  
          stato_prossimo <= SCRITTURA;  
        else  
          stato_prossimo <= RICEZIONE;  
        end if;  
    end case;  
  end if;  
end process;
```

```

end if;

when SCRITTURA =>
    temp_RD <= '0';
    temp_in_mem <= temp_DBOUT;
    if (temp_RDA = '1') then
        e_mem <= '1';
        e_contatore <= '1';
        stato_prossimo <= IDLE;
    else
        stato_prossimo <= SCRITTURA;
    end if;

when others =>
    stato_prossimo <= IDLE;
end case;

end if;
end process;

cambio_stato: process (clk)
begin
    if (rising_edge(clk) and clk = '1') then
        stato_attuale <= stato_prossimo;
    end if;
end process;
end behavioral;

```

Il codice che implementa SIS_TOT è il seguente:

```
entity sis_tot is
```

```
Port (
      start_tot : in STD_LOGIC;
      rst_tot : in STD_LOGIC;
      clk_tot : in STD_LOGIC
);
end sis_tot;

architecture sis_totArch of sis_tot is
begin
  signal txd_rxd : std_logic;
  signal CTS : std_logic;
  signal RTS : std_logic;
  signal stop : std_logic := '0';

component unita_A is
  Port (
    clk : in std_logic;
    start: in std_logic;
    reset : in std_logic;
    stop_read : out std_logic := '0';
    out_TXD :out std_logic;
    canale_invio_richieste : out STD_LOGIC := '0';
    canale lettura_risposte : in STD_LOGIC := '0'
  );
end component;

component unita_B
port(
  clk : in std_logic;
```

```

        reset: in std_logic;
        start: in std_logic;
        RXD: in std_logic;
        stop_read : in std_logic := '0';
        canale_invio_richieste : in STD_LOGIC := '0';
        canale lettura_risposte : out STD_LOGIC := '0';
        uscita : out std_logic_vector(7 downto 0)
    );
end component;

signal uscita_mem : std_logic_vector(7 downto 0);

begin
    nodo_a: unita_A
    port map
    (
        clk => clk_tot,
        start => start_tot,
        reset => rst_tot,
        stop_read => stop,
        out_TXD => txd_rxd,
        canale_invio_richieste => RTS,
        canale lettura_risposte => CTS
    );
    nodo_b: unita_B
    port map
    (

```

```
clk => clk_tot,  
reset => rst_tot,  
start => start_tot,  
RXD => txd_rxd,  
stop_read => stop,  
canale_invio_richieste => RTS,  
canale_lettura_risposte => CTS,  
uscita => uscita_mem  
) ;
```

```
aggiorna_val: process (clk_tot)  
begin  
end process;  
end sis_totArch;
```

6.1.4 Simulazione

Di seguito riportiamo l'implementazione della testbench nella quale istanziamo il sistema totale:

```
entity sis_tot_tb is  
end entity sis_tot_tb;  
  
architecture bench of sis_tot_tb is  
component sis_tot  
port (  
    start_tot : in STD_LOGIC;  
    rst_tot : in STD_LOGIC;  
    clk_tot : in STD_LOGIC
```

```
) ;  
  
end component;  
  
  
signal CLK, RST: std_logic := '0';  
signal start : std_logic := '0';  
  
  
constant clock_period: time := 10 ns;  
  
  
begin  
uut: sis_tot  
port map (  
    rst_tot => RST,  
    clk_tot => CLK,  
    start_tot => start  
) ;  
  
  
stimulus: process  
begin  
    RST <= '1';  
    start <= '0';  
    wait for 100 ns;  
    RST <= '0';  
    start <= '1';  
    wait for 10 ns;  
    start <= '0';  
    wait for 20000 ns;  
    start <= '0';  
    wait for 100 ns;
```

```
start <= '1';
wait for 10 ns;
start <= '0';
wait for 20000 ns;
start <= '0';
wait for 100 ns;
start <= '1';
wait for 10 ns;
start <= '0';
wait for 20000 ns;
start <= '0';
wait for 100 ns;
start <= '1';
wait for 10 ns;
start <= '0';
wait for 20000 ns;
start <= '0';
wait for 100 ns;
start <= '1';
wait for 10 ns;
start <= '0';
wait for 20000 ns;
start <= '0';
wait for 100 ns;
start <= '1';
wait for 10 ns;
start <= '0';
wait for 20000 ns;
start <= '0';
wait for 100 ns;
start <= '1';
wait for 10 ns;
start <= '0';
wait for 20000 ns;
```

```

start <= '0';
wait for 100 ns;
start <= '1';
wait for 10 ns;
start <= '0';
wait for 20000 ns;
start <= '0';
wait for 100 ns;
start <= '1';
wait for 10 ns;
start <= '0';
wait for 20000 ns;
start <= '0';
wait for 100 ns;
start <= '1';
wait for 10ns;
start <= '0';
wait for 20000 ns;

wait;
end process;

clocking: process
begin
    CLK <= '0';
    wait for clock_period / 2;
    CLK <= '1';
    wait for clock_period / 2;

```

```
end process;  
  
