



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

*Elaborato Finale di Architettura
dei Sistemi Digitali*

Anno Accademico 2022/2023

Gruppo 11:

Francesco Pio Manna M63001485
Claudio Pisa M63001512

Indice

1 Reti combinatorie elementari	1
1.1 Traccia	1
1.1.1 Esercizio 1.1	1
1.1.2 Esercizio 1.2	1
1.1.3 Esercizio 1.3	2
1.2 Esercizio 1.1 – Multiplexer	2
1.2.1 Descrizione generale, schemi e componenti utilizzati	2
1.2.2 Testing del sistema	8
1.3 Esercizio 1.2 - Rete di interconnessione 16:4	10
1.3.1 Descrizione e componenti utilizzati	10
1.3.2 Realizzazione del sistema complessivo	13
1.3.3 Testing del sistema complessivo	15
1.4 Esercizio 1.3 – Implementazione sulla board della rete 16:4	16
1.5 Esercizio 2.1 - Encoder BCD	19
1.5.1 Traccia	19
1.5.2 Descrizione della soluzione	19
1.5.3 Testing simulato	21

1.6	Esercizio 2.2 - Encoder con LED	22
1.7	Esercizio 2.3 - Encoder con display a sette segmenti . .	24
2	Reti sequenziali elementari	29
2.1	Traccia	29
2.1.1	Esercizio 3.1	29
2.1.2	Esercizio 3.2	30
2.1.3	Esercizio 4.1	30
2.1.4	Esercizio 5.1	31
2.1.5	Esercizio 5.2	31
2.1.6	Esercizio 5.3	32
2.1.7	Esercizio 6.1	32
2.1.8	Esercizio 6.2	32
2.2	Esercizio 3.1 – Riconoscitore di sequenza	34
2.2.1	Descrizione e considerazioni sulle scelte progettuali	36
2.2.2	Testing del sistema	37
2.3	Esercizio 3.2 – Riconoscitore su board	39
2.3.1	Button debouncer	39
2.3.2	Riconoscitore di sequenza	43
2.3.3	Sistema complessivo: riconoscitore di sequenza on board	45
2.4	Esercizio 4.1 – Shift Register	47
2.4.1	Implementazione behavioral	48
2.4.2	Implementazione structural	53
2.4.3	Testing dei componenti	61
2.5	Esercizio 5.1 – Cronometro	63

2.5.1	Testing del sistema complessivo	74
2.6	Esercizio 5.2 – Cronometro onboard	74
2.6.1	Caricamento dell'orario	75
2.7	Esercizio 5.3 – Cronometro con intertempo	84
2.8	Esercizio 6.1 – Sistema di testing	88
2.8.1	Unità operativa	89
2.8.2	Unità di controllo	101
2.8.3	Sistema complessivo	107
2.8.4	Simulazione del sistema	109
2.9	Esercizio 6.2 – Sistema di testing on board	110
3	Comunicazione con handshaking	113
3.1	Traccia	113
3.1.1	Esercizio 7	113
3.2	Esercizio 7 – Sistema di handshaking	114
3.2.1	Introduzione	114
3.2.2	Unità operativa - entità A	115
3.2.3	Unità di controllo – entità A	118
3.2.4	Unità operativa – entità B	122
3.2.5	Unità di controllo – entità B	128
3.2.6	Sistema complessivo	135
4	Processore	137
4.1	Traccia	137
4.2	Esercizio 8 – Processore	138
4.2.1	Introduzione	138
4.2.2	Unità operativa	138

4.2.3	Unità di controllo	141
4.2.4	Approfondimento di alcune istruzioni	143
4.2.5	Modifica di una microistruzione	147
5	Interfaccia seriale	149
5.1	Traccia	149
5.1.1	Esercizio 9.1	149
5.1.2	Esercizio 9.2	150
5.2	Esercizio 9.1	150
5.2.1	Nodo A	150
5.2.2	Nodo B	151
5.3	Esercizio 9.2	153
5.3.1	Unità operativa – entità A	154
5.3.2	Unità di controllo – entità A	154
5.3.3	Unità operativa – entità B	159
5.3.4	Unità di controllo – entità B	160
5.3.5	Sistema complessivo	164
6	Switch multistadio	169
6.1	Traccia	169
6.2	Esercizio 10 – Omega Network	170
6.2.1	Introduzione	170
6.2.2	Unità operativa	174
6.2.3	Unità di controllo	182
6.2.4	Testing del sistema complessivo	189
7	Macchine aritmetiche	190

7.1	Traccia	190
7.2	Esercizio 11 – Moltiplicatore di Robertson	191
7.2.1	Introduzione	191
7.2.2	Unità operativa	193
7.2.3	Unità di controllo	209
7.3	Sistema complessivo	215
8	Esercizio libero	220
8.1	Traccia	220
8.2	Esercizio 12	221
8.2.1	Unità operativa – entità A	221
8.2.2	Unità di controllo – entità A	229
8.2.3	Unità operativa – entità B	234
8.2.4	Unità di controllo – entità B	240
8.2.5	Sistema complessivo	244

Capitolo 1

Reti combinatorie elementari

1.1 Traccia

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 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.

1.1.3 Esercizio 1.3

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

1.2 Esercizio 1.1 – Multiplexer

1.2.1 Descrizione generale, schemi e componenti utilizzati

Il primo esercizio richiedeva di realizzare un multiplexer 16:1 a partire da multiplexer 4:1.

Il **multiplexer** è uno tra i componenti maggiormente utilizzati nella progettazione dei sistemi digitali. Questo componente è in grado di *selezionare uno tra i vari ingressi possibili e di trasferire il dato in esso presente in uscita*. È sempre dotato di uno o più ingressi di selezione, che permettono di scegliere quale ingresso convogliare verso l'uscita. In generale, possiamo avere multiplexer *lineari* e *indirizzabili*. Nello specifico, abbiamo realizzato un multiplexer indirizzabile, che presenta un decoder (non disegnato in figura) negli ingressi di selezione che toglie la necessità di avere una linea di selezione specifica per ogni

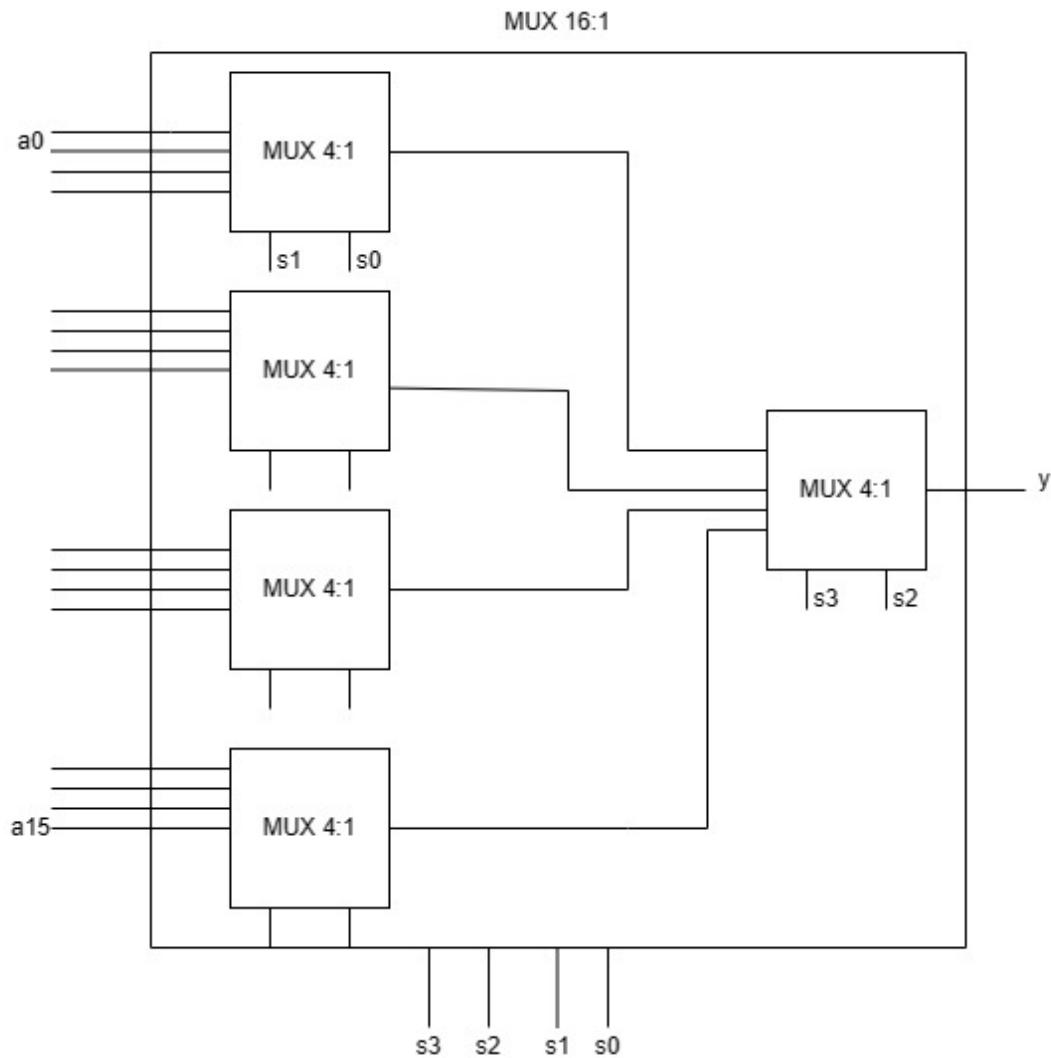


Figura 1.1: Realizzazione di un multiplexer 16:1 per composizione ingresso.

Il blocco fondamentale del multiplexer 16:1 è il multiplexer 4:1, che permette di comporre un sistema più grande (mux 16:1) a partire dalla composizione di sistemi più piccoli (mux 4:1).

Multiplexer 4:1

Come accennato in precedenza, il blocco fondamentale del nostro sistema è il multiplexer 4:1.

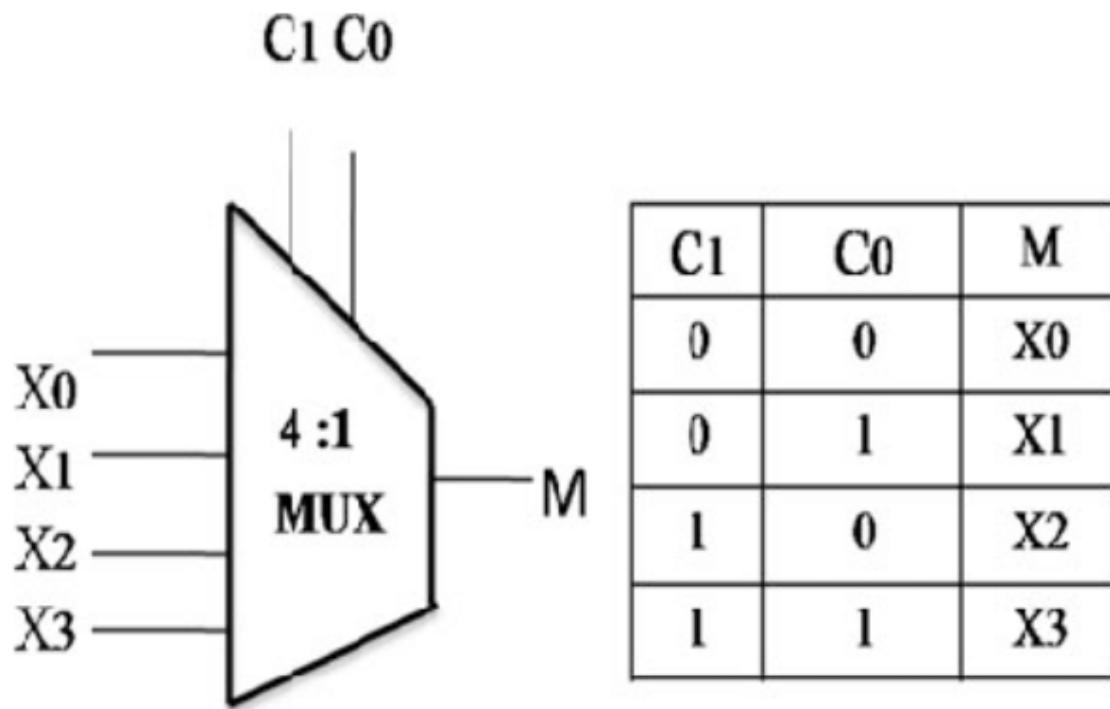


Figura 1.2: Multiplexer 4:1

Il multiplexer 4:1 è stato realizzato a un livello di astrazione RTL (Register Transfer Level), che permette di *descrivere esplicitamente le trasformazioni che i dati subiscono*. Quindi, abbiamo utilizzato il costrutto `when` e un'assegnazione condizionata.

```
1  entity mux_4_1 is
2      port(
3          input_a : in std_logic_vector(0 to 3);
4          input_s : in std_logic_vector(1 downto 0);
```

```
5      output_y : out std_logic --uscita
6  );
7 end mux_4_1;
8
9 architecture rtl of mux_4_1 is
10 begin
11     output_y <= input_a(0) when input_s = "00" else
12         input_a(1) when input_s = "01" else
13             input_a(2) when input_s = "10" else
14                 input_a(3) when input_s = "11" else
15                     '-';
16 end rtl;
```

Come descritto nel codice VHDL, il multiplexer 4:1 presenta:

- quattro linee di ingresso;
- due linee di selezione;
- una singola uscita.

L'uscita viene selezionata in base alla tabella di verità presente in figura 1.2.

Multiplexer 16:1

Il multiplexer 16:1 è stato realizzato seguendo un approccio strutturale. Questo approccio permette di *costruire sistemi complessi a partire da sistemi più semplici*. Questo approccio permette inoltre di usare al meglio i principi ingegneristici della modularità e della separazione

degli interessi: una volta sviluppato un componente e una volta assicuratoci che questo sia funzionante (ossia che assolve alle funzioni per cui è stato progettato e che rispetti le specifiche iniziali), potrà essere utilizzato anche per i progetti futuri *senza dover riprogettare ogni volta quel componente da zero*. Riportiamo l'implementazione VHDL del multiplexer 16:1.

```
1 entity mux_16_1 is
2 Port (
3     a: in std_logic_vector(0 to 15);
4     s: in std_logic_vector(3 downto 0);
5     y: out std_logic
6 );
7 end mux_16_1;
8
9 architecture Structural of mux_16_1 is
10
11 signal u: std_logic_vector(0 to 3);
12 component mux_4_1
13 port(
14     input_a: in std_logic_vector(0 to 3);
15     input_s: in std_logic_vector(1 downto 0);
16     output_y: out std_logic
17 );
18 end component;
19
```

```
20 begin
21   mux_0_3: for i in 0 to 3 generate m: mux_4_1
22     port map(
23       input_a(0 to 3) => a(i*4 to i*4+3),
24       input_s(1 downto 0) => s(1 downto 0),
25       output_y => u(i)
26     );
27   end generate;
28
29   mux_4: mux_4_1
30     port map(
31       input_a(0 to 3) => u(0 to 3),
32       input_s(1 downto 0) => s(3 downto 2),
33       output_y => y
34     );
35 end Structural;
```

Descrivendo brevemente questo codice, possiamo notare il FOR utilizzato per istanziare i mux 4:1 precedentemente realizzati. A differenza di quanto avviene nei linguaggi di programmazione, in VHDL, che è un linguaggio di descrizione dell'hardware, il FOR è inteso come FOR spaziale, ossia che inserisce componenti fisici reali sulla board. Viene inoltre utilizzato un segnale interno u, che verrà associato alle quattro uscite dei multiplexer 4:1 presenti al primo livello. Infine, l'uscita dell'unico multiplexer 4:1 presente al secondo livello sarà l'uscita complessiva del sistema.