end architecture bench;
```

I risultati ottenuti dalla simulazione del testbench sono i seguenti:

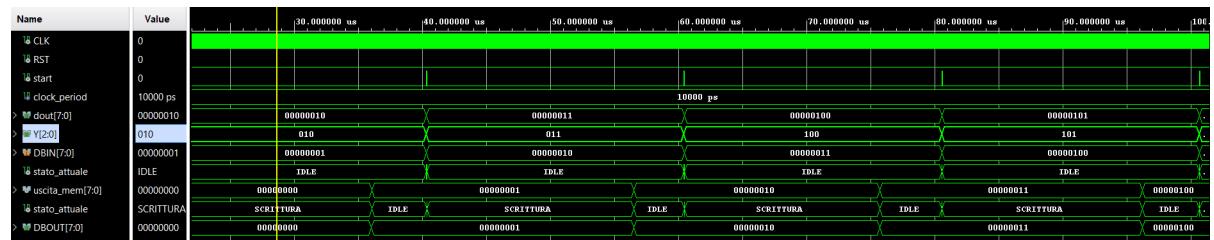


Figura 6.4: Simulazione sistema complessivo

Capitolo 7

Switch multistadio

All'interno di un sistema di elaborazione distribuito, in cui molteplici dispositivi devono comunicare tra loro scambiandosi messaggi, è fondamentale utilizzare un'infrastruttura di interconnessione efficiente, scalabile e modulare, che mantenga cioè una latenza di comunicazione limitata all'aumentare dei nodi interconnessi e che sia facilmente integrabile in sistemi più grandi. Tra le diverse soluzioni possibili, una particolarmente scalabile è la **rete di interconnessione multistadio**, cioè una rete che si basa sull'utilizzo di più stadi intermedi per stabilire la comunicazione tra i nodi del sistema. Il componente fondamentale di tale tipologia di rete è lo **switch**, che in generale connette due sorgenti X₁ e X₂ con due destinazioni Y₁ e Y₂ sfruttando la connessione tra l'uscita di un MUX 2:1 e l'ingresso di un DEMUX 1:2. In particolare, vederemo una particolare rete di interconnessione multistadio NxN che prende il nome di **omega network**, la quale sfrutta

una tecnica chiamata *perfect shuffling* per realizzare il collegamento tra gli switch posizionati nei $\log_2 N$ stadi intermedi.

7.1 Esercizio 11: Switch multistadio

7.1.1 Esercizio 11.1

Progettare e implementare in VHDL uno switch multistadio secondo il modello omega network. Lo switch deve consentire lo scambio di messaggi di 2 bit ciascuno da un nodo sorgente a un nodo destinazione in una rete con 4 nodi, implementando uno schema a priorità fissa fra i nodi (es. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).

7.1.2 Progettazione e architettura

Per progettare la rete richiesta, siamo partiti dalla progettazione di uno **switch elementare**, ottenuto per composizione sfruttando un MUX 2:1 e un DEMUX 1:2. In realtà, una omega network utilizza un tipo di switch simile a quello elementare che però può assumere 4 stati diversi; per semplicità possiamo comunque impiegare lo switch elementare. L’interfaccia dello switch è visualizzabile in Figura 7.1; oltre agli ingressi e alle uscite che già conosciamo possiamo osservare le linee di selezione SRC e DST che si occupano di scegliere quale sorgente (uscita MUX) e quale destinazione (uscita DEMUX) abilitare.

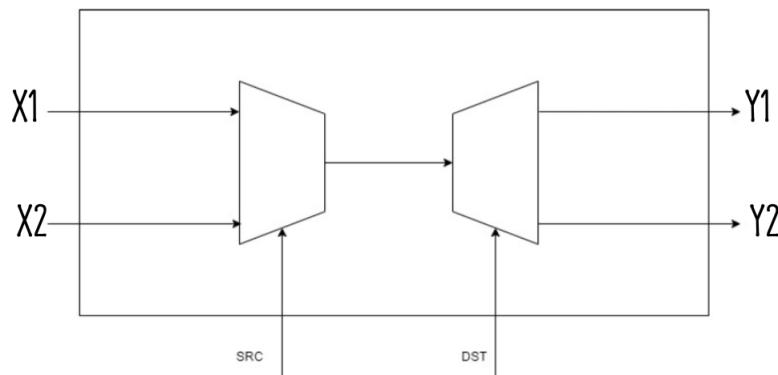


Figura 7.1: Interfaccia switch

Abbiamo progettato di conseguenza l'**unità operativa** della rete per composizione di 4 switch, realizzando così una rete di interconnessione composta da $\log_2 4 = 2$ stadi intermedi contenenti ognuno 2 switch. Chiaramente, i collegamenti tra linee di ingresso e di uscita sono stati realizzati in maniera tale da rispettare il *perfect shuffling*. Dunque, nel primo stadio, il primo switch riceve in ingresso il primo segnale del primo del secondo "mazzo", rispettivamente 0 e 2, mentre il secondo switch riceve in input i segnali 1 e 3, che sono rispettivamente i secondi segnali del primo e del secondo "mazzo". Successivamente si mischiano i "mazzi" (per un totale di $\log_2 4 = 2$ mischiate) per definire gli ingressi degli switch nel secondo stadio: dividiamo il "mazzo" di segnali in metà uguali e le interlacciamo perfettamente, in modo da ritrovarci con il primo segnale della metà sinistra seguito dal primo segnale della metà destra in ingresso al primo switch e così via. In Figura possiamo osservare una schematizzazione della rete così progettata.

I messaggi sono stati pensati come stringhe di 6 bit divise in tre

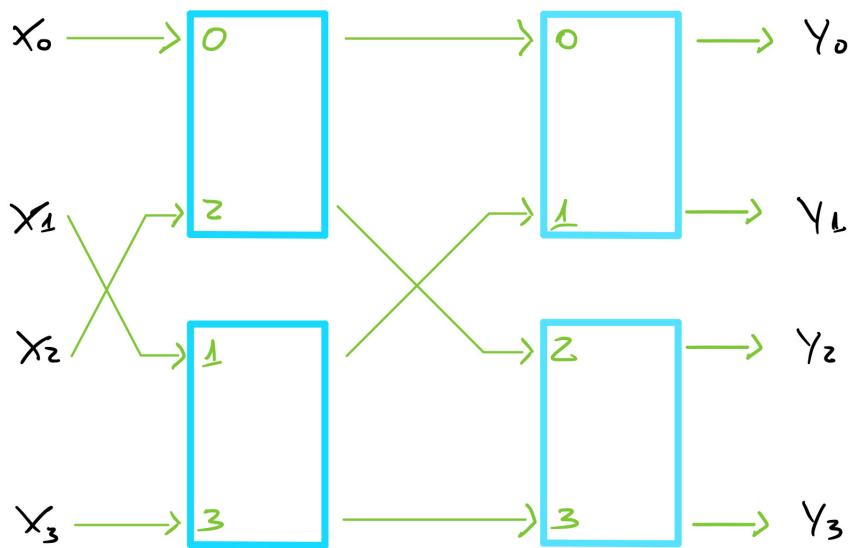


Figura 7.2: Omega network a due stadi

segmenti di due bit; partendo dai due bit più significativi troviamo sorgente, destinazione e messaggio. Poiché i messaggi devono essere filtrati sulla base di una priorità assegnata ai nodi sorgente, abbiamo progettato un gestore di priorità che, come suggerito dalla traccia, fornisce un segnale di selezione per le sorgenti sulla base di una priorità decrescente (il primo nodo ha priorità maggiore del secondo e così via). Dunque, se ci sono messaggi che possono entrare in conflitto, il gestore di priorità fornisce un singolo segnale di selezione per il messaggio che ha priorità maggiore. Tale segnale di selezione, oltre a definire quale sorgente e quale destinazione considerare, verrà utilizzato come selettore di una rete composta da un MUX 4:1 e un DEMUX 4:1 per ottenere in uscita il messaggio che deve essere instradato tra i due nodi abilitati. Il modulo costituito da gestore di priorità e dalla rete appena vista costituisce la nostra **unità di controllo**. Complessivamente, la

rete di interconnessione da noi progettata è realizzata per composizione, interconnettendo l'unità operativa e quella di controllo. Possiamo visualizzare lo schematico della rete in Figura 7.3.

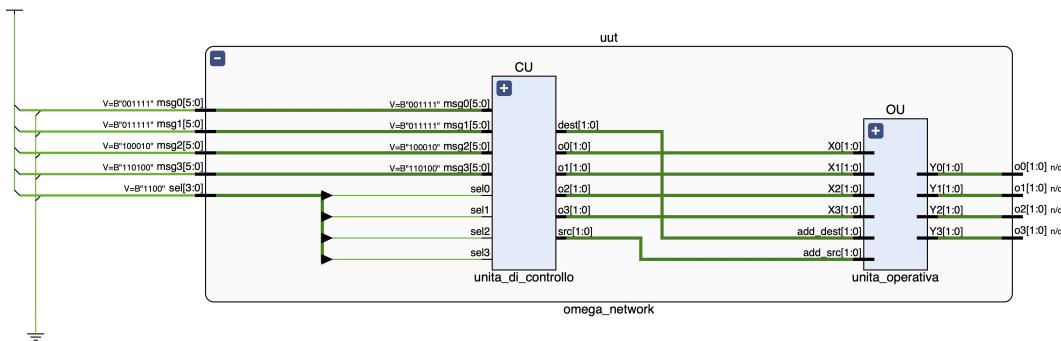


Figura 7.3: Schematic rete di interconnessione

7.1.3 Implementazione

Partiamo innanzitutto dal visualizzare le macchine combinatorie elementari che sono state realizzate per costruire i moduli più complessi, cioè il MUX 2:1 che è stato implementato secondo l'approccio dataflow e il DEMUX 1:2 che invece è stato realizzato in maniera behavioral.

```

ENTITY mux_2_1 IS
PORT (
    a0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    a1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    s : IN STD_LOGIC;
    y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END mux_2_1;
    
```

```

ARCHITECTURE dataflow OF mux_2_1 IS
BEGIN
    y <= a0 WHEN s = '0' ELSE
    a1 WHEN s = '1' ELSE
    "--";
END dataflow;

ENTITY demux_1_2 IS
PORT (
    d : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    s : IN STD_LOGIC;
    y1, y2 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END demux_1_2;

ARCHITECTURE behavioral OF demux_1_2 IS
BEGIN
    PROCESS (d, s) IS BEGIN
        IF (s = '0') THEN
            y1 <= d;
            y2 <= "00";
        ELSIF (s = '1') THEN
            y2 <= d;
            y1 <= "00";
        END IF;
    END PROCESS;
END behavioral;

```

Successivamente, sono stati realizzato il MUX 4:1 secondo l'ap-

proccio structural (per composizione di tre MUX 2:1) e il DEMUX 1:4 secondo l'approccio behavioral.

```
ENTITY mux_4_1 IS
PORT (
b0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
b1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
b2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
b3 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
s0 : IN STD_LOGIC;
s1 : IN STD_LOGIC;
y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END mux_4_1;
```

```
ARCHITECTURE structural OF mux_4_1 IS
SIGNAL u0 : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL u1 : STD_LOGIC_VECTOR(1 DOWNTO 0);
```

```
COMPONENT mux_2_1
PORT (
a0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
a1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
s : IN STD_LOGIC;
y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END COMPONENT;
```

BEGIN

```
mux0 : mux_2_1
```

```
POR T MAP (
```

```
 a0 => b0,
```

```
 a1 => b1,
```

```
 s => s0,
```

```
 y => u0
```

```
) ;
```

```
mux1 : mux_2_1
```

```
POR T MAP (
```

```
 a0 => b2,
```

```
 a1 => b3,
```

```
 s => s0,
```

```
 y => u1
```

```
) ;
```

```
mux2 : mux_2_1
```

```
POR T MAP (
```

```
 a0 => u0,
```

```
 a1 => u1,
```

```
 s => s1,
```

```
 y => y
```

```
) ;
```

```
END structural;
```

```
ENTITY demux_1_4 IS
```

```
POR T (
```

```
 a : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
```

```
 s : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
```

```
y0, y1, y2, y3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END demux_1_4;
```

```
ARCHITECTURE behavioral OF demux_1_4 IS
```

```
BEGIN
```

```
PROCESS (a, s) IS
```

```
BEGIN
```

```
IF (s = "00") THEN
```

```
    y0 <= a;
```

```
    y1 <= "00";
```

```
    y2 <= "00";
```

```
    y3 <= "00";
```

```
ELSIF (s = "01") THEN
```

```
    y1 <= a;
```

```
    y0 <= "00";
```

```
    y2 <= "00";
```

```
    y3 <= "00";
```

```
ELSIF (s = "10") THEN
```

```
    y2 <= a;
```

```
    y0 <= "00";
```

```
    y1 <= "00";
```

```
    y3 <= "00";
```

```
ELSE
```

```
    y3 <= a;
```

```
    y0 <= "00";
```

```
    y1 <= "00";
```

```

y2 <= "00";
END IF;
END PROCESS;

END behavioral;

```

Il gestore di priorità è stato realizzato secondo un approccio dataflow, in modo tale da abilitare un solo messaggio sulla base della priorità decrescente valutata sulle linee di selezione in ingresso.

```

ENTITY gestore_priorita IS
  PORT (
    sel0, sel1, sel2, sel3 : IN STD_LOGIC;
    output : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
  );
END gestore_priorita;

ARCHITECTURE dataflow OF gestore_priorita IS

BEGIN
  output <= "00" WHEN sel0 = '1' ELSE
    "01" WHEN sel1 = '1' ELSE
    "10" WHEN sel2 = '1' ELSE
    "11" WHEN sel3 = '1' ELSE
    "--";
END dataflow;

```

Lo switch è stato realizzato per composizione di un MUX 2:1 e un DEMUX 1:2, secondo un approccio structural; in ingresso riceve i

messaggi da instradare provenienti da due sorgenti diverse e gli indirizzi di sorgente e destinazione che regolano rispettivamente il MUX e il DEMUX per abilitare l'ingresso e l'uscita corretti, dato che anche le linee di uscita collegano lo switch a due destinazioni differenti.

```
ENTITY switch IS
  PORT (
    add_src : IN STD_LOGIC;
    add_dest : IN STD_LOGIC;
    X1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    X2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    Y1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
    Y2 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
  );
END switch;
```

```
ARCHITECTURE structural OF switch IS
```

```
COMPONENT mux_2_1 IS
  PORT (
    a0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    a1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    s : IN STD_LOGIC;
    y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
  );
END COMPONENT;
```

```
COMPONENT demux_1_2 IS
```

```

PORT (
      d : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
      s : IN STD_LOGIC;
      y1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
      y2 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);

END COMPONENT;

SIGNAL temp : STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN

mux : mux_2_1
PORT MAP (
      a0 => X1,
      a1 => X2,
      s => add_src,
      y => temp
);

demux : demux_1_2
PORT MAP (
      d => temp,
      s => add_dest,
      y1 => Y1,
      y2 => Y2
);

```

```
END structural;
```

Ancora per composizione abbiamo ottenuto l'unità operativa utilizzando quattro switch che sono stati disposti su due layer che rappresentano gli stadi intermedi della rete, come abbiamo già spiegato in precedenza.

```
ENTITY unita_operativa IS
    PORT (
        X0, X1, X2, X3 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        add_src, add_dest : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        Y0, Y1, Y2, Y3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
    );
END unita_operativa;

ARCHITECTURE structural OF unita_operativa IS

COMPONENT switch IS
    PORT (
        add_src : IN STD_LOGIC;
        add_dest : IN STD_LOGIC;
        X1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        X2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        Y1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        Y2 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
    );
END COMPONENT;

SIGNAL link0, link1, link2, link3 : STD_LOGIC_VECTOR(1 DOWNTO 0);
```

BEGIN

```
S0 : switch
PORT MAP (
    add_src => add_src(1),
    add_dest => add_dest(1),
    X1 => X0,
    X2 => X2,
    Y1 => link0,
    Y2 => link2
);
```

```
S1 : switch
PORT MAP (
    add_src => add_src(1),
    add_dest => add_dest(1),
    X1 => X1,
    X2 => X3,
    Y1 => link1,
    Y2 => link3
);
```

```
S2 : switch
PORT MAP (
    add_src => add_src(0),
    add_dest => add_dest(0),
    X1 => link0,
```

```

        X2 => link1,
        Y1 => Y0,
        Y2 => Y1
    ) ;

S3 : switch
PORT MAP (
    add_src => add_src(0),
    add_dest => add_dest(0),
    X1 => link2,
    X2 => link3,
    Y1 => Y2,
    Y2 => Y3
) ;

END structural;

```

L'unità di controllo è stata ottenuta anch'essa in maniera structural sfruttando il gestore di priorità, un MUX 4:1 e un DEMUX 1:4.