Notiamo che i bit meno significativi delle selezioni del sistema comple-

sivo vengono utilizzati in ingresso ai multiplexer del primo livello, mentre i bit più significativi vengono utilizzati in ingresso al multiplexer del secondo livello.

1.2.2 Testing del sistema

La fase di testing è una delle fasi più importanti della progettazione di un sistema digitale. Assicurarsi che un componente sviluppato sia funzionante e rispetti le specifiche iniziali è fondamentale per costruire sistemi complessi. È chiaro che il testing non può mai essere esaustivo, in quanto più un sistema diventa complesso, più aumenteranno gli ingressi e gli stati, più sarà difficile testare tutte le combinazioni. L'approccio al testing dunque prevede di testare casi significativi e casi al limite.

Per assicurarci che il sistema sviluppato funzioni, abbiamo a disposizione lo strumento del testbench. Il **testbench** genera e fornisce gli ingressi al nostro sistema, che viene stimolato, e in funzione di questi stimoli genererà un'uscita, la quale può essere analizzata per compararla con l'uscita desiderata. Il testbench viene realizzato come un process, quindi a livello comportamentale, senza sensitivity list. Infatti, la sensitivity list non è altro che una lista di segnali a cui il sistema reagisce in qualche modo. Ma il testbench realizza la "scatola esterna" del sistema, è l'ambiente in cui il sistema opera, dunque fuori da questo ambiente non c'è nulla (ecco perchè la sensitivity list è assente).

Quando utilizziamo un approccio strutturale per la realizzazione di un sistema, una corretta best-practice prevede che alla fine ci ritroviamo i testbench di tutti i componenti che compongono un sistema. Nel caso dell'esercizio 1.1., quando è stato testato il multiplexer 16:1, già erano stati sviluppati in precedenza i testbench del multiplexer 4:1, che compone il 16:1. Questo concetto è abbastanza intuitivo: parto da un componente piccolo, verifico la sua correttezza, se è corretto lo utilizzo per costruire un sistema più grande, altrimenti devo riprogettarlo.

La simulazione del multiplexer 16:1 è andata a buon fine, pertanto questo componente è pronto per essere utilizzato per l'esercizio successivo.

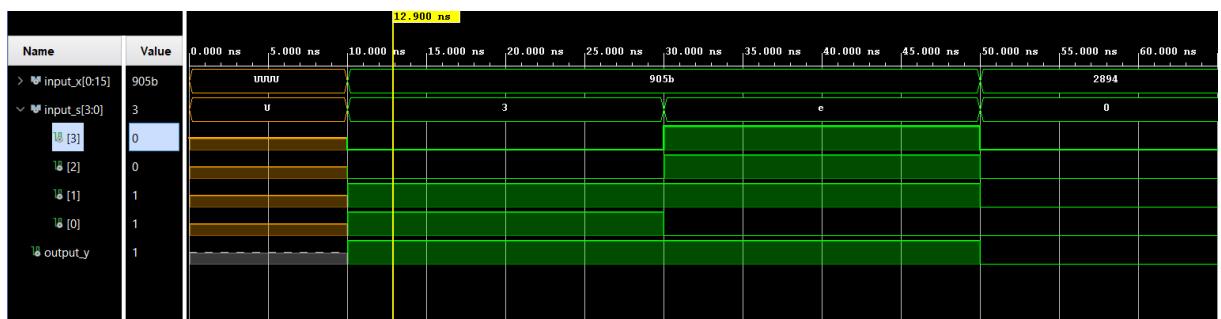


Figura 1.3: Simulazione multiplexer 16:1

1.3 Esercizio 1.2 - Rete di interconnessione

16:4

1.3.1 Descrizione e componenti utilizzati

Per la realizzazione di questo sistema abbiamo seguito un approccio strutturale: a partire da un multiplexer 16:1 (precedentemente sviluppato) e da un demultiplexer 1:4, collegando opportunamente l'uscita del primo blocco con l'ingresso del secondo blocco otteniamo la rete desiderata.

Demultiplexer 1:4

Il demultiplexer 1:4 è una macchina duale al multiplexer 4:1 in quanto distribuisce il segnale di ingresso su una delle uscite selezionata dagli ingressi di controllo. Le altre uscite, cioè quelle non selezionate, mantengono il loro valore precedente. Viene spesso utilizzato in accoppiata a un multiplexer (ad esempio nelle centrali telefoniche).

Come possiamo notare dalla figura 1.5, il demultiplexer 1:4 possiede:

- un ingresso;
- due ingressi di selezione;
- quattro uscite.

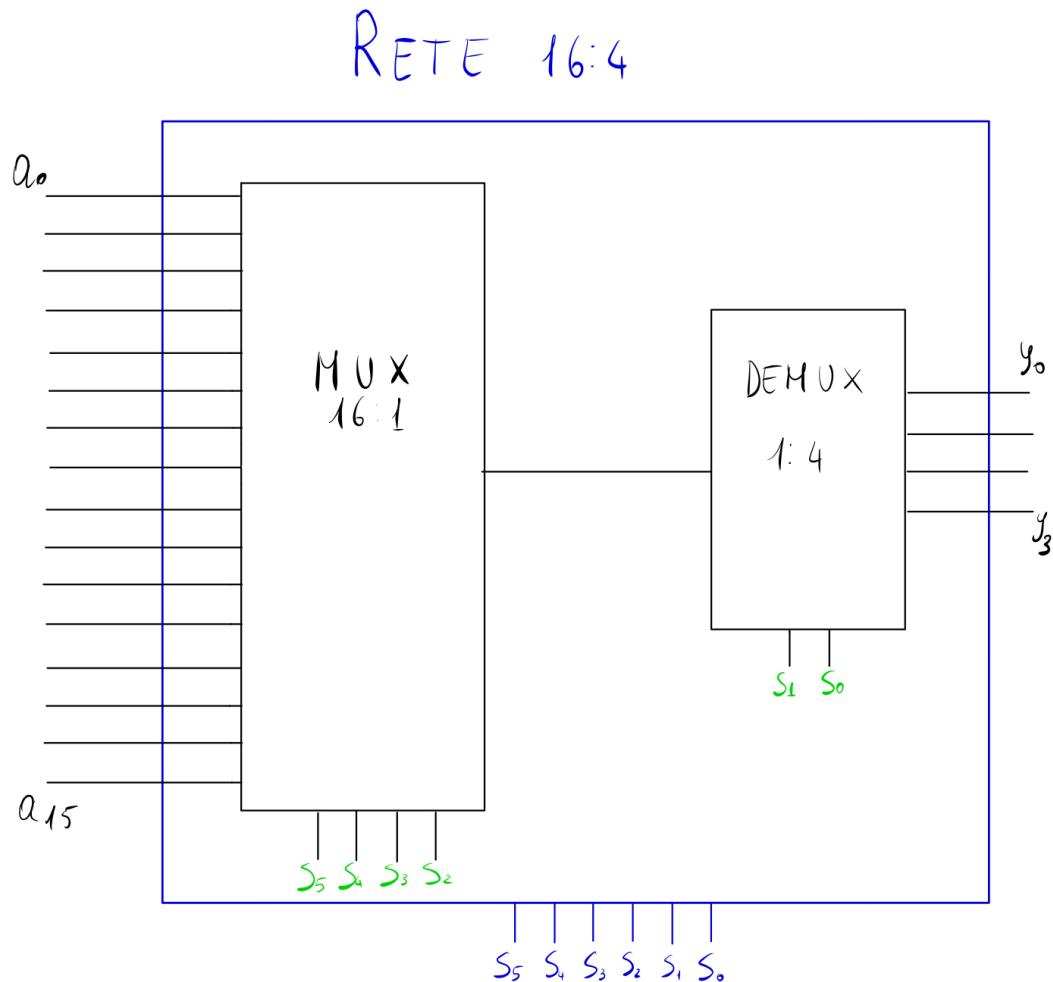


Figura 1.4: Rete di interconnessione 16:4

Per sviluppare il componente in VHDL abbiamo usato un approccio comportamentale. Questo approccio permette di concentrarci esclusivamente sul funzionamento del componente, cioè a quello che deve fare, senza preoccuparsi degli aspetti progettuali. Ne deduciamo che è conveniente utilizzarlo per sistemi abbastanza semplici, altrimenti il componente potrebbe essere simulabile ma non sintetizzabile sul FPGA. Segue l'implementazione VHDL del demultiplexer 1:4.

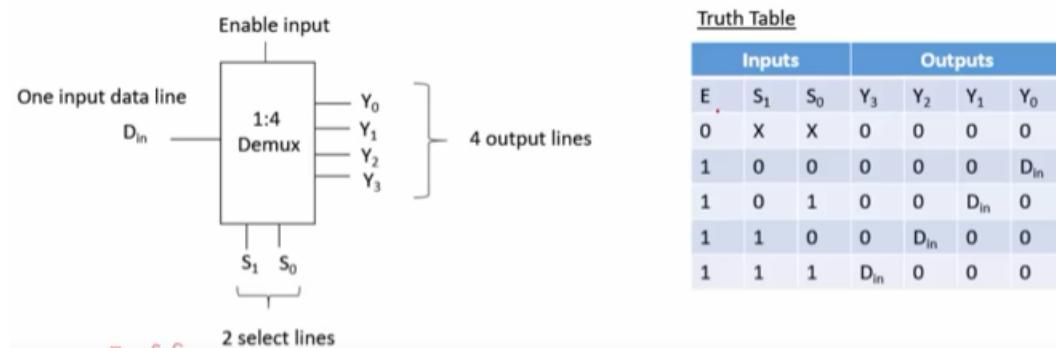


Figura 1.5: Demultiplexer 1:4

```

1 entity demux_1_4 is
2 port(
3     x : in std_logic;
4     k : in std_logic_vector(1 downto 0);
5     z : out std_logic_vector(0 to 3)
6 );
7 end demux_1_4;
8
9 architecture Behavioral of demux_1_4 is
10 begin
11     process(x, k)
12     begin
13         if(k = "00") then
14             z(0) <= x;
15             z(1) <= '0';
16             z(2) <= '0';
17             z(3) <= '0';
18         elsif(k = "01") then
19             z(0) <= '0';
20             z(1) <= x;

```

```
21      z(2) <= '0';  
22      z(3) <= '0';  
23      elsif(k = "10") then  
24          z(0) <= '0';  
25          z(1) <= '0';  
26          z(2) <= x;  
27          z(3) <= '0';  
28      elsif(k = "11") then  
29          z(0) <= '0';  
30          z(1) <= '0';  
31          z(2) <= '0';  
32          z(3) <= x;  
33      end if;  
34  end process;  
35 end Behavioral;
```

1.3.2 Realizzazione del sistema complessivo

Sulla base del multiplexer 16:1 e del demultiplexer 1:4 sviluppati, è stato possibile realizzare la rete di interconnessione 16:4. Questo sistema manderà in uscita quattro linee di ingresso, opportunamente selezionate dagli ingressi di selezione. Notiamo che i bit meno significativi delle selezioni vengono utilizzate nel multiplexer 16:1 al primo livello, mentre i bit più significativi vengono utilizzati per il demultiplexer 1:4 al secondo livello. In seguito è riportata l'implementazione VHDL del sistema.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity rete_16_4 is
5   port(
6     a : in std_logic_vector(0 to 15);
7     s : in std_logic_vector(5 downto 0);
8     y : out std_logic_vector(0 to 3)
9   );
10
11 end rete_16_4;
12
13 architecture structural of rete_16_4 is
14   signal f: std_logic; --uscita del mux al primo livello
15   component mux_16_1 is
16     port(
17       a : in std_logic_vector(0 to 15);
18       s : in std_logic_vector(3 downto 0);
19       y : out std_logic
20     );
21   end component;
22
23   component demux_1_4 is
24     port(
25       x : in std_logic;
26       k : in std_logic_vector(1 downto 0);
27       z : out std_logic_vector(0 to 3)
28     );

```

```
29      end component;  
30  
31      begin  
32          mux : mux_16_1  
33          port map(  
34              a(0 to 15) => a(0 to 15),  
35              s(3 downto 0) => s(5 downto 2),  
36              y => f  
37          );  
38  
39          demux : demux_1_4  
40          port map(  
41              x => f,  
42              k(1 downto 0) => s(1 downto 0),  
43              z(0 to 3) => y(0 to 3)  
44          );  
45      end structural;
```

1.3.3 Testing del sistema complessivo

Dopo aver testato opportunamente il multiplexer 16:1 nella sezione precedente, e il demultiplexer 1:4, siamo dunque pronti per testare la rete di interconnessione. La simulazione del sistema complessiva è riportata in figura 1.7.

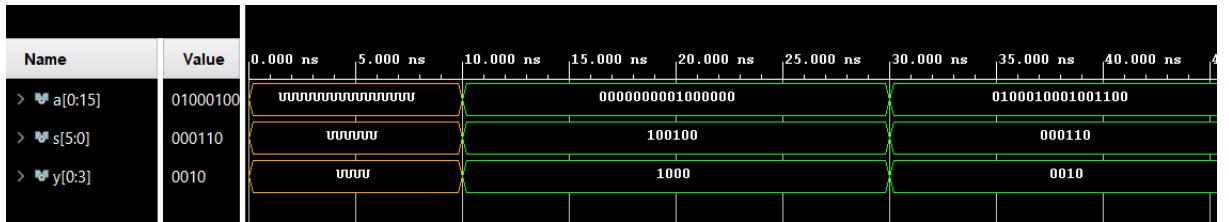


Figura 1.6: Simulazione demultiplexer 1:4

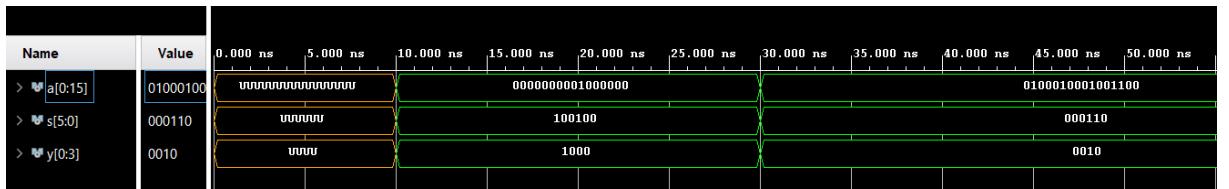


Figura 1.7: Simulazione rete di interconnessione 16:4

1.4 Esercizio 1.3 – Implementazione sulla board della rete 16:4

L’implementazione sulla board di questo progetto è stata abbastanza immediata in quanto ha richiesto semplicemente l’utilizzo di 16 switch e 4 led forniti dalla scheda. In sostanza, questo è un semplice mapping, che non ha richiesto modifiche del top module per l’inserimento di ulteriori componenti preposti alla realizzazione dei collegamenti poiché si tratta semplicemente di collegare un filo. Per realizzare questo mapping, abbiamo utilizzato il file dei constraints. Questo file permette di specificare constraints temporali (per stabilire ad esempio una certa frequenza con la quale il sistema deve operare) o constraints fisici (e quindi determinare come determinati pin vengono mappati nel mio progetto finale). Nel nostro caso, abbiamo specificato constraints fisi-

ci, mappando gli ingressi della rete di interconnessione con gli switch e le uscite con i led.