```

ENTITY unita_di_controllo IS
PORT (
    msg0, msg1, msg2, msg3 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    sel0, sel1, sel2, sel3 : IN STD_LOGIC;
    o0, o1, o2, o3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
    src, dest : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END unita_di_controllo;

```

```
ARCHITECTURE structural OF unita_di_controllo IS
```

```
COMPONENT mux_4_1 IS
  PORT (
    b0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    b1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    b2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    b3 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    s0 : IN STD_LOGIC;
    s1 : IN STD_LOGIC;
    y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
  );
END COMPONENT;
```

```
COMPONENT demux_1_4 IS
  PORT (
    a : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    s : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    y0, y1, y2, y3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
  );
END COMPONENT;
```

```
COMPONENT gestore_priorita IS
  PORT (
    sel0, sel1, sel2, sel3 : IN STD_LOGIC;
    output : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
  );
END COMPONENT;
```

```

SIGNAL temp_sel, temp_link : STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN

    gestore : gestore_priorita
    PORT MAP (
        sel0 => sel0,
        sel1 => sel1,
        sel2 => sel2,
        sel3 => sel3,
        output => temp_sel
    );

    mux : mux_4_1
    PORT MAP (
        b0 => msg0(1 DOWNTO 0),
        b1 => msg1(1 DOWNTO 0),
        b2 => msg2(1 DOWNTO 0),
        b3 => msg3(1 DOWNTO 0),
        s0 => temp_sel(0),
        s1 => temp_sel(1),
        y => temp_link
    );

    demux : demux_1_4
    PORT MAP (
        a => temp_link,

```

```

        s => temp_sel,
        y0 => o0,
        y1 => o1,
        y2 => o2,
        y3 => o3
    );
}

src <= msg0(5 DOWNTO 4) WHEN temp_sel = "00" ELSE
    msg1(5 DOWNTO 4) WHEN temp_sel = "01" ELSE
    msg2(5 DOWNTO 4) WHEN temp_sel = "10" ELSE
    msg3(5 DOWNTO 4) WHEN temp_sel = "11" ELSE
    "00";

dest <= msg0(3 DOWNTO 2) WHEN temp_sel = "00" ELSE
    msg1(3 DOWNTO 2) WHEN temp_sel = "01" ELSE
    msg2(3 DOWNTO 2) WHEN temp_sel = "10" ELSE
    msg3(3 DOWNTO 2) WHEN temp_sel = "11" ELSE
    "00";

END structural;

```

La rete di interconnessione di tipo omega network è stata infine implementata per composizione delle due unità viste. Gli ingressi del sistema complessivo sono i messaggi provenienti dai quattro nodi e il segnale di selezione dei messaggi, che vanno in ingresso all'unità di controllo. Questa unità elabora tali input e fornisce in output il messaggio che verrà instradato e gli indirizzi dei nodi sorgente e destinazione che sono stati abilitati per la comunicazione. Tali output sono conservati

in appositi segnali di appoggio che vanno in ingresso all'unità operativa (realizzando così il collegamento tra le due unità), che si occupa effettivamente di instradare il messaggio, riportandolo poi in uscita al sistema complessivo che realizza la rete.

```
ENTITY omega_network IS
  PORT (
    -- 2 bit sorgente + 2 bit destinazione + 2 bit informazione
    msg0 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    msg1 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    msg2 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    msg3 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    sel : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    o0, o1, o2, o3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
  );
END omega_network;
```

```
ARCHITECTURE structural OF omega_network IS

COMPONENT unita_operativa IS
  PORT (
    X0, X1, X2, X3 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    add_src, add_dest : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    Y0, Y1, Y2, Y3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
  );
END COMPONENT;
```

```
COMPONENT unita_di_controllo IS
```

```

PORT (
    msg0, msg1, msg2, msg3 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    sel0, sel1, sel2, sel3 : IN STD_LOGIC;
    o0, o1, o2, o3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
    src, dest : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);

END COMPONENT;

SIGNAL temp_src, temp_dest : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL temp_link0, temp_link1, temp_link2, temp_link3 : STD_LOGIC_


BEGIN

    OU : unita_operativa
    PORT MAP (
        temp_link0, temp_link1, temp_link2, temp_link3,
        temp_src, temp_dest, o0, o1, o2, o3
    );

    CU : unita_di_controllo
    PORT MAP (

        msg0, msg1, msg2, msg3,
        sel(0), sel(1), sel(2), sel(3),
        temp_link0, temp_link1, temp_link2, temp_link3,
        temp_src, temp_dest
    );

END structural;

```

7.1.4 Simulazione

Il testbench che abbiamo preparato simula 5 diverse richieste di comunicazione, in particolare nella prima, nella seconda e nella quinta sono due le sorgenti a voler inviare un messaggio, nella terza una sola sorgente vuole comunicare e infine nella quarta tutti i nodi vogliono mandare il proprio messaggio.

```
ENTITY omega_network_tb IS
END;

ARCHITECTURE bench OF omega_network_tb IS

COMPONENT omega_network
PORT (
    msg0 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    msg1 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    msg2 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    msg3 : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    sel : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    o0, o1, o2, o3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END COMPONENT;

SIGNAL msg0 : STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL msg1 : STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL msg2 : STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL msg3 : STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL sel : STD_LOGIC_VECTOR(3 DOWNTO 0);
```

```

SIGNAL o0, o1, o2, o3 : STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN

    uut : omega_network PORT MAP (
        msg0 => msg0,
        msg1 => msg1,
        msg2 => msg2,
        msg3 => msg3,
        sel => sel,
        o0 => o0,
        o1 => o1,
        o2 => o2,
        o3 => o3);

stimulus : PROCESS
BEGIN

    sel <= "1010"; --vogliono trasmettere 1 e 3
    msg0 <= "001011";
    msg1 <= "011111"; --dovrei vedere 11 sull'uscita 3
    msg2 <= "100010";
    msg3 <= "110100";
    WAIT FOR 10 ns;
    sel <= "0101"; --vogliono trasmettere 0 e 2
    msg0 <= "001001"; --dovrei vedere 01 sull'uscita 2
    msg1 <= "011111";
    msg2 <= "100010";
    msg3 <= "110100";

```

```
WAIT FOR 10 ns;

sel <= "1000"; --vuole trasmettere 3
msg0 <= "001011";
msg1 <= "011111";
msg2 <= "100010";
msg3 <= "110110"; --dovrei vedere 10 sull'uscita 1
WAIT FOR 10 ns;
sel <= "1111"; --vogliono trasmettere tutti
msg0 <= "001111"; --dovrei vedere 11 sull'uscita 3
msg1 <= "011111";
msg2 <= "100010";
msg3 <= "110100";
WAIT FOR 10 ns;
sel <= "1100"; --vogliono trasmettere 2 e 3
msg0 <= "001111";
msg1 <= "011111";
msg2 <= "100010"; --dovrei vedere 10 sull'uscita 0
msg3 <= "110100";
WAIT FOR 10 ns;
WAIT;
END PROCESS;

END;
```

L'output della simulazione è visualizzabile in Figura 7.4, dove abbiamo evidenziato gli esiti del quinto gruppo di messaggi. Dal segnale di selezione *sel*, che assume il valore 1100, possiamo capire che i nodi che vogliono comunicare sono 2 e 3. Il nodo 2 vuole trasmettere il mes-

saggio 10 al nodo 0, mentre il nodo 3 vuole trasmettere il messaggio 00 al nodo 1. Poiché 2 ha la priorità su 3, in output osserveremo che il segnale o_0 , cioè la linea di uscita riservata al nodo destinazione 0, contiene proprio il valore 10.

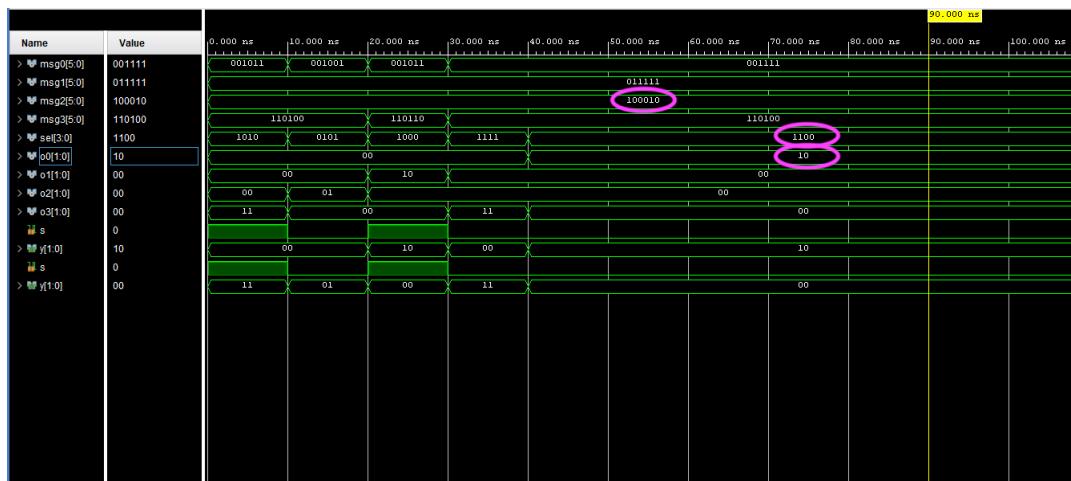


Figura 7.4: Simulazione omega network

Capitolo 8

Appendice

8.1 Multiplexer 2:1

8.1.1 Progetto e architettura

Il **Multiplexer 2:1** è un componente che presenta tre ingressi e un'uscita: due dei tre segnali di ingresso sono per i dati effettivi, mentre attraverso il terzo segnale di ingresso, che funge da **selettore**, è possibile selezionare uno dei due ingressi per riportarlo in uscita. In Figura 8.1 è riportato lo schematic del MUX 2.1.

8.1.2 Implementazione

La prima operazione è stata definire l'**entity mux_2_1**: abbiamo dichiarato 3 ingressi, due per l'input e uno per il selettore, e un segnale per l'output. L'interfaccia del componente è mostrata di seguito.

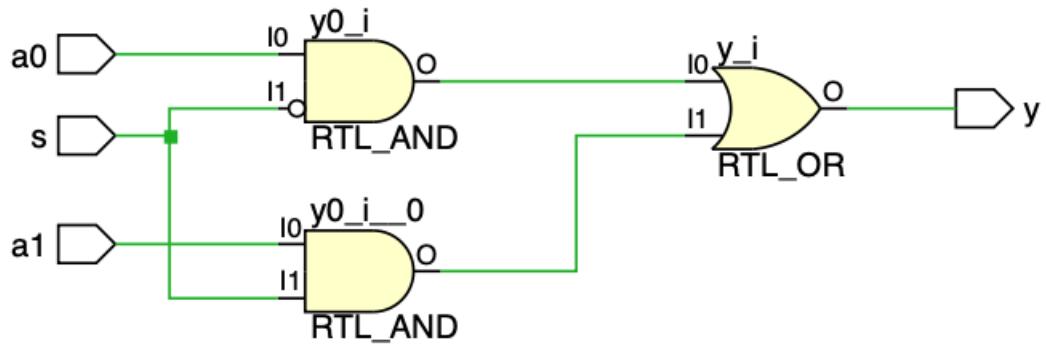


Figura 8.1: Architettura di multiplexer 2:1

```

ENTITY mux_2_1 IS
PORT (
    a0 : IN STD_LOGIC;
    a1 : IN STD_LOGIC;
    s : IN STD_LOGIC;
    y : OUT STD_LOGIC
);
END mux_2_1;

```

Successivamente abbiamo sviluppato il multiplexer 2:1 con un'architettura *dataflow* al quale assegnamo il valore dell'uscita mediante una funzione logica nella quale se il selettore assume valore $s=0$ all'uscita viene assegnato il valore $a0$ altrimenti viene assegnato il valore $a1$. Dii seguito è riportata l'implementazione.

```

ARCHITECTURE dataflow OF mux_2_1 IS
BEGIN
    y <= ((a0 AND (NOT s)) OR (a1 AND s));
END dataflow;

```

8.2 Multiplexer 4:1

8.2.1 Progetto e architettura

Il Multiplexer 4:1 è una macchina combinatoria che presenta 4 ingressi e una sola uscita, anche in questo caso è stato realizzato come un MUX **indirizzabile**, oltre ai 4 ingressi abbiamo quindi due selettori che sono necessari per scegliere quale dei quattro ingressi avere in uscita. Per realizzare il **MUX 4:1** sono necessari 3 multiplexer 2:1. Ciascuno dei primi due multiplexer prende in entrata 2 dei 4 ingressi e produce un'uscita in base ad uno dei due ingressi di selezione del MUX 4:1. Il secondo segnale di selezione sarà usato per gestire l'uscita del terzo MUX 2:1. La struttura può essere vista di seguito.