```

11 | ##Switches
12 | set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { s[0] }]; #IO_L24N_T3_R50_15 Sch=sw[0]
13 | set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { s[1] }]; #IO_L3N_T0_DQS_EMCCLR_14 Sch=sw[1]
14 | set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { s[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
15 | set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { s[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
16 | set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { s[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
17 | set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { s[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
18 | #set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { SW[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
19 | #set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { SW[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
20 | #set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMOS18 } [get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
21 | #set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS18 } [get_ports { SW[9] }]; #IO_25_34 Sch=sv[9]
22 | #set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMOS33 } [get_ports { SW[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
23 | #set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
24 | #set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sv[12]
25 | #set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
26 | #set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMOS33 } [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sv[14]
27 | #set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]
28 |
29 | ## LEDs
30 | set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports { y[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
31 | set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [get_ports { y[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
32 | set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [get_ports { y[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
33 | set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [get_ports { y[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]

```

Figura 1.8: File dei constraints per la rete 16:4

L'ultima fase dell'implementazione sulla board è la generazione del bitstream. Il bitstream non è altro che un file che rappresenta una "mappa" della memoria, ciò determina che bit c'è in ogni cella. Proprio per le sue caratteristiche, indipendentemente dalla complessità del programma, il bitstream avrà sempre la stessa dimensione.

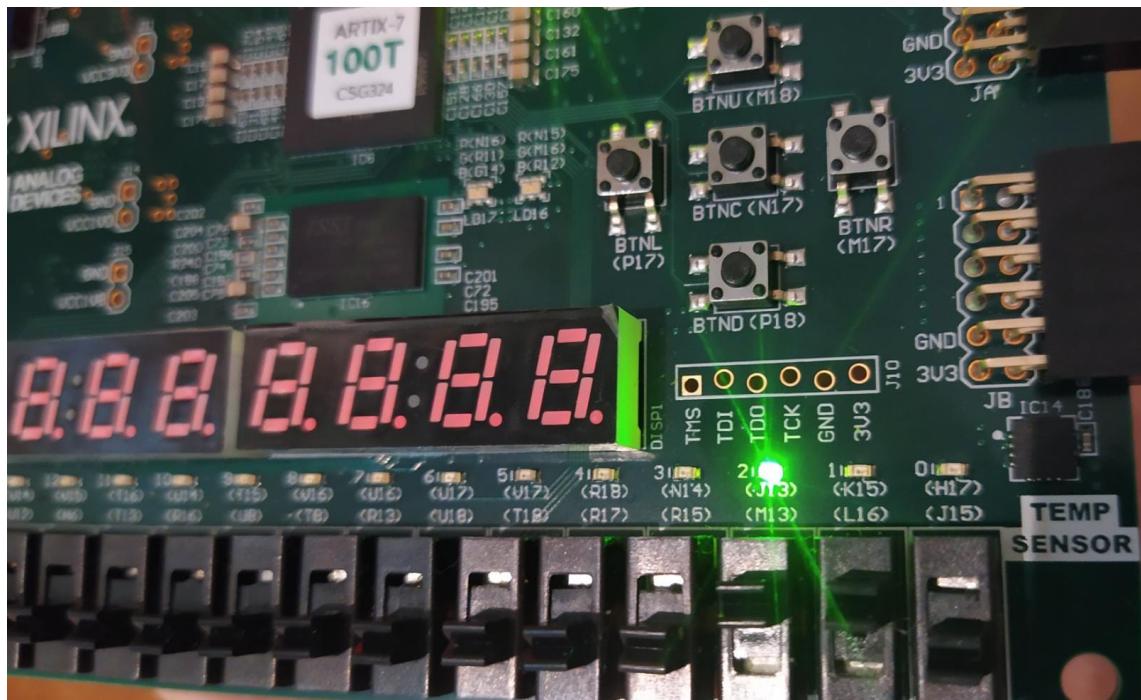


Figura 1.9: Implementazione sulla board della rete 16:4

Come possiamo notare dalla figura 1.9, dato l’ingresso 0100010001001100 e la selezione 000110 otteniamo in uscita 0010 . L’1 dell’uscita sarebbe $y(2)$, associata al led J13.

1.5 Esercizio 2.1 - Encoder BCD

1.5.1 Traccia

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

- Input: 0000000001 Output: 0000 (cifra 0)
- Input: 0000000010 Output: 0001 (cifra 1)
- Input: 0000000100 Output: 0010 (cifra 2)
- ...

1.5.2 Descrizione della soluzione

Un encoder è un circuito digitale combinatorio dotato di 2^n segnali di ingresso e di n segnali di uscita. Se viene attivata una delle linee di ingresso, in uscita viene prodotto il codice corrispondente. In particolare, l'esercizio in questione richiede l'implementazione di un encoder BCD. Il componente ha 10 ingressi, $[X_0, \dots, X_9]$, e l'uscita è codificata su 4 bit, dunque $[Y_0, \dots, Y_3]$. Quindi, questo dispositivo non sfrutta tutte le 16 combinazioni possibili delle uscite.

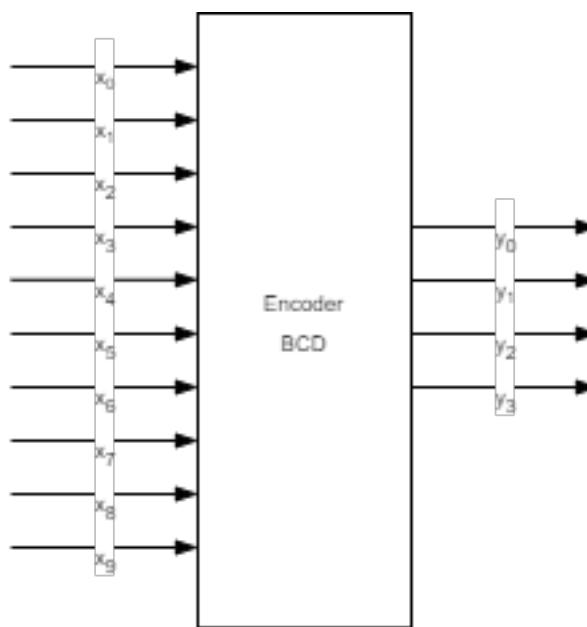


Figura 1.10: Encoder BCD

Riportiamo di seguito l'implementazione in VHDL. Essendo un componente elementare, l'architettura è stata definita di tipo behavioral.

```

1 entity enc_bcd is
2 Port(
3     x : in STD_LOGIC_VECTOR(0 to 9); — ingressi encoder
4     y : out STD_LOGIC_VECTOR(0 to 3) — uscite encoder
5 );
6 end enc_bcd;
7
8 architecture Behavioral of enc_bcd is
9 begin
10 process(x)
11 begin
12 case x is
13     when "0000000001" => y <= "0000";

```

```
14      when "0000000010" => y <= "0001";
15      when "0000000100" => y <= "0010";
16      when "0000001000" => y <= "0011";
17      when "0000010000" => y <= "0100";
18      when "0000100000" => y <= "0101";
19      when "0001000000" => y <= "0110";
20      when "0010000000" => y <= "0111";
21      when "0100000000" => y <= "1000";
22      when "1000000000" => y <= "1001";
23      when others => y <= "1111";
24
25  end case;
26 end process;
27
28 end Behavioral;
```

1.5.3 Testing simulato

Si riporta di seguito il testing effettuato per l'encoder BCD. Il comportamento, come è possibile verificare dall'immagine, è quello atteso. Per i primi $50ns$ abbiamo che non essendo definito il segnale allora l'uscita è 1111, ovvero 15 in decimale.



Figura 1.11: Simulazione encoder bcd

Trascorso tale lasso di tempo, si osserva come, dato in ingresso il segnale 0000100000, ci si aspetta la codifica di X_5 , infatti in uscita otteniamo 101, ovvero 5, come si evince dalla figura 1.11.

1.6 Esercizio 2.2 - Encoder con LED

Questo esercizio è molto simile al precedente con la rete 16:4, in quanto basta semplicemente modificare il file dei constraints, senza introdurre nuove unità. Nel primo caso di test riportato in figura 1.12, notiamo gli effetti del costrutto *when others*: infatti, quando si presenta il caso con tutti 0 (tutti gli switch abbassati) in uscita ho 1111 (quindi i quattro led sono accesi). Nel secondo test in figura 1.13 è riportato lo stesso caso di test analizzato nella sezione precedente.

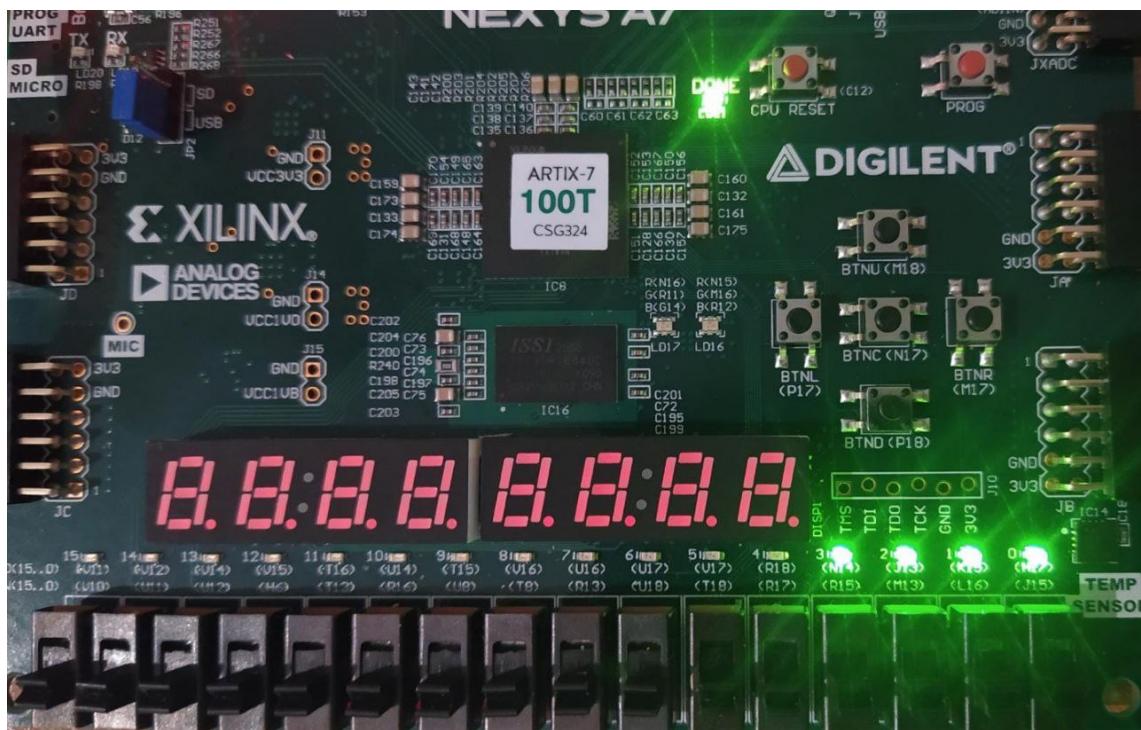


Figura 1.12: Encoder su board con led: caso when others

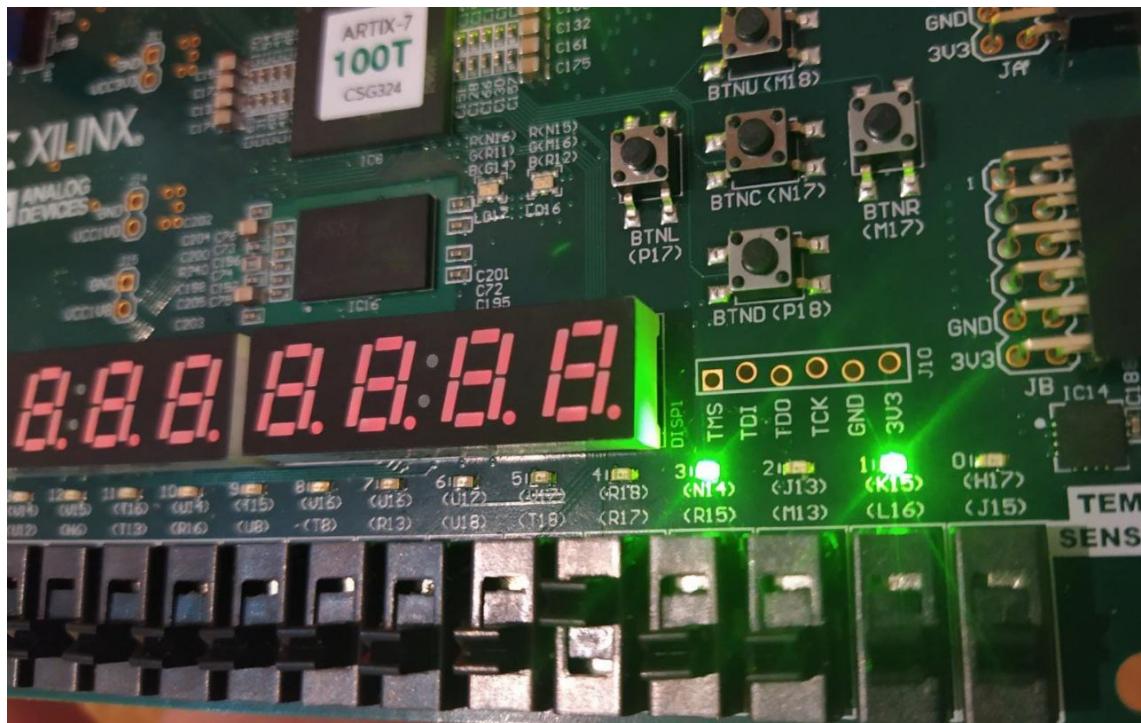


Figura 1.13: Encoder con board su led

1.7 Esercizio 2.3 - Encoder con display a sette segmenti

Questo progetto ha richiesto l'implementazione dell'esercizio precedente mostrando in output la cifra decimale sul display a sette segmenti.

Display a sette segmenti

La board di sviluppo è dotata di *otto display a sette segmenti*. L'utilizzo di questi display è leggermente più complicato degli switch e dei led. Ogni display è dotato di un **anodo** e di un **catodo**: l'anodo mi permette di specificare quale display utilizzare, il catodo mi permette di specificare la cifra da rappresentare su un determinato display. Di default, tutti i display condividono lo stesso catodo, quindi significa che, senza effettuare modifiche, *tutti i display mostreranno la stessa cifra*.

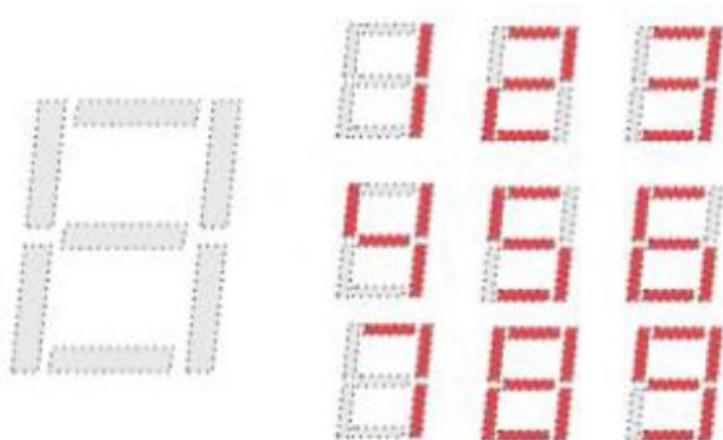


Figura 1.14: Display a sette segmenti

Per mostrare cifre diverse sui vari display dobbiamo utilizzare una tecnica di "refresh" che permette all'occhio umano di vedere cifre distinte sui vari display. Questa tecnica semplicemente prevede di far *variare le cifre velocemente sul display dando l'illusione della persistenza*, sfruttando quindi una caratteristica dell'occhio umano.

Nella nostra implementazione, abbiamo per semplicità, lasciato la funzionalità di default degli schermi, andando a visualizzare la stessa cifra su tutti i display. Pertanto, non è stato necessario introdurre il meccanismo di refresh precedentemente citato. Tuttavia, è stato necessario gestire i catodi e ricreare le varie cifre possibili e mapparli sulle configurazioni degli anodi.

Encoder on board e cathodes manager

Per la realizzazione del progetto, abbiamo definito una nuova entity che utilizza l'encoder sviluppato nell'esercizio precedente e un gestore dei catodi, che ha il compito di gestire il display.