```
ENTITY mux_4_1 IS
PORT (
    b : IN STD_LOGIC_VECTOR(0 TO 3);
    r : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    u : OUT STD_LOGIC
);
END mux_4_1;

ARCHITECTURE structural OF mux_4_1 IS
SIGNAL temp : STD_LOGIC_VECTOR(0 TO 1);

COMPONENT mux_2_1
PORT (
    a0 : IN STD_LOGIC;
```

```

a1 : IN STD_LOGIC;
s : IN STD_LOGIC;
y : OUT STD_LOGIC
);
END COMPONENT;

```

In Figura 8.2 è riportato lo schematic del MUX 4:1.

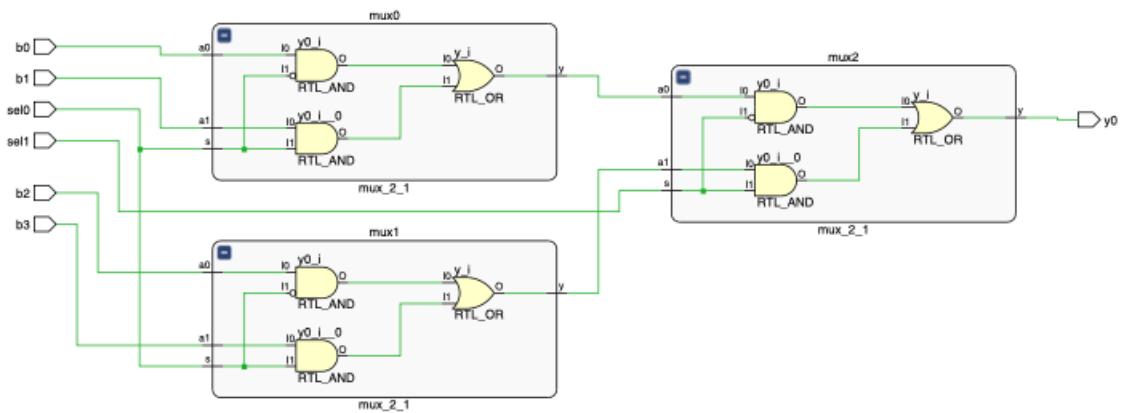


Figura 8.2: Architettura di MUX 4:1

8.2.2 Implementazione

Per il multiplexer 4:1 abbiamo utilizzato un approccio per composizione utilizzando il multiplexer 2:1 già implementato. Il MUX 4:1 presenta i 6 ingressi ($b0, b1, b2, b3, s0, s1$) definiti come std_logic e 1 uscita ($y0$) anch'essa definita con std_logic. Siccome i componenti necessari sono già stati creati precedentemente abbiamo potuto usare un approccio di tipo strutturale, istanziando i tre multiplexer e realizzando opportunamente il collegamento tra le loro linee di ingresso

e uscita; in particolare, usiamo i segnali temp come segnali di appoggio per collegare le uscite dei primi due multiplexer con il multiplexer finale.

```
BEGIN

mux0 : mux_2_1
PORT MAP (
    a0 => b(0),
    a1 => b(1),
    s => r(0),
    y => temp(0)
);

mux1 : mux_2_1
PORT MAP (
    a0 => b(2),
    a1 => b(3),
    s => r(0),
    y => temp(1)
);

mux2 : mux_2_1
PORT MAP (
    a0 => temp(0),
    a1 => temp(1),
    s => r(1),
    y => u
);
END structural;
```

8.3 Demultiplexer 1:2

8.3.1 Progetto e architettura

Il **Demultiplexer 1:2** è una macchina combinatoria che presenta un ingresso e due uscite. Abbiamo deciso di implementare un DEMUX **indirizzabile**, dunque usiamo un selettore che ci permette di scegliere su quale linea di uscita pilotare l'ingresso. In Figura 8.3 è riportato lo schema del DEMUX 1:2.

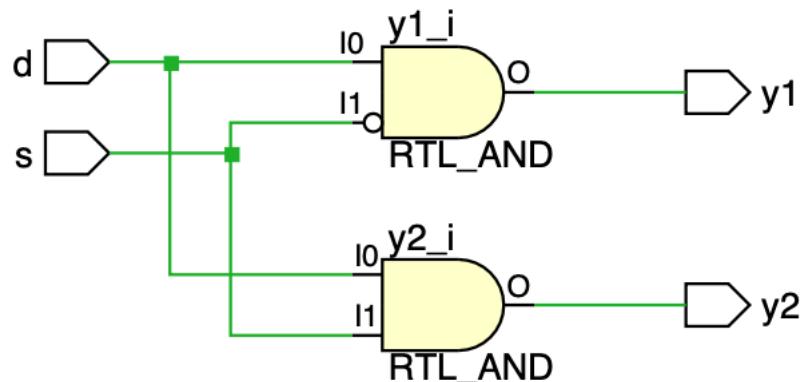


Figura 8.3: Architettura del DEMUX 1:2

8.3.2 Implementazione

Di seguito è riportata l'implementazione del Demultiplexer realizzato attraverso un approccio *dataflow*.

```
ENTITY demux_1_2 IS
  PORT (
    d : IN STD_LOGIC;
    s : IN STD_LOGIC;
    y1 : OUT STD_LOGIC;
    y2 : OUT STD_LOGIC
  );
END demux_1_2;

ARCHITECTURE dataflow OF demux_1_2 IS

BEGIN
  y1 <= d AND (NOT s);
  y2 <= d AND s;

END dataflow;
```

8.4 Contatore modulo N

8.4.1 Progetto e architettura

Il contatore modulo N è un dispositivo che permette di effettuare un conteggio da 0 a $N-1$ e la velocità (cioè la frequenza) del conteggio è misurata in termini di colpi di clock al secondo. In Figura 8.4 possiamo osservare uno schema del contatore.

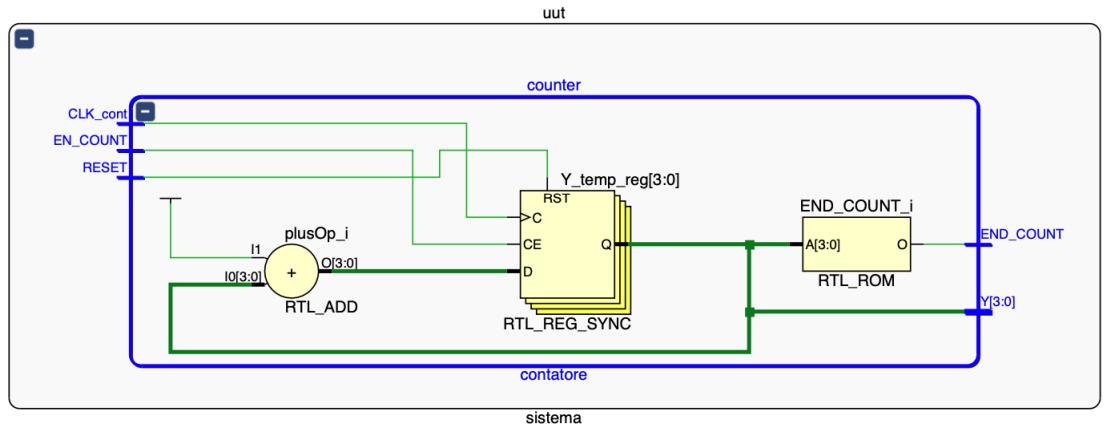


Figura 8.4: Architettura di un contatore modulo N

8.4.2 Implementazione

Nell'entity del contatore innanzitutto abbiamo inserito un generic N per scegliere il modulo del conteggio; abbiamo poi posto in ingresso il segnale di *clock*, il *reset* per azzerare il contatore, *en_count* per abilitare il conteggio al momento opportuno ed *end_count* che si alza nel momento in cui il conteggio è terminato; quest'ultimo è usato per avvertire altri componenti della terminazione dell'operazione. Come uscita abbiamo chiaramente il valore corrente del conteggio Y, espresso su N bit. Il comportamento del contatore è realizzato mediante un process sensibile al segnale di clock, in tal modo il contatore effettua le operazioni solo sul fronte di salita del clock. Se il segnale di reset è alto si assegna all'uscita temporanea il valore 0, altrimenti si ricontrolla che l'abilitazione è alta per riassegnare all'uscita temporanea il valore precedente, maggiorato di uno. Al termine del processo di assegna all'uscita il segnale temporaneo e se il valore di conteggio è uguale a

$2^N - 1$ si alza anche il segnale di *end_count*.