```
1 entity Encoder_onBoard is
2   Port (
3     switch : in STD_LOGIC_VECTOR(9 downto 0);
4     led : out STD_LOGIC_VECTOR(3 downto 0);
5     catodi : out STD_LOGIC_VECTOR(7 downto 0);
6     anodi : out STD_LOGIC_VECTOR(7 downto 0)
7   );
8 end Encoder_onBoard;
```

L'**encoder on board** presenta dunque in input i valori forniti tramite gli switch e come uscita i led, gli anodi e i catodi necessari al funzionamento del display. Il **cathodes manager** invece ha in input il valore (cifra) da rappresentare e in uscita la configurazione per rappresentare quella cifra sul display a sette segmenti

```

1 entity cathodes_manager is
2   Port ( value : in STD_LOGIC_VECTOR (3 downto 0);
3         dot : in STD_LOGIC;
4         cathodes_dot : out STD_LOGIC_VECTOR (7 downto 0));
5 end cathodes_manager;
```

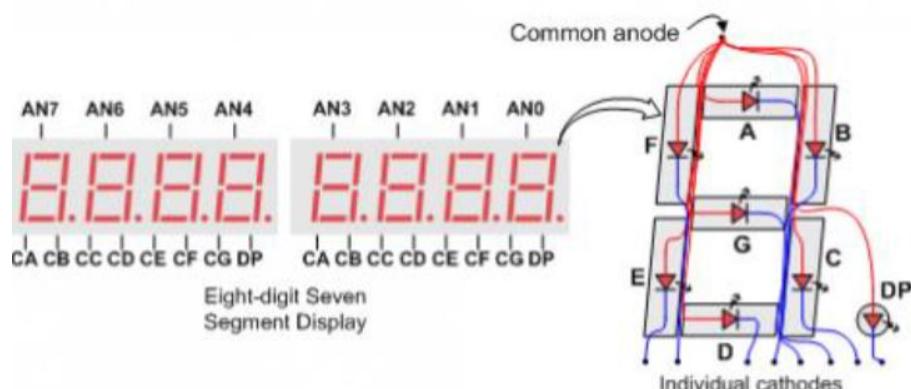


Figura 1.15: Display a sette segmenti con anodi e catodi

Di seguito sono riportate tutte le configurazioni che permettono di rappresentare tutte le possibili cifre.

```

1 constant zero    : std_logic_vector(6 downto 0) := "1000000";
2 constant one     : std_logic_vector(6 downto 0) := "1111001";
3 constant two      : std_logic_vector(6 downto 0) := "0100100";
```

```

4 constant three : std_logic_vector(6 downto 0) := "0110000";
5 constant four : std_logic_vector(6 downto 0) := "0011001";
6 constant five : std_logic_vector(6 downto 0) := "0010010";
7 constant six : std_logic_vector(6 downto 0) := "0000010";
8 constant seven : std_logic_vector(6 downto 0) := "1111000";
9 constant eight : std_logic_vector(6 downto 0) := "0000000";
10 constant nine : std_logic_vector(6 downto 0) := "0010000";
11 constant a : std_logic_vector(6 downto 0) := "0001000";
12 constant b : std_logic_vector(6 downto 0) := "0000011";
13 constant c : std_logic_vector(6 downto 0) := "1000110";
14 constant d : std_logic_vector(6 downto 0) := "0100001";
15 constant e : std_logic_vector(6 downto 0) := "0000110";
16 constant f : std_logic_vector(6 downto 0) := "0001110";

```

Da un punto di vista implementativo, l'esercizio ha richiesto dunque una modifica al file dei constraints per *associare i segnali del cathodes manager ai segnali fisici degli anodi e dei catodi presenti sulla board.*

```

55 ##7 segment display
56 set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMS33 } [get_ports { catodi[0] }]; #IO_L24N_T3_A00_D16_14 Sch=oe
57 set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMS33 } [get_ports { catodi[1] }]; #IO_25_14 Sch=ob
58 set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMS33 } [get_ports { catodi[2] }]; #IO_25_15 Sch=oc
59 set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMS33 } [get_ports { catodi[3] }]; #IO_L17P_T2_A26_15 Sch=cd
60 set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMS33 } [get_ports { catodi[4] }]; #IO_L13P_T2_MRCC_14 Sch=cd
61 set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMS33 } [get_ports { catodi[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
62 set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMS33 } [get_ports { catodi[6] }]; #IO_L4P_TO_D04_14 Sch=og
63 set_property -dict { PACKAGE_PIN H15 IOSTANDARD LVCMS33 } [get_ports { catodi[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
64 set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMS33 } [get_ports { anodi[0] }]; #IO_L23P_T3_FOB_B_15 Sch=an[0]
65 set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMS33 } [get_ports { anodi[1] }]; #IO_L23N_T3_FWB_B_15 Sch=an[1]
66 set_property -dict { PACKAGE_PIN T9 IOSTANDARD LVCMS33 } [get_ports { anodi[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
67 set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMS33 } [get_ports { anodi[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
68 set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMS33 } [get_ports { anodi[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
69 set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMS33 } [get_ports { anodi[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
70 set_property -dict { PACKAGE_PIN K2 IOSTANDARD LVCMS33 } [get_ports { anodi[6] }]; #IO_L23P_T3_35 Sch=an[6]
71 set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMS33 } [get_ports { anodi[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
--
```

Figura 1.16: File dei constraints per supporto ai display

Test del progetto sulla board

Di seguito sono riportati i risultati dell'implementazione del progetto.

Possiamo notare che, quando seleziono una configurazione non definita, questa viene ancora coperta dal when others inserito da noi durante la progettazione dell'encoder standard, mostrando a schermo al cifra F (1111).

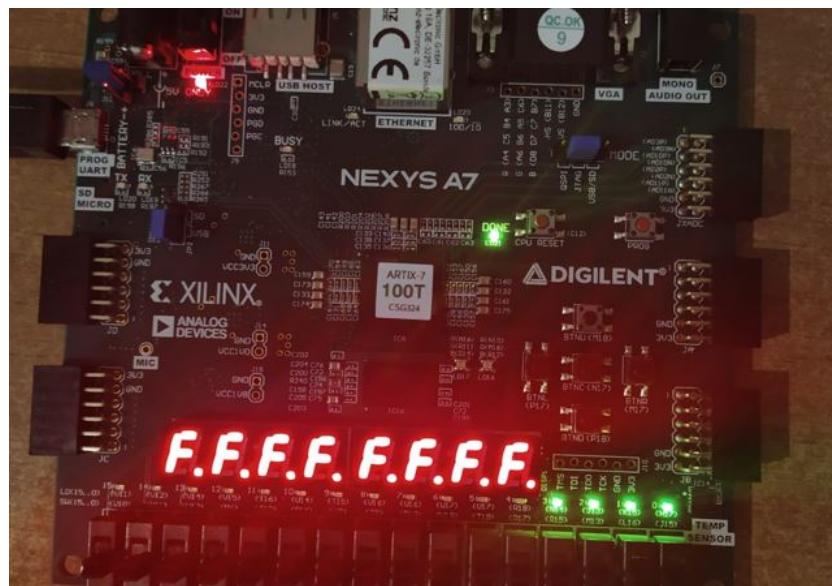


Figura 1.17: Output del display caso 0000000000

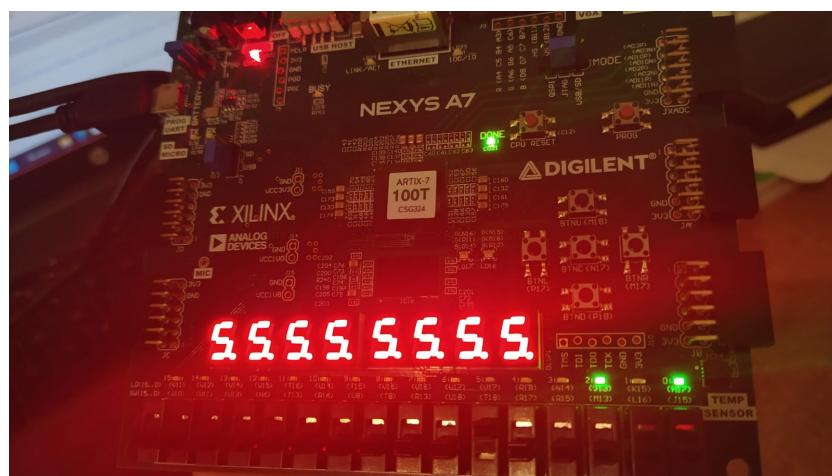


Figura 1.18: Output del display nel caso 0000100000

Capitolo 2

Reti sequenziali elementari

2.1 Traccia

2.1.1 Esercizio 3.1

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

sequenza viene correttamente riconosciuta.

2.1.2 Esercizio 3.2

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

2.1.3 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. Il componente deve essere realizzato utilizzando sia a) un approccio comportamentale sia b) un approccio strutturale.

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

2.1.4 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.

2.1.5 Esercizio 5.2

Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando i display a 7 segmenti per la visualizzazione dell'orario (o una combinazione di display e led nel caso in cui i display a disposizione siano in numero inferiore a quello necessario), gli switch per l'immissione dell'orario iniziale e due bottoni, uno per il set dell'orario e uno per il reset. Si utilizzi una codifica a scelta dello studente per la visualizzazione dell'orario sui display (esadecimale o decimale).

2.1.6 Esercizio 5.3

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

2.1.7 Esercizio 6.1

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

2.1.8 Esercizio 6.2

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

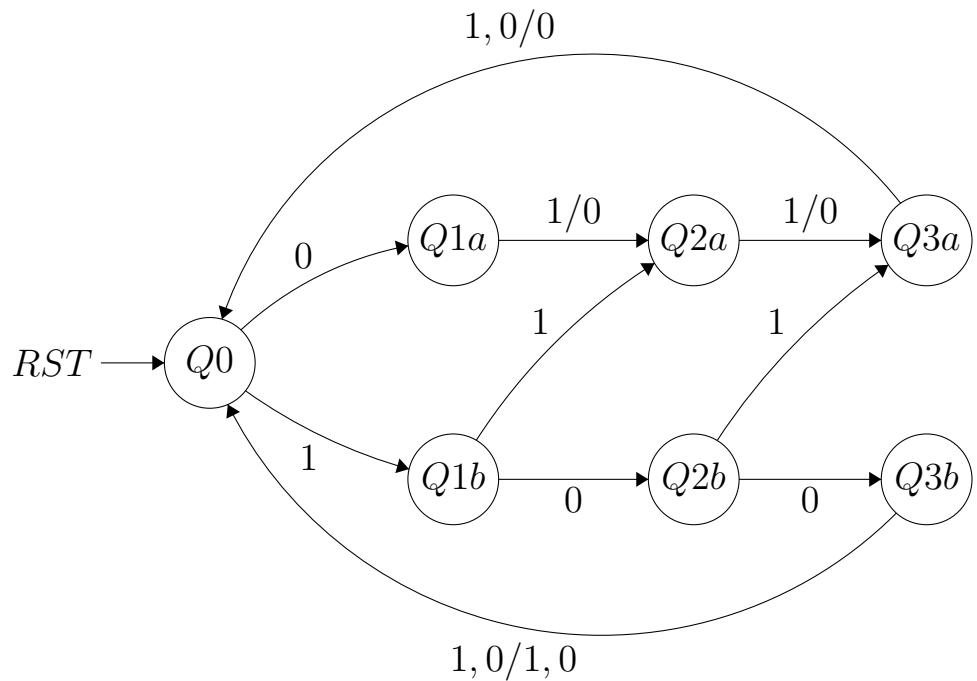
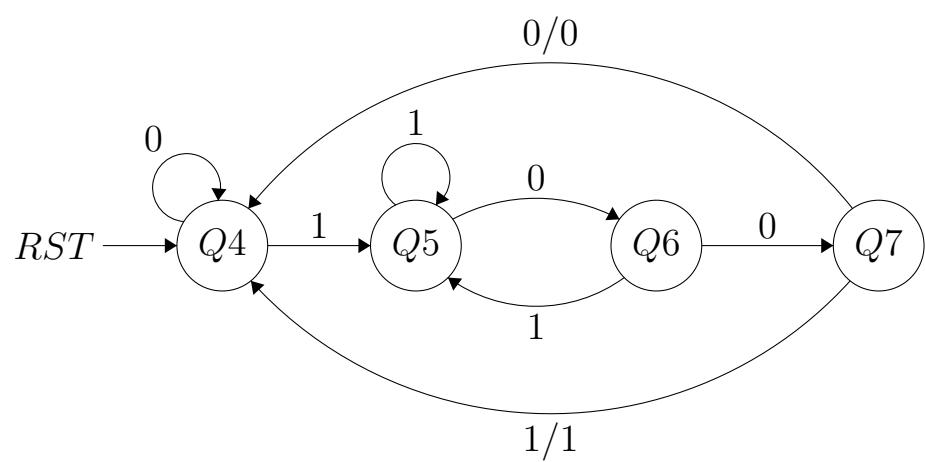
CAPITOLO 2. RETI SEQUENZIALI ELEMENTARI

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

2.2 Esercizio 3.1 – Riconoscitore di sequenze

Il riconoscitore di sequenze è stato modellato tramite un'automa a stati finiti (ASF) basandoci sul modello di Mealy. La particolarità di tale automa è il segnale di modo di funzionamento M . Il comportamento, che varia in base al segnale M , è stato modellato con due automi distinti. Per $M=0$, ovvero quando il sistema deve riconoscere la sequenza di gruppi di 4 bit, possiamo interpretare l'automa come diviso in due parti, un percorso dove si ha la sequenza corretta mentre l'altro la sequenza errata. Si può notare che quando $M=0$, il numero di stati tende ad aumentare notevolmente in quanto il controllo sulla correttezza della sequenza verrà comunque fatto ogni 4 bit, indipendentemente se quella sequenza parziale potrà portarmi o meno alla sequenza da riconoscere.

Dalla figura 2.1 si può notare come, partendo dallo stato q_0 , se arriva in ingresso il valore corretto, in questo caso 1, allora il sistema si porta nello stato q_1b , a questo punto se dovessero arrivare sempre valori corretti il sistema arrivo fino allo stato q_3b per poi tornare allo stato q_0 con uscita 1 nel caso la sequenza ottenuta sia corretta, altrimenti l'uscita sarà 0. Si osserva come per gli stati q_1b e q_2b , nel caso non vi sia stato presentato il valore corretto in ingresso allora il sistema si porta nella sequenza errata, rispettivamente negli stati q_2a e q_3a .


 Figura 2.1: Automa per $M = 0$

 Figura 2.2: Automa per $M = 1$

2.2.1 Descrizione e considerazioni sulle scelte progettuali

Abbiamo deciso di gestire il sistema in maniera totalmente sincrona andando ad inserire il clock nella sensitivity list. Il RST è stato gestito anch'esso in maniera sincrona, in quanto lo valuto sul fronte del clock (che è anche la soluzione preferita da Xilinx).