```
ENTITY cont_mod_N IS
  GENERIC (
    N : POSITIVE := 3
  );
  PORT (
    CLK_cont : IN STD_LOGIC;
    RESET : IN STD_LOGIC := '0';
    EN_COUNT : IN STD_LOGIC;
    END_COUNT : OUT std_logic;
    Y : OUT STD_LOGIC_VECTOR(N - 1 DOWNTO 0)
  );
END cont_mod_N;

ARCHITECTURE Behavioral OF cont_mod_N IS

  SIGNAL Y_temp : STD_LOGIC_VECTOR(N - 1 DOWNTO 0) := (OTHERS => '0');

BEGIN
  count : PROCESS (CLK_cont)
  BEGIN
    IF rising_edge(CLK_cont) THEN
      IF (RESET = '1') THEN
        Y_temp <= (OTHERS => '0');
      ELSE
        IF (EN_COUNT = '1') THEN
          Y_temp <= STD_LOGIC_VECTOR(unsigned(Y_temp) + 1);
        END IF;
      END IF;
    END IF;
  END PROCESS;
END;
```

```
    END IF;  
  
    END IF;  
  
END IF;  
  
  
END PROCESS;  
  
Y <= Y_temp;  
  
END_COUNT <= '1' when (to_integer(unsigned(Y_temp)) = 2**N)  
else '0';  
  
END Behavioral;
```

8.5 ROM sequenziale

8.5.1 Progetto e architettura

La ROM è un componente che permette solo di leggere i dati che sono contenuti al suo interno ma non di scriverli. La ROM presenta in ingresso 4 segnali:

- *CLK*: per la sincronizzazione delle operazioni;
- *address*: stringa di bit in ingresso usata per accedere al contenuto di una locazione di memoria;
- *read*: segnale di abilitazione per la lettura dei dati contenuti nella ROM;
- *dout*: segnale che riporta in uscita i valori contenuti nella ROM.

Lo schema è riportato in Figura 8.5:

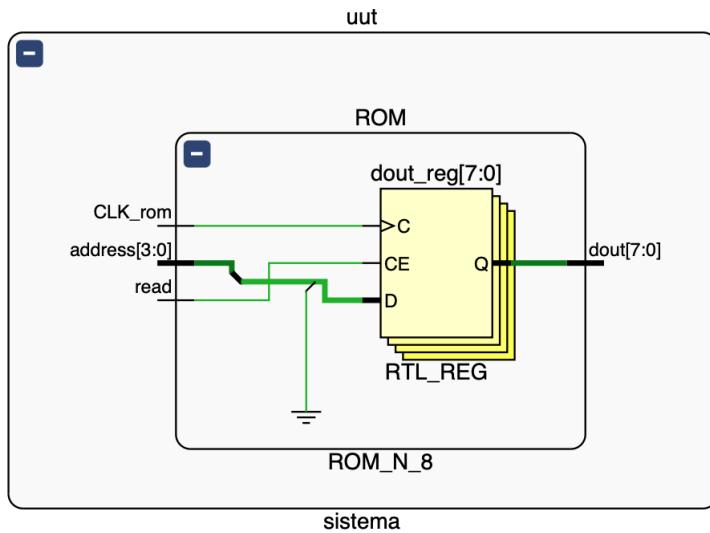


Figura 8.5: Architettura di una ROM con dimensionamento 16x8

8.5.2 Implementazione

Come riportato di seguito è presente anche un elemento di GENERIC che viene utilizzato per indicare il numero di bit su cui è codificato l'indirizzo e per indicare la lunghezza della ROM.

```
ARCHITECTURE Behavioral of ROM_8_8 is
```

```
CONSTANT N : positive := 2**len_add;
```

```
TYPE MEMORY_N_8 IS ARRAY (0 to N-1) OF std_logic_vector(7 downto 0);
```

```
constant ROM_N_8 : MEMORY_N_8 := (
    "00000000",
    "00000001",
    "00000010",
    "00000011",
    "00000100",
    "00000101",
```

```
"00000110",
"00000111"
);

BEGIN

main : process(CLK_rom)
begin
    if rising_edge(CLK_rom) then
        if(read = '1') then
            dout <= ROM_N_8(to_integer(unsigned(address)));
        end if;
    end if;

end process main;
END Behavioral;
```

Come si può osservare nel codice la ROM_N_8 è realizzata come un array dove il numero di elementi dipende dal GENERIC mentre la lunghezza in bit dei dati è impostata su 8 valori. Si accede all'apposito elemento effettuando una conversione dell'address ad integer.

8.6 Memoria

8.6.1 Progetto e architettura

La MEMORIA presenta una struttura simile a quella della ROM ma oltre alla lettura si può effettuare anche la scrittura. Anche in que-

sto caso abbiamo usato un GENERIC per impostare la lunghezza dell'address e della memoria.

8.6.2 Implementazione

Come si può osservare in figura 8.6 nell'entity sono stati dichiarati i seguenti segnali:

- **CLK_mem**: è il clock che viene dato alla memoria per eseguire le operazioni;
- *write*: Segnale di abilitazione per effettuare la scrittura sulla memoria;
- *address*: segnale usato per passare la locazione dove effettuare la lettura o la scrittura;
- *input_val*: il valore che si vuole scrivere in memoria;
- *out_val*: Il valore letto dalla memoria e restituito in uscita.

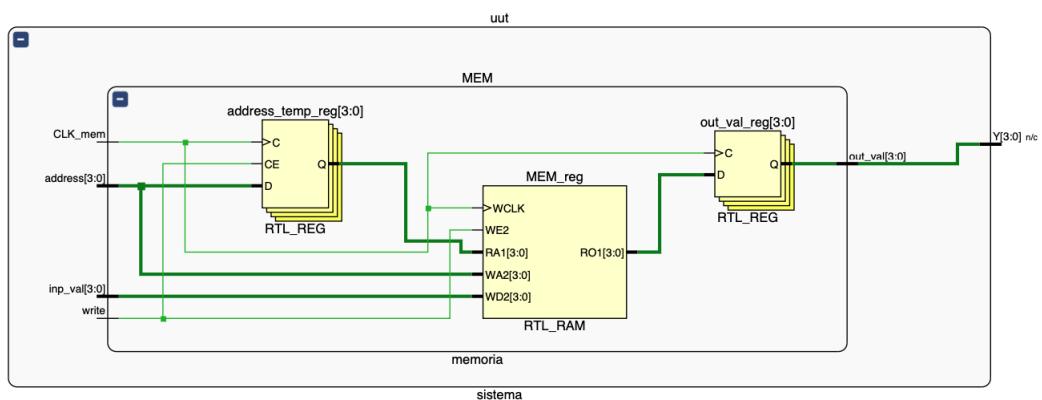


Figura 8.6: Architettura di una memoria 16x8

```
entity memoria is
    Generic(
        len_add : positive := 4
    );
    Port (
        CLK_mem : in std_logic;
        write : in std_logic;
        address : in std_logic_vector (len_add-1 downto 0);
        inp_val : in std_logic_vector(3 downto 0);
        out_val : out std_logic_vector(3 downto 0)
    );
end memoria;
```

Per l'implementazione della memoria abbiamo istanziato un segnale di appoggio *address_temp* che memorizza l'ultima locazione interessata da un'operazione di scrittura. Abbiamo realizzato poi un process sensibile al fronte di salita del clock per descrivere il comportamento del componente.