```
1 stato_uscita : process(clk)
2 begin
3   if rising_edge(CLK) then
4     if RST = '1' then
5       stato_corrente <= q0;
6       y <= '0';
```

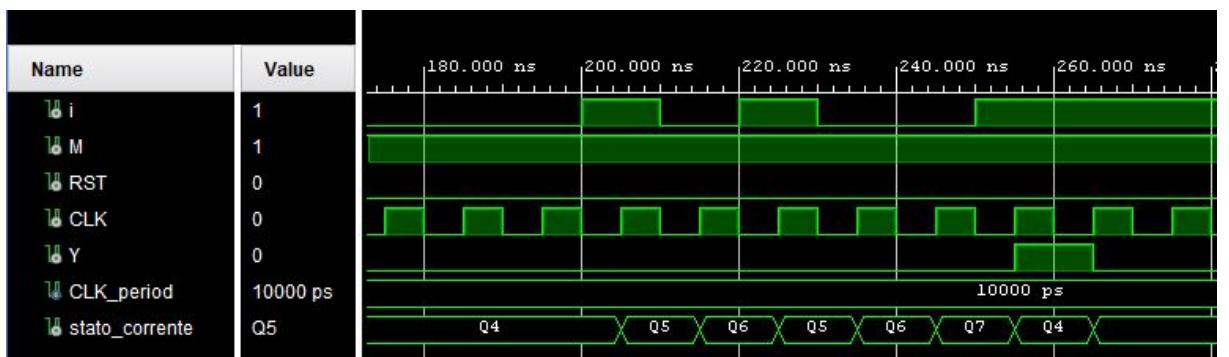
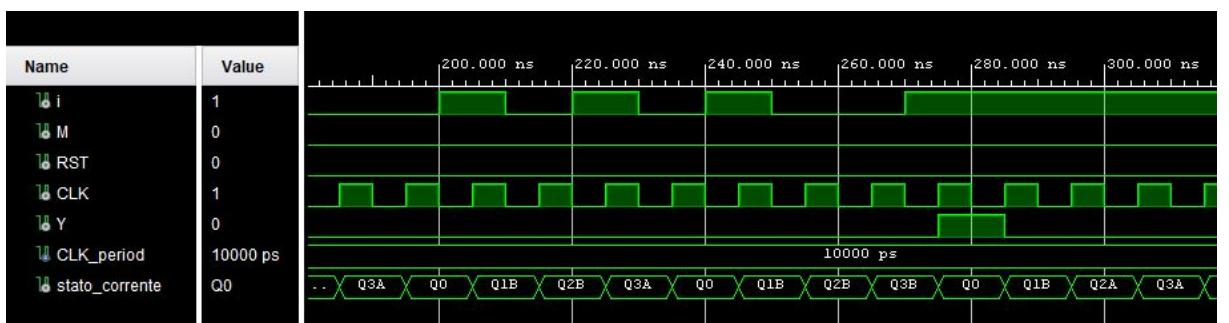
L'inserimento del when others nel codice è stata una scelta progettuale. La presenza del costrutto garantisce la coerenza del sistema in ogni momento, ma chiaramente sarà uno svantaggio in caso di test perché sarà più difficile capire da dove proviene l'errore poiché potrebbe essere coperto dal costrutto. Quindi, dal punto di vista del funzionamento è meglio, ma è peggio da un punto di vista del testing. Abbiamo scelto di realizzare il clock in modo periodico per semplicità.

```
1 when others =>
2   stato_corrente <= Q0;
3   y <= '0';
```

Il progetto è stato realizzato con un solo process. Questo porta a delle differenze con il modello con due process, visibili in fase di simulazione. In particolare, con il modello con un solo process, quando la sequenza viene riconosciuta, l'uscita non si alza all'arrivo dell'ultimo bit corretto ma bensì quando torno nello stato iniziale. Se avessimo voluto l'uscita "istantanea", avremmo dovuto utilizzare il modello con due process in modo da rendere la parte che genera Y combinatoria e non più sequenziale. Infatti, col modello a un process, poiché l'uscita viene presentata dopo, al prossimo fronte del clock, dovrà essere in qualche modo mantenuta. Questo non porta conseguenze da un punto di vista funzionale ma in fase di sintesi il modello a un process richiede un flip flop in più rispetto a quello a due process (che richiede solo i due flip flop per lo stato) per memorizzare l'uscita.

2.2.2 Testing del sistema

Le considerazioni fatte nella precedente sezione possono essere confermate e visualizzate in fase di simulazione, sia per $M=0$ sia per $M=1$.


 Figura 2.3: Simulazione automa per $M = 1$

 Figura 2.4: Simulazione automa per $M = 0$

2.3 Esercizio 3.2 – Riconoscitore su board

Per implementare il riconoscitore di sequenza precedentemente sviluppato, abbiamo seguito un approccio strutturale mettendo insieme i seguenti componenti:

- Riconoscitore di sequenza, precedentemente sviluppato, con alcune modifiche;
- Button debouncer.

Il progetto ha richiesto l'utilizzo del clock fisico della scheda trattandosi di una macchina sequenziale.

2.3.1 Button debouncer

Il progetto ha previsto l'utilizzo dei bottoni che, in un certo senso, hanno lo scopo di *"rendere "valido" l'input del dato in ingresso della sequenza e del modo.*

Un problema che abbiamo incontrato durante lo sviluppo è stato quello del *button bouncing*, legato ai due bottoni utilizzati per acquisire l'input in sincronismo con il segnale di temporizzazione. Il button bouncing è un fenomeno che causa una oscillazione alla pressione del pulsante, che deve essere in qualche modo gestita per far evolvere correttamente la macchina.

Il **debouncer** in sostanza si comporta da filtro digitale, dando al segnale di ingresso (con oscillazione) una forma ideale (senza oscillazio-

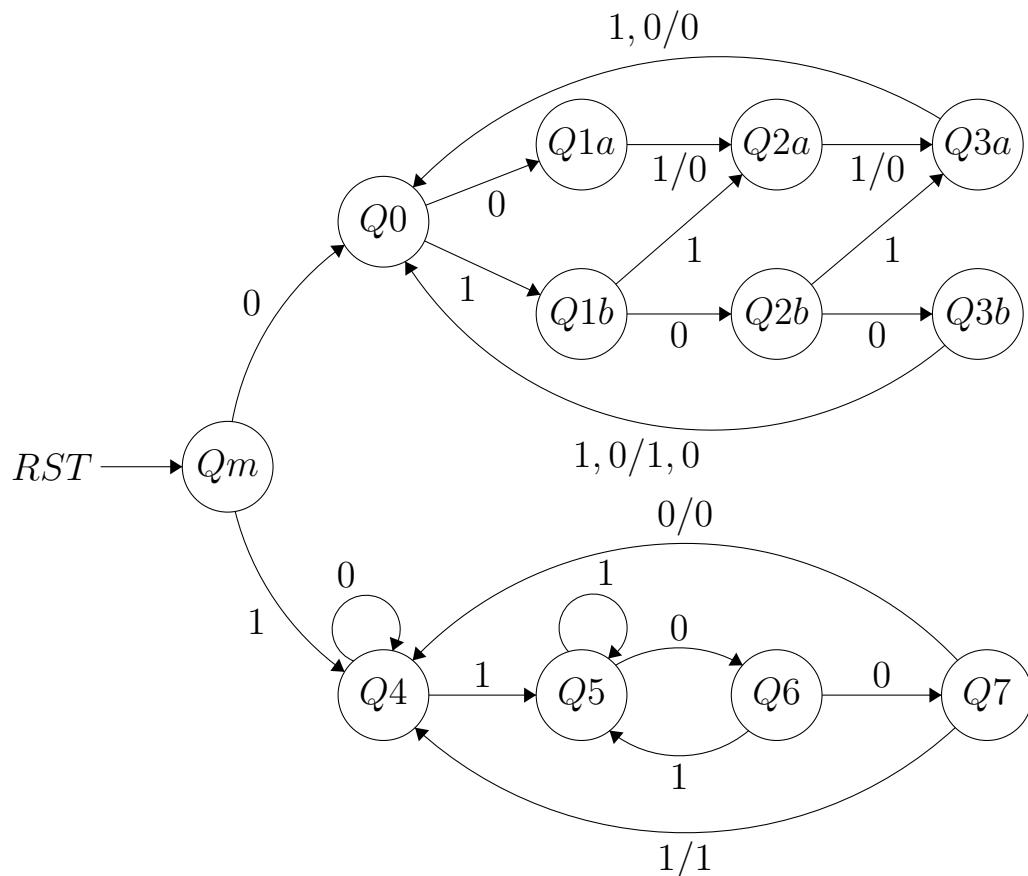


Figura 2.5: Caption

ne), così il segnale sarà alto una sola volta. Intuitivamente, il debouncer permette di *determinare un certo numero di fronti di clock dopo i quali l'oscillazione è terminata*. Per descrivere il funzionamento del button debouncer possiamo utilizzare un'automa a stati finiti. Questo automa presenta due soli stati (*PRESSED* e *NOT PRESSED*), due ingressi, il clock e il segnale del bottone originale e un'uscita, ovvero il segnale ripulito.

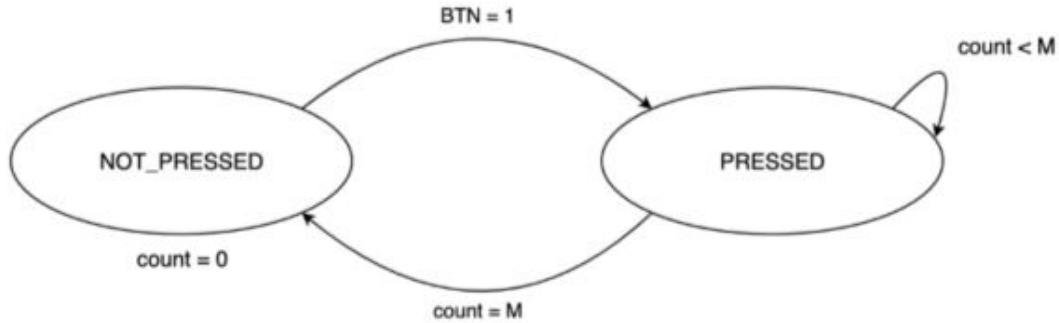


Figura 2.6: Automa button debouncer

```

1 entity ButtonDebouncer is
2     generic (
3         CLK_period:  integer := 10;
4         btn_noise_time:  integer := 6500000
5     );
6     Port ( RST : in STD_LOGIC;
7             CLK : in STD_LOGIC;
8             BTN : in STD_LOGIC;
9             CLEARED_BTN : out STD_LOGIC);
10 end ButtonDebouncer;

```

Il funzionamento è piuttosto semplice: fin quando non viene premuto, il debouncer rimarrà nello stato *NON PREMUTO*. Appena viene premuto, passerà nello stato *PRESSED*, e vi rimarrà fin quando il valore di conteggio non supererà un certo valore M , che quantifica *il numero di fronti di clock da aspettare prima che l'oscillazione si esaurisca*. Il valore M viene calcolato come *rapporto tra il tempo di oscillazione del bottone e il periodo del clock della scheda*. Una volta raggiunto M , il debouncer torna allo stato iniziale *NOT PRESSED* e

azzerà il valore di conteggio.

```
1 deb: process (CLK)
2 variable count: integer := 0;
3
4 begin
5   if rising_edge(CLK) then
6
7     if( RST = '1' ) then
8       BTN_state <= NOT_PRESSED;
9       CLEARED_BTN <= '0';
10    else
11      case BTN_state is
12        when NOT_PRESSED =>
13          CLEARED_BTN<= '0';
14          if( BTN = '1' ) then
15            BTN_state <= PRESSED;
16          else
17            BTN_state <= NOT_PRESSED;
18          end if;
19        when PRESSED =>
20          if(count = max_count -1) then
21            count:=0;
22            CLEARED_BTN <= '1';
23            BTN_state <= NOT_PRESSED;
24          else
25            count:= count+1;
26            BTN_state <= PRESSED;
```

```
27         end if;
28
29         when others =>
30             BTN_state <= NOT_PRESSED;
31         end case;
32     end if;
33 end process;
```

2.3.2 Riconoscitore di sequenza

Il riconoscitore di sequenza base è stato modificato per essere implementato agilmente sulla board. Alla entity originale abbiamo aggiunto un segnale enable che funge da *abilitazione per la lettura del dato della sequenza in input*. Quindi, in un certo senso, lo rende valido.

```
1  entity ric_seq is
2      port( i: in std_logic;
3             M: in std_logic;
4             RST : in std_logic;
5             CLK : in std_logic;
6             enable : in std_logic;
7             y: out std_logic
8      );
9  end ric_seq;
```

Successivamente, abbiamo definito un nuovo stato nell'automa, QM , che è uno stato in cui la macchina si va a trovare se viene premuto

il bottone associato al reset. QM rappresenta uno stato in cui non è ancora determinato il modo di riconoscimento della sequenza. Infatti, da QM , se seleziono tramite switch il modo 0, andrò nello stato "iniziale" dell'automa per $M=0$ del precedente esercizio ($Q0$), mentre se seleziono il modo 1 andrò nello stato iniziale dell'automa per $M=1$ del precedente esercizio ($Q4$).

Quindi, il modo può essere cambiato durante l'evoluzione della macchina seguendo due strategie:

- *Sul fronte di salita del clock, premendo il tasto di reset e, in funzione del valore dello switch del modo in quel determinato momento, andrò nella parte di automa "corretto";*
- *Premendo il bottone di abilitazione del modo.* In quel caso, la macchina tornerà in $Q0$ se abbasso lo switch del modo ($M=0$) oppure tornerà in $Q4$ se alzo lo switch del modo ($M=1$).

```

1  if rising_edge (CLK) then
2      if RST = '1' then
3          stato_corrente <= QM;
4          y <= '0';
5          if M = '0' then
6              stato_corrente <= Q0;
7          else
8              stato_corrente <= Q4;
9          end if;
10     end if;
```

```
11      if(enable = '1' and last_enable = '0') then
12          if(M = '0') then --codice automa per M = '0'
13              ....
14          elsif(M = '1') --codice automa per M = '1'
15              ....
16
17      last_enable = '0';
```

2.3.3 Sistema complessivo: riconoscitore di sequenza on board

Mettendo insieme i due componenti appena descritti abbiamo ottenuto il **riconoscitore on board**. In particolare, abbiamo utilizzato due component debouncer per la gestione dei due bottoni utilizzati (uno per l'input e uno per il modo) e il riconoscitore di sequenza standard. Attraverso un attento port map, è stato possibile mappare i segnali di ingresso dell'input e del modo attraverso due switch, e le abilitazioni per la lettura di tali valori attraverso un bottone. L'output invece viene mostrato attraverso l'illuminazione o meno di un led.