```
architecture Behavioral of memoria is
    CONSTANT N : positive := 2**len_add;
    signal address_temp : std_logic_vector(len_add-1 downto 0);

    type MEMORY_N_4 is array (0 to N-1) of std_logic_vector(len_add-1 do
    signal MEM : MEMORY_N_4;
begin
    process (CLK_mem)
        begin

```

```
if rising_edge(CLK_mem) then
    if(write = '1') then
        MEM(to_integer(unsigned(address))) <= inp_val;
        address_temp <= address;
    end if;
    out_val <= MEM(to_integer(unsigned(address_temp)));
end if;
end process;
end Behavioral;
```

8.7 Button Debouncer

8.7.1 Progetto e architettura

Il Button Debouncer è un componente che "ripulisce" il segnale in input dal bottone, in modo da presentare in uscita un impulso della durata di un colpo di clock per segnalare l'avvenuta pressione del bottone. Il debouncing dei pulsanti è una tecnica comunemente utilizzata nella progettazione di circuiti digitali per garantire che la singola pressione di un pulsante o di un interruttore produca un unico segnale di uscita. Senza il debouncing, la natura meccanica di questi pulsanti o interruttori può far sì che il segnale di uscita "rimbalzi" rapidamente tra gli stati on e off prima di stabilizzarsi, con conseguenti comportamenti imprevisti.

8.7.2 Implementazione

Per realizzare il Button Debouncer è stato creato un process sensibile al solo segnale di clock, attivo sul suo fronte di salita; all'interno di questo process abbiamo definito le transizioni dell'automa con cui è possibile modellare il componenete. Gli stati possibili della macchina sono i seguenti:

- NOT_PRESSED: se il pulsante (BTN) è alto, lo stato del pulsante viene impostato su CHK_PRESSED, indicando che la pressione del pulsante viene verificata per il rimbalzo. Se il pulsante non è alto, lo stato del pulsante rimane NOT_PRESSED;
- CHK_PRESSED: un contatore viene controllato rispetto al conteggio calcolato a partire dalle costanti dichiarate come generics. Se il contatore ha raggiunto questo massimo e il pulsante è ancora alto, si presume che la pressione del pulsante non sia un rimbalzo. Il contatore viene azzerato, e viene generato l'impulso alto ripulito e lo stato del pulsante viene impostato su PRESSED. Se a questo punto il pulsante non è alto, lo stato del pulsante viene riportato a NOT_PRESSED;
- PRESSED: Viene abbassata l'uscita per garantire che il segnale sia alto per un solo impulso di clock. Se il pulsante è basso, lo stato del pulsante viene impostato su CHK_NOT_PRESSED, indicando che il rilascio del pulsante viene controllato per evitare

il rimbalzo. Se il pulsante è ancora alto, lo stato del pulsante rimane PRESSED;

- **CHK_NOT_PRESSED**: simile a **CHK_PRESSED**, il contatore viene controllato rispetto al conteggio massimo. Se il contatore ha raggiunto il massimo e il pulsante è ancora basso, si presume che il rilascio del pulsante non sia un rimbalzo. Il contatore viene azzerato e lo stato del pulsante viene impostato su NOT_PRESSED. Se il pulsante non è ancora basso a questo punto, lo stato del pulsante viene riportato a PRESSED.

Di seguito è riportato il codice VHDL di una FSM che implementa il meccanismo suddetto.

```
ENTITY Debouncer IS
  GENERIC (
    CLK_period: integer := 10; -- periodo del clock (della board)
    btn_noise_time: integer := 10000000 -- durata stimata dell'onda
                                         -- il valore di default
  );
  PORT ( RST : in STD_LOGIC;
         CLK : in STD_LOGIC;
         BTN : in STD_LOGIC;
         CLEARED_BTN : out STD_LOGIC);
END Debouncer;

ARCHITECTURE Behavioral OF Debouncer IS
```

```

-- questo componente prende in input il segnale proveniente dal bottone
-- segnale "ripulito" che presenta un impulso della durata di un colpo
-- segnalare l'avvenuta pressione del bottone.

-- Il debouncer implementa un semplice automa di 4 stati:
-- si parte da NOT_PRESSED e, appena si rileva BTN=1, si va in CHK_PRESSED;
-- si attende un certo tempo in modo da "superare" l'oscillazione: se dopo
-- si alza il segnale ripulito in output (avrà la durata di un impulso)
-- e si va in PRESSED; anche qui quando si rileva BTN=0
-- si va in uno stato intermedio CHK_NOT_PRESSED in cui si aspetta un'altra
-- l'oscillazione: se dopo questo tempo è ancora basso si ritorna in NOT_PRESSED.

-- con questo automa se si mantiene il bottone premuto non vengono generate
-- uscite

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

CONSTANT max_count : integer := btn_noise_time/CLK_period; -- 1000000;

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 =>

    IF( BTN = '1' ) THEN

        BTN_state <= CHK_PRESSED;

    ELSE

        BTN_state <= NOT_PRESSED;

    END IF;

    WHEN CHK_PRESSED =>

        IF(count = max_count -1) THEN

            IF(BTN = '1') THEN --se arrivo a count max ed è

                count:=0;

                CLEARED_BTN <= '1';

                BTN_state <= PRESSED;

            ELSE

                count:=0;

                BTN_state <= NOT_PRESSED;

            end if;

        else

            count:= count+1;

            BTN_state <= CHK_PRESSED;

        end if;

    when PRESSED =>

        CLEARED_BTN<= '0'; --questo lo metto per fare in mod

        if(BTN = '0') then

```

```

    BTN_state <= CHK_NOT_PRESSED;

else

    BTN_state <= PRESSED;

end if;

when CHK_NOT_PRESSED =>

    if(count = max_count -1) then

        if(BTN = '0') then --se arrivo a count max ed è

            count:=0;

            BTN_state <= NOT_PRESSED;

        else

            count:=0;

            BTN_state <= PRESSED;

        end if;

    else

        count:= count+1;

        BTN_state <= CHK_NOT_PRESSED;

    end if;

when others =>

    BTN_state <= NOT_PRESSED;

end case;

end if;

end if;

end process;

end Behavioral;

```