```
1  btn_ing : ButtonDebouncer port map (
2      RST => rst,
3      CLK => clk,
4      BTN => ing_en,
5      CLRD_BTN => clrd_ing
```

```
6      );
7
8      btn_mode : ButtonDebouncer port map (
9          RST => rst,
10         CLK => clk,
11         BTN => mode_en,
12         CLRD_BTN => clrd_mode
13     );
14
15      riconoscitore : ric_seq port map (
16          i => ing,
17          M => mode,
18          RST => reset_temp,
19          CLK => clk,
20          enable => clrd_ing,
21          y => yled
22      );
```

2.4 Esercizio 4.1 – Shift Register

Il **registro a scorrimento** è un componente costituito da una catena di celle di memoria (tipicamente realizzate attraverso flip flop) interconnesse tra loro: ad ogni impulso di clock essi consentono *lo scorrimento dei bit da una cella a quella immediatamente adiacente*. Lo scorrimento può avvenire in un'unica direzione o in direzioni variabili.

I registri a scorrimento hanno diverse applicazioni:

- convertitore serie-parallelo, poiché consente di passare dal mondo del calcolatore, fatto di byte e quindi parallelo, al mondo delle telecomunicazioni, realizzato con canali e quindi seriale;
- possono essere utilizzati per variare la velocità di trasferimento dei dati seriali su una linea nel caso in cui un dispositivo lento debba trasferire dei dati ad un dispositivo più veloce, i bit che arrivano dal dispositivo lento vengono memorizzati nel registro per poi essere inviati con una frequenza superiore a quello più veloce;
- possono essere utilizzati come linea di ritardo per far pervenire ad un elemento di un sistema un segnale con un certo ritardo rispetto al momento in cui viene generato.

L'ingresso nei registri a scorrimento può essere di tipo **seriale** o **parallelo**. Nel tipo seriale, i bit vengono caricati uno alla volta, mentre in quello parallelo tutti insieme. Analogi discorsi possono essere fatti

per l' uscita, che quindi può essere seriale (se prendiamo come uscita complessiva solo l'uscita dell'ultimo flip flop) o parallela (se prendiamo come uscita tutte le uscite dei flip flop che compongono il registro a scorrimento). Combinando opportunamente tutte le possibili combinazioni di ingressi e uscite otteniamo quattro tipologie di shift register: SISO, SIPO, PISO e PIPO. Nel nostro caso, abbiamo deciso di implementare un SIPO (Serial Input Parallel Output).

2.4.1 Implementazione behavioral

Andiamo prima di tutto a definire l'entity del nostro componente. Osserviamo che in ingresso abbiamo il clock, poiché in corrispondenza del fronte di salita del clock avremo lo shift. Tuttavia, è presente anche un segnale di abilitazione (en), che *stabilisce quando è valido il dato che mettiamo in ingresso*. Anche l'enable è attivo sul fronte di salita, e abbiamo simulato questa caratteristica *salvando il valore precedente di en in un signal lastEn*: il fronte di salita si presenta quando lastEn è 0 e en è 1.

```
1  entity shift_reg_behav is
2      port(
3          clock : in std_logic;
4          rst : in std_logic;
5          dir : in std_logic; --0(destra), 1(sinistra)
6          mode : in std_logic; --0(un bit), 1(due bit)
```

```
7      ing : in std_logic;  
8      y : out std_logic_vector(0 to 3);  
9      en : in std_logic  
10     );  
11 end shift_reg_behav;
```

A livello progettuale, abbiamo deciso di implementare uno shift register SIPO a 4 bit che permette di shiftare a destra o a sinistra di uno o due bit.

Il process è sensibile al clock, per permettere lo shift solo sul fronte di salita del clock. Quando si presenta il fronte di salita dell'abilitazione, in funzione della particolare configurazione di dir e mode, avrò lo shift desiderato.

Per specificare il fronte di salita del segnale enable non abbiamo utilizzato il costrutto rising edge perché quest'ultimo è *specifico per il segnale di clock e non per i segnali di logica implementati dall'utente*, quindi da un punto di vista di sintesi *verrà trattato in modo particolare*.

Notiamo inoltre che per realizzare lo shift a due bit, essendo l'input seriale, abbiamo aggiunto uno zero riempitivo.

Successivamente, abbiamo definito un segnale interno all'architecture temp che rappresenta *lo stato dei flip flop che compongono lo shift register*. Poiché preleviamo l'uscita in parallelo, in uscita avremo lo stato finale di tutti i flip flop dopo lo shift. L'assegnazione uscita \leftarrow temp, in funzione di dove viene fatta, *assume un diverso significato*:

- se la faccio fuori dal process, *verrà eseguita in maniera concorrente*;
- se la faccio nel process, all'esterno del if riguardante il fronte di salita del clock, *il valore dell'uscita verrà aggiornato alla prossima attivazione del process*, quindi sul fronte di discesa successivo a quando è stato aggiornato temp(3);
- se la faccio nel process, all'interno del if riguardante il clock, *l'uscita non verrà aggiornata contestualmente a temp(3)* e ci vorrà un altro fronte del clock per vedere l'uscita. Quindi, per conservare il valore di temp, verrà utilizzato un ulteriore flip flop.

Analizzando la parte di codice che riguarda lo shift, notiamo che è possibile evitare la sovrascrittura dei valori memorizzati nei flip flop precedenti, sfruttando una caratteristica dei signal, ossia il fatto che *aggiornano il proprio valore al colpo di clock successivo* (quindi l'aggiornamento viene chiesto in un certo istante e viene fatto in un istante successivo).

```

1 architecture Behavioral of shift_reg_behav is
2
3 signal lastEn : std_logic := '0';
4 signal temp : std_logic_vector(0 to 3) := "0000";
5
6 begin
7 process(clock)

```

```

8      begin
9
10     if rising_edge(clock) then
11        if(rst = '1') then
12          y(0 to 3) <= "0000";
13        elsif(en = '1' and lastEn = '0') then
14          if (dir = '0' and mode = '0') then --shift di un bit
15            verso destra
16            temp(0) <= ing;
17            temp(1) <= temp(0);
18            temp(2) <= temp(1);
19            temp(3) <= temp(2);
20        elsif(dir = '0' and mode = '1') then --shift di due bit verso
21          destra
22          temp(0) <= '0';
23          temp(1) <= ing;
24          temp(2) <= temp(1);
25          temp(3) <= temp(2);
26        elsif(dir = '1' and mode = '0') then --shift di un bit verso
27          sx
28          temp(3) <= ing;
29          temp(2) <= temp(3);
30          temp(1) <= temp(2);
31          temp(0) <= temp(1);
32        elsif(dir = '1' and mode = '1') then --shift di due bit verso
33          sx
34          temp(3) <= '0';
35          temp(2) <= ing;

```

```

32      temp(1) <= temp(2);
33      temp(0) <= temp(1);
34      end if;
35  end if;
36
37      lastEn <= en
38      end if;
39      y(0 to 3) <= temp(0 to 3);
40  end process;
41 end Behavioral;
```

Testing del sistema

In allegato la simulazione dello shift register realizzato con un approccio comportamentale mettendo in input la sequenza *1011*.

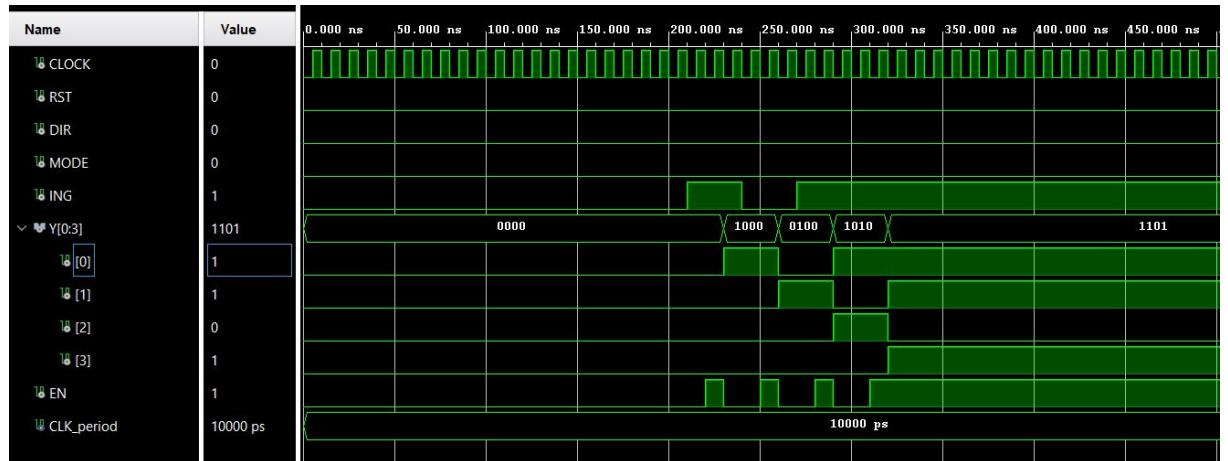


Figura 2.7: Simulazione shift register behavioral

2.4.2 Implementazione structural

L'implementazione strutturale dello shift register ha previsto la composizione della macchina a partire da macchine più semplici. In particolare, sono stati necessari i seguenti componenti:

- flip flop D;
- multiplexer 4:1, già realizzato precedentemente.

Flip flop D

Il flip flop D è un flip flop *a memorizzazione dell'ingresso* ed è di tipo *edge triggered* nel nostro caso. Il flip flop presenta in uscita il valore che ha in ingresso, in corrispondenza del fronte di salita del clock e del fronte di salita dell'abilitazione. Inoltre, è dotato di un segnale di reset, necessario per *inizializzare la macchina ad un valore noto*.

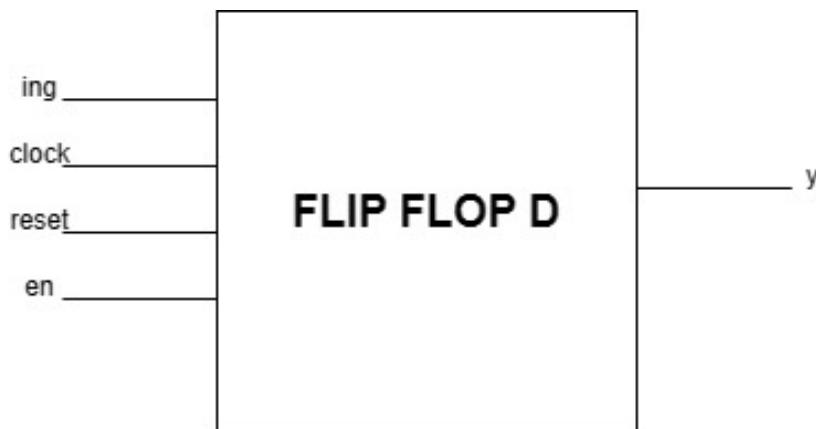


Figura 2.8: Schema flip flop D

Abbiamo realizzato il flip flop in maniera comportamentale. La board Nexys A7-100T, di default è dotata di flip flop di tipo D, per questo motivo abbiamo deciso di implementare proprio questa tipologia di flip flop. In fase di sintesi, indipendentemente dalla tipologia di flip flop implementata, sulla board verrà comunque implementato attraverso flip flop di tipo D. Osserviamo la presenza del segnale interno lastEn, che ci permette di "simulare" il fronte di salita dell'abilitazione.

```
1 entity flip_flop_d is
2 port(
3     clk : in std_logic;
4     en : in std_logic;
5     input : in std_logic;
6     output : out std_logic;
7     rst : in std_logic
8 );
9 end flip_flop_d;
10
11 architecture Behavioral of flip_flop_d is
12
13 signal lastEn : std_logic := '0';
14
15 begin
16 process(clk)
17 begin
18     if(rising_edge(clk)) then
19         if(rst = '1') then
```

```
20      output <= '0';  
21  
22      end if;  
23  
24      if( en = '1' and lastEn = '0' ) then  
25          output <= input;  
26      end if;  
27  
28  end process;  
29  end Behavioral;
```

Realizzazione del sistema complessivo

L'elemento fondamentale per la realizzazione di questa variante del classico shift register è stato il multiplexer 4:1. Infatti, l'uscita di un flip flop non va direttamente in ingresso al flip flop successivo, ma viene elaborata *in funzione della particolare configurazione direzione-modo che stiamo utilizzando in quel momento*. Ad ogni flip flop è associato un multiplexer 4:1 e l'uscita del multiplexer i-esimo va in ingresso al flip flop i-esimo. Osserviamo che le linee di ingresso di ogni multiplexer non sono altro che *tutti i possibili stati in cui quel flip flop può andare a trovarsi*, in funzione del particolare modo e della particolare direzione, i cui valori fungono da abilitazione.

Per chiarire meglio l'idea, prendiamo in esame un singolo multiplexer (ad esempio il primo), poi il discorso può essere facilmente esteso a n multiplexer.

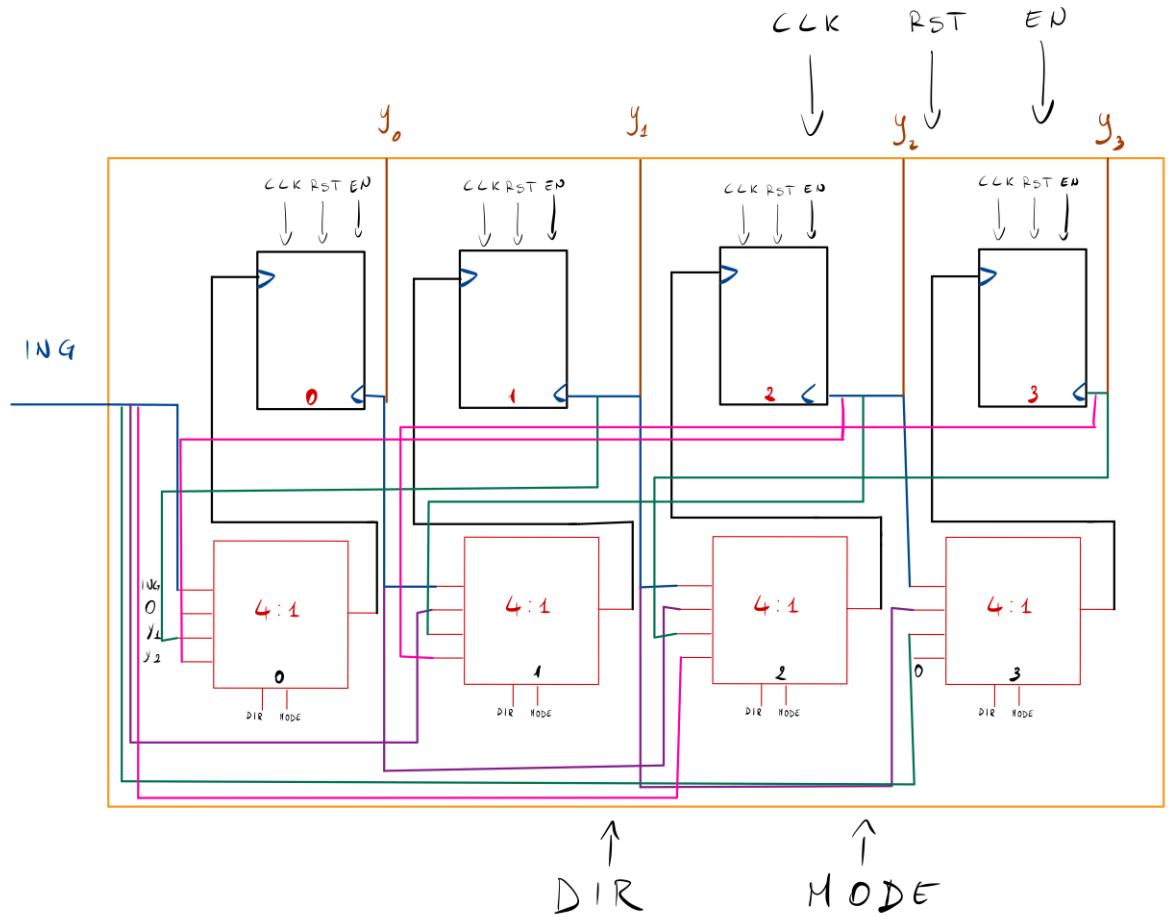


Figura 2.9: Schematic dello shift register realizzato con un approccio strutturale

Consideriamo tutti i possibili valori che possono assumere le due selezioni:

- caso 00 (shift di un bit verso dx): il primo flip flop dovrà avere in ingresso proprio il dato che voglio inserire nello shift register (ing);

- caso 01 (shift di due bit verso dx): il primo flip flop dovrà avere in ingresso un bit "riempitivo" (0 in questo caso), poiché stiamo shiftando di due bit ma in ingresso (essendo seriale) presento un solo valore alla volta;
- caso 10 (shift di un bit verso sx): il primo flip flop dovrà avere in ingresso l'uscita del secondo flip flop (Y1);
- caso 11 (shift di due bit verso sx): il primo flip flop dovrà avere in ingresso l'uscita del terzo flip flop (Y2).

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity shift_reg_structural is
5 port(
6     clock : in std_logic;
7     rst : in std_logic;
8     dir : in std_logic; --destra o sinistra
9     mode : in std_logic; --shift di un bit o due bit
10    ing : in std_logic;
11    y : out std_logic_vector(0 to 3);
12    en : in std_logic
13 );
14 end shift_reg_structural;
15
16 architecture Structural of shift_reg_structural is
17

```

```
18 signal output_mux : std_logic_vector(0 to 3); —segnale interno che
19   rappresenta le uscite dei mux
20
21 component mux_4_1 is
22 port(
23   input_a : in std_logic_vector(0 to 3);
24   input_s : in std_logic_vector(1 downto 0);
25   output_y : out std_logic
26 );
27 end component;
28
29 component flip_flop_d is
30 port(
31   clk : in std_logic;
32   en : in std_logic;
33   input : in std_logic;
34   output : out std_logic;
35   rst : in std_logic
36 );
37 end component;
38
39 begin
40
41   y(0 to 3) <= output_ff(0 to 3);
42
43   mux_0 : mux_4_1
44   port map(
```

```
45  input_a(0) => ing,
46  input_a(1) => '0', --unspecified
47  input_a(2) => output_ff(1),
48  input_a(3) => output_ff(2),
49  input_s(0) => dir,
50  input_s(1) => mode,
51  output_y => output_mux(0)
52 );
53
54 mux_1 : mux_4_1
55 port map(
56  input_a(0) => output_ff(0),
57  input_a(1) => ing,
58  input_a(2) => output_ff(1),
59  input_a(3) => output_ff(3),
60  input_s(0) => dir,
61  input_s(1) => mode,
62  output_y => output_mux(1)
63 );
64
65 mux_2 : mux_4_1
66 port map(
67  input_a(0) => output_ff(1),
68  input_a(1) => output_ff(0),
69  input_a(2) => output_ff(3),
70  input_a(3) => ing,
71  input_s(0) => dir,
72  input_s(1) => mode,
```

```

73     output_y    => output_mux(2)
74 );
75
76 mux_3 : mux_4_1
77 port map(
78     input_a(0) => output_ff(2),
79     input_a(1) => output_ff(1),
80     input_a(2) => ing,
81     input_a(3) => '0', --unspecified
82     input_s(0) => dir,
83     input_s(1) => mode,
84     output_y    => output_mux(3)
85 );
86
87 ff_0 : flip_flop_d
88 port map(
89     clk => clock,
90     en => en,
91     input => output_mux(0),
92     output => output_ff(0),
93     rst => rst
94 );
95
96 ff_1 : flip_flop_d
97 port map(
98     clk => clock,
99     en => en,
100    input => output_mux(1),

```

```
101     output => output_ff(1),
102     rst => rst
103 );
104
105 ff_2 : flip_flop_d
106 port map(
107     clk => clock,
108     en => en,
109     input => output_mux(2),
110     output => output_ff(2),
111     rst => rst
112 );
113
114 ff_3 : flip_flop_d
115 port map(
116     clk => clock,
117     en => en,
118     input => output_mux(3),
119     output => output_ff(3),
120     rst => rst
121 );
122
123 end Structural;
```

2.4.3 Testing dei componenti

Un sistema grande è composto da sistemi più piccoli, seguendo un approccio incrementale. Per passare alla progettazione di un componente

più grande, bisogna prima *testare i componenti più piccoli*. Infatti, alla fine del progetto, abbiamo i testbench del multiplexer 4:1, del flip flop D e solo alla fine del sistema complessivo.



Figura 2.10: Simulazione flip flop D

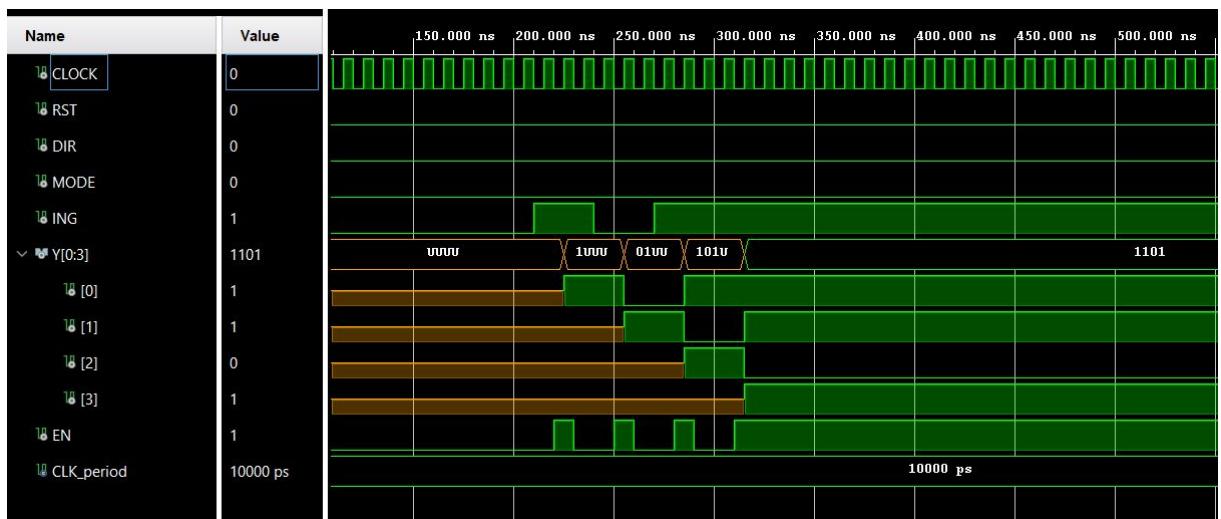


Figura 2.11: Simulazione shift register strutturale

2.5 Esercizio 5.1 – Cronometro

Il cronometro è un dispositivo che permette di contare il tempo trascorso a partire da un certo valore. Il nostro progetto prevede la possibilità di inizializzare il cronometro da 0 (con un segnale di reset) oppure da un valore noto impostato dall’utente (tramite un segnale di set).

Il cronometro è costituito da 3 contatori, uno per i secondi, uno per i minuti e uno per le ore.

Contatore

Abbiamo deciso di realizzare il contatore seguendo un approccio comportamentale. Il contatore è stato realizzato mediante l’utilizzo di un generic. I generic sono molto utili ad esempio quando utilizziamo componenti che svolgono la stessa funzione ma su input di dimensione diversa. I parametri che cambiano nel nostro caso sono il numero di bit su cui il contatore deve presentare il risultato (6 per i secondi e i minuti, 5 per le ore) e il modulo del contatore (60 per i secondi e i minuti e 24 per le ore).

```
1 entity contatore is
2 Generic(
3     bits : integer := 100; -- n bits per rappresentare il risultato
4     modulo : integer := 60 -- modulo del contatore
5 );
6
```

```
7 Port(  
8     clk : in std_logic;  
9     rst : in std_logic;  
10    en : in std_logic;  
11    set_en : in std_logic;  
12    set : in std_logic_vector(0 to bits-1); —  $2^6$   
13    uscita : out std_logic;  
14    numero : out std_logic_vector(0 to bits-1)  
15 );  
16 end contatore;
```

Come possiamo notare dalle entity del contatore, abbiamo un segnale di abilitazione globale delle macchina, en, e un segnale di abilitazione del preset, *seten*. Set invece contiene il valore esplicito a cui vogliamo inizializzare il contatore. Questo valore verrà utilizzato per inizializzare il contatore solo se *seten* è alto.

L'uscita assume il valore 1 non quando raggiunge il valore modulo-1 ma quando assume il valore modulo-2, per permettere una corretta propagazione dell'uscita del contatore precedente all'ingresso del contatore successivo, considerando che i contatori commutano sul fronte di discesa del clock. Il valore di conteggio invece torna a 0 quando il contatore è arrivato al valore modulo-1, quindi sul fronte di discesa del clock successivo il contatore inizierà a contare da capo.

Infine abbiamo l'uscita numero, che rappresenta lo stato del conteggio in un certo momento del contatore, rappresentato su 5 o 6 bit a seconda se è il contatore delle ore, dei minuti o dei secondi.

Il reset è stato implementato in modo asincrono. Un segnale è asincrono quando la variazione del suo valore causa una variazione dello stato del sistema indipendentemente dal segnale di conteggio.

```

1 architecture Behavioral of contatore is
2 begin
3   process(clk)
4     variable count : integer := 0;
5     begin
6       if(rst = '1') then -- reset asincrono
7         count := 0;
8         numero <= std_logic_vector(to_unsigned(0, bits));
9       elsif(set_en = '1') then
10        count := to_integer(unsigned(set));
11      elsif(falling_edge(clk)) then
12        if(en = '1') then
13          uscita <= '0';
14          if(count = modulo-2) then
15            uscita <= '1';
16          end if;
17          if(count = modulo-1) then
18            count := 0;
19          else
20            count := count+1;
21          end if;
22        end if;
23      end if;
24      numero <= std_logic_vector(to_unsigned(count, bits));

```

```
25    end process;
```

```
26 end Behavioral;
```

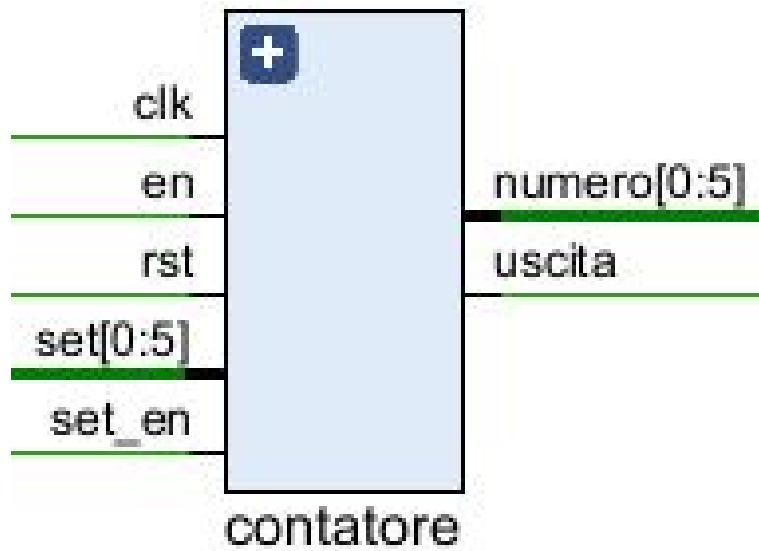


Figura 2.12: Schematic del contatore

Divisore di frequenza

Il divisore di frequenza è uno dei possibili utilizzi di un contatore. Dato un clock di ingresso ad una certa frequenza permette di ottenere in uscita un clock con una frequenza differente. Basa il suo funzionamento sul conteggio del numero di fronti di clock da aspettare prima di alzare l'uscita, che può essere ottenuto come rapporto tra la frequenza del clock in ingresso e la frequenza del clock desiderata.

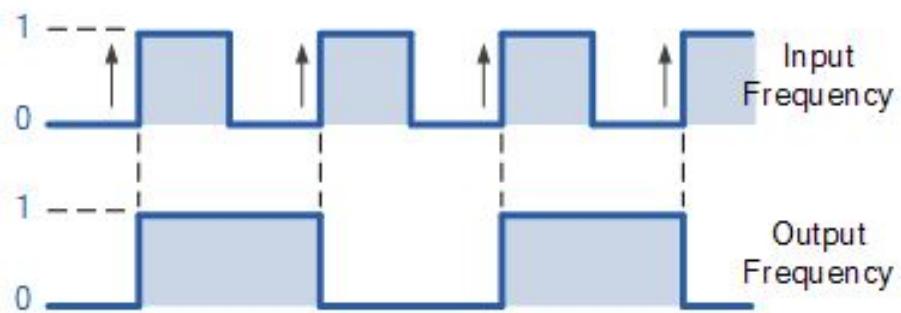


Figura 2.13: Funzionamento del divisore di frequenza

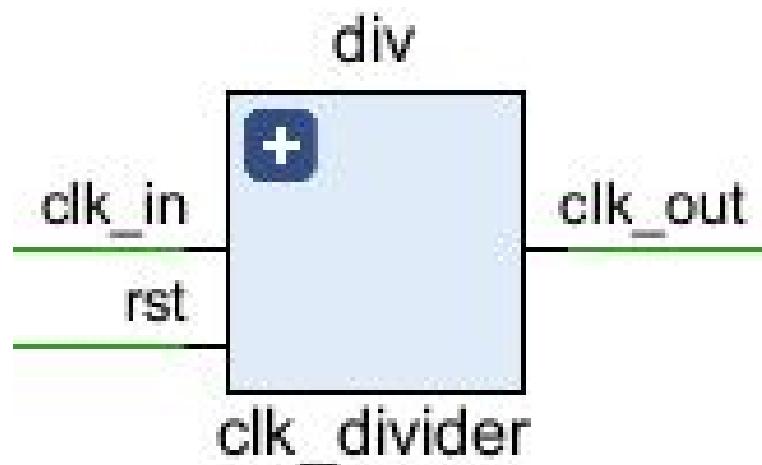


Figura 2.14: Schematic divisore di frequenza

Sistema complessivo: cronometro strutturale

Il cronometro prevede come ingressi quelli indicati nella entity che segue. Notiamo la presenza di 3 segnali di ingresso che consentono il preset della macchina ad un valore noto e 3 segnali di uscita per visualizzare lo stato attuale del conteggio.

```
1 entity cronometro is
2   Port (
3     CLK      : in std_logic;
4     RST      : in std_logic;
5     EN       : in std_logic;
6     set_EN    : in std_logic;           — abilitazione del preset
7     set_h    : in std_logic_vector(0 to 4);
8     set_m    : in std_logic_vector(0 to 5);
9     set_s    : in std_logic_vector(0 to 5);
10    h       : out std_logic_vector(0 to 4);
11    m       : out std_logic_vector(0 to 5);
12    s       : out std_logic_vector(0 to 5)
13  );
14 end cronometro;
```

Per la realizzazione del sistema complessivo abbiamo seguito un approccio strutturale collegando in parallelo 3 contatori, uno per i secondi, uno per i minuti e uno per le ore. I contatori devono essere costituiti da flip flop edge triggered sul fronte di discesa: edge triggered perché devono commutare solo in istanti ben precisi, sul fronte di discesa perché altrimenti appena commuta il primo commuterebbe

anche il secondo e così via.

Con lo schema parallelo, il primo contatore, quello dei secondi, commuta ogni volta che arriva il clock, il secondo contatore, quello dei minuti, invece quando arriva il clock e l'uscita del contatore dei secondi è alta e così via. Per realizzare questo comportamento, basta mettere in ingresso il clock a tutti i contatori e metterlo in AND con le uscite dei contatori precedenti. In questo modo si evita anche il ritardo di propagazione tipico dello schema seriale.

Osserviamo inoltre che è necessario un divisore di frequenza affinché i contatori possano correttamente scandire i secondi, i minuti e le ore. Il divisore di frequenza prende una frequenza in ingresso pari alla frequenza della board (100 MHz) e restituisce in uscita un frequenza di 1 Hz, per permettere al contatore dei secondi di incrementare il valore di conteggio ogni secondo. I contatori successivi commuteranno rispettivamente ogni minuto e ogni ora, grazie all'uscita del precedente contatore e al clock in ingresso.

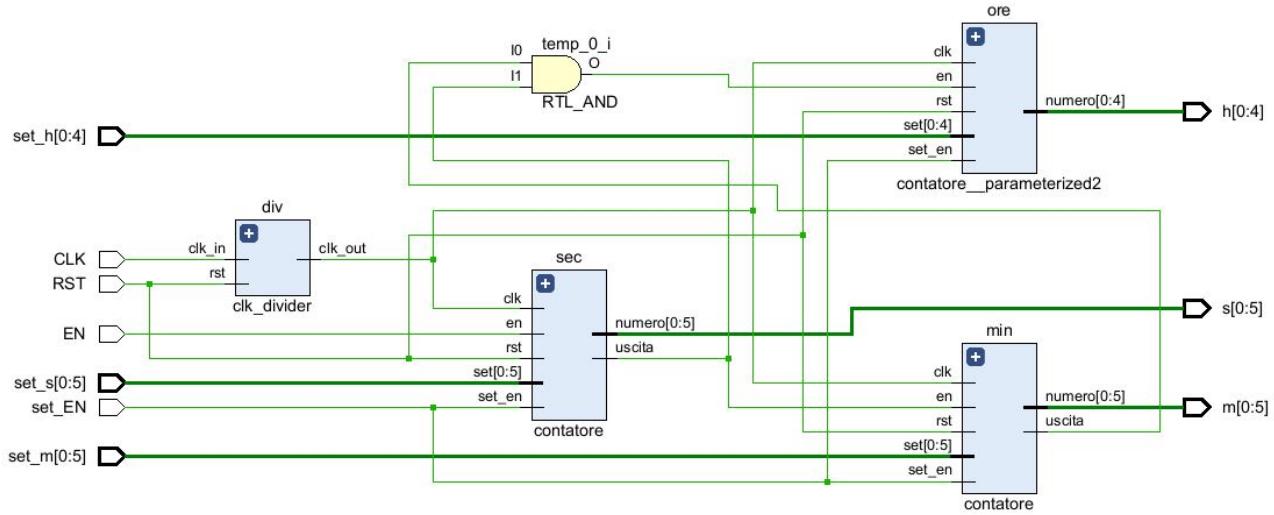


Figura 2.15: Schematic del cronometro

```

1 architecture Structural of cronometro is
2
3 component clk_divider is
4     Generic(
5         clk_freq_in : integer := 100000000; --100Mhz della board
6         clk_freq_out: integer := 1           --1Mhz uscita desiderata
7     );
8
9     Port(
10        clk_in : in std_logic; -- board clock
11        rst : in std_logic;
12        clk_out : out std_logic
13    );
14 end component clk_divider;
15
16 component contatore is
17     Generic(

```

```

18     bits : integer := 100; -- n bits per rappresentare il risultato
19     modulo : integer := 60 -- modulo del contatore
20 );
21
22 Port(
23     clk : in std_logic;
24     rst : in std_logic;
25     en : in std_logic;
26     set_en : in std_logic;
27     set : in std_logic_vector(0 to bits-1); -- 2^6
28     uscita : out std_logic;
29     numero : out std_logic_vector(0 to bits-1)
30 );
31 end component contatore;
32
33 signal y : std_logic_vector(0 to 2) := "000";
34 signal clk_out_tmp : std_logic := '0';
35 signal temp : std_logic_vector(0 to 1);
36 begin
37     div : clk_divider
38         generic map(
39             clk_freq_in => 100000000,
40             clk_freq_out => 1
41         )
42         port map(
43             clk_in => CLK,
44             rst => RST,
45             clk_out => clk_out_tmp

```

```
46      );
47
48  sec : contatore
49
50    generic map(
51      bits => 6,
52      modulo => 60
53    )
54
55    port map(
56      clk => clk_out_tmp,
57      rst => RST,
58      en => EN,
59      set_en => set_EN,
60      set => set_s,
61      uscita => y(0),
62      numero => s
63    );
64
65  min : contatore
66
67    generic map(
68      bits => 6,
69      modulo => 60
70    )
71
72    port map(
73      clk => clk_out_tmp,
```

```
74      set_en => set_EN,
75      set => set_m,
76      uscita => y(1),
77      numero => m
78  );
79
80      temp(1) <= y(1) and y(0);
81
82  ore : contatore
83  generic map(
84      bits => 5,
85      modulo => 24
86  )
87  port map(
88      clk => clk_out_tmp,
89      rst => RST,
90      en => temp(1),
91      set_en => set_EN,
92      set => set_h,
93      uscita => y(2),
94      numero => h
95  );
96
97 end Structural;
```

2.5.1 Testing del sistema complessivo

Nell'esercizio in questione, in fase di prova, abbiamo realizzato il sistema complessivo senza aggiungere il divisore di frequenza, perché essendo richiesta solo la simulazione senza l'implementazione sulla board, è stato sufficiente generare un clock di 1 Hz nel testbench. Nonostante ciò, nel testbench abbiamo utilizzato un clock di 0.1 ns per velocizzare la simulazione. Infatti, la generazione di una simulazione di durata di un secondo, il minimo che servirebbe per verificare il corretto funzionamento del clock, richiede molto tempo. Quindi, abbiamo velocizzato il clock per permettere la visualizzazione grafica del corretto funzionamento del cronometro.

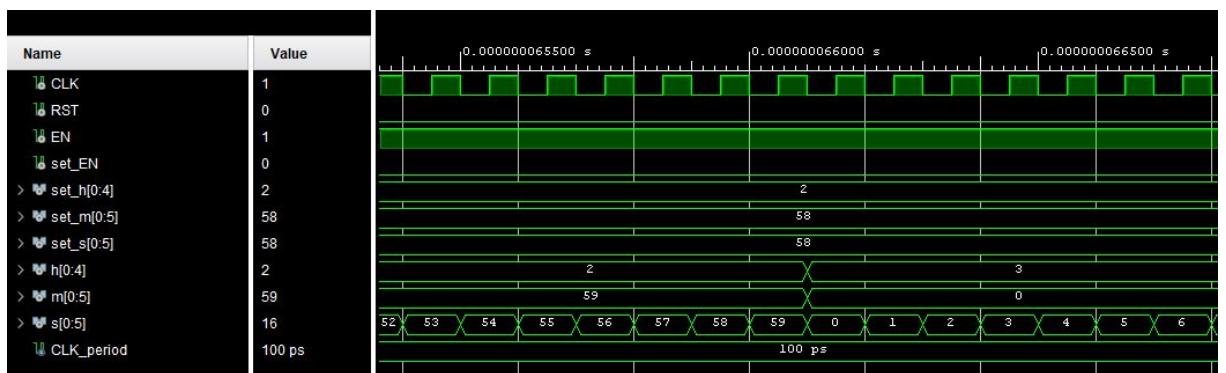


Figura 2.16: Simulazione del cronometro

2.6 Esercizio 5.2 – Cronometro onboard

Il seguente punto prevede di poter caricare un orario da cui il cronometro possa partire, andando ad impostare ore, minuti e secondi.

2.6.1 Caricamento dell'orario

Per poter caricare in memoria l'orario si è fatto ausilio degli switch sulla board, in particolare sono stati utilizzati 3 switch per la selezione, mentre per caricare il valore vengono utilizzati 6 switch. Si osserva che per le ore bastano solo 5 bit, arrivando ad un massimo di 23, a differenza dei minuti e dei secondi, i quali arrivano fino a 59; dunque vi sarà solo una lieve differenza a livello implementativo.

Unità di controllo

Tramite l'unità di controllo abbiamo gestito l'input. Il valore viene caricato all'interno di un vettore che funge da buffer, `value_in`, di 6 bit, siccome il valore massimo rappresentabile è 59.

```
1  entity control_unit is
2      Port(
3          clk      : in std_logic;
4          rst      : in std_logic;
5          set_btn  : in std_logic;
6          sel_h    : in std_logic;
7          sel_m    : in std_logic;
8          sel_s    : in std_logic;
9          set_h    : out std_logic_vector(0 to 4);
10         set_m   : out std_logic_vector(0 to 5);
11         set_s   : out std_logic_vector(0 to 5);
12         value_in : in std_logic_vector(0 to 5) -- input buffer
13     );
14 end control_unit;
```

A seconda delle selezione quindi viene copiato il valore dal buffer al vettore apposito di uscita, il quale può essere `set_h`, `set_m` oppure `set_s`. Riportiamo quanto segue l'implementazione di quanto detto.

```
1  input : process(clk, rst)
2
3      if rst = '1' then
4
5          set_h <= (others => '0');
6
7          set_m <= (others => '0');
8
9          set_s <= (others => '0');
10
11     end if;
12
13
14     if clk'event AND clk = '1' then
15
16         if set_btn = '1' then
17
18             if sel_h = '1' then
19
20                 set_h(0 to 4) <= value_in(0 to 4);
21
22             elsif sel_m = '1' then
23
24                 set_m <= value_in;
25
26             elsif sel_s = '1' then
27
28                 set_s <= value_in;
29
30             end if;
31
32         end if;
33
34     end if;
35
36 end process;
```

Per caricare l'orario dunque, bisogna prima abilitare una delle tre selezioni, impostare l'ingresso desiderato, ed infine bisogna premere il

bottone apposito per acquisire l'input. Si nota infatti come alla riga 10 venga controllato se tale bottone, tramite il segnale `set_btn`, sia stato premuto.

Rappresentazione decimale

Per rappresentare l'ora in formato decimale è stato implementato un convertitore, il quale prende in ingresso l'uscita del cronometro, di 6 bit, e fornisce come uscita un segnale di 8 bit. Questo perché per rappresentare in decimale un numero a due cifre, avendo 4 bit per rappresentare ciascuna di essa, abbiamo bisogno di massimo $2 \cdot 4 = 8$ bit.

```
1  entity converter is
2
3      Port (
4          clk      :  in  std_logic;
5          vIn      :  in  std_logic_vector(0 to 5);    — ingresso
6          vOut     :  out std_logic_vector(7 downto 0) — uscita
7      );
8
9  end converter;
```

Per generare il segnale quindi andiamo a verificare quale sia il valore del segnale in ingresso, dopodiché in base ad esso bisogna andare a scrivere la cifra per l'unità, i primi 4 bit, e l'eventuale cifra per la decina. Dunque viene fatta una prima verifica per capire quale valore inserire in base al valore dell'unità, righe 5–16, mentre le righe 22–27 si occupano della cifra per la decina.

```

1 process(clk)
2 begin
3     if rising_edge(clk) then
4         -- scrivi i primi 4 bit
5         case to_integer(unsigned(vIn)) is
6             when 0|10|20|30|40|50 => vOut(3 downto 0) <= "0000";
7             when 1|11|21|31|41|51 => vOut(3 downto 0) <= "0001";
8             when 2|12|22|32|42|52 => vOut(3 downto 0) <= "0010";
9             when 3|13|23|33|43|53 => vOut(3 downto 0) <= "0011";
10            when 4|14|24|34|44|54 => vOut(3 downto 0) <= "0100";
11            when 5|15|25|35|45|55 => vOut(3 downto 0) <= "0101";
12            when 6|16|26|36|46|56 => vOut(3 downto 0) <= "0110";
13            when 7|17|27|37|47|57 => vOut(3 downto 0) <= "0111";
14            when 8|18|28|38|48|58 => vOut(3 downto 0) <= "1000";
15            when 9|19|29|39|49|59 => vOut(3 downto 0) <= "1001";
16            when others => vOut(3 downto 0) <= "1111";
17        end case;
18
19
20        -- scrivi i secondi 4 bit
21        case to_integer(unsigned(vIn)) is
22            when 0 to 9 => vOut(7 downto 4) <= "0000";
23            when 10 to 19 => vOut(7 downto 4) <= "0001";
24            when 20 to 29 => vOut(7 downto 4) <= "0010";
25            when 30 to 39 => vOut(7 downto 4) <= "0011";
26            when 40 to 49 => vOut(7 downto 4) <= "0100";
27            when 50 to 59 => vOut(7 downto 4) <= "0101";
28            when others => vOut(7 downto 4) <= "1111";

```

```

29  end case;
30
31 end process;
```

Ad esempio, se avessimo in ingresso $vIn = 1111_2 = 15_{10}$, allora nei primi 4 bit del segnale $vOut$ saranno dedicati a rappresentare il numero 5, andando a scrivere $0101_2 = 5_{10}$, infatti da come si nota alla riga 11 del codice. Dopodiché siccome per la decina si ha il numero 1, si ricade nella condizione presente alla riga 23, andando a scrivere nei secondi 4 bit $0001_2 = 1_{10}$. In fine si ottiene $vOut = 0001\ 0101$.

Top module – Cronometro on board

Viene dato un segnale di abilitazione della macchina complessiva, EN, per poter far partire il conteggio. Il caricamento dell'orario avviene prima di attivare l'abilitazione complessiva, andando ad alzare gli opportuni switch di selezione e utilizzando il bottone apposito, set_btn, per poter prelevare il valore.

```

1 entity cronometro_onboard is
2
3     Port (
4         CLK      : in std_logic;
5         RST      : in std_logic;
6         EN       : in std_logic; -- enable per cronometro
7         set_btn  : in std_logic; -- enable per l'input
8         sel_H    : in std_logic; -- selezioni per l'ora
9         sel_M    : in std_logic; -- selezione per i minuti
10        sel_S   : in std_logic; -- selezione per i secondi
```

```

10      value_in      :  in  std_logic_vector(0 to 5); -- input buffer
11      cathodes      :  out std_logic_vector(7 downto 0);
12      anodes        :  out std_logic_vector(7 downto 0)
13  );
14 end cronometro_onboard;

```

Vengono utilizzati i seguenti segnali intermedi.

```

1  -- segnale pulito del bottone
2  signal set_clrd : std_logic := '0';
3
4  -- segnali per le uscite della control unit
5  signal cu_h_out : std_logic_vector(0 to 4) := (others => '0');
6  signal cu_m_out : std_logic_vector(0 to 5) := (others => '0');
7  signal cu_s_out : std_logic_vector(0 to 5) := (others => '0');
8
9  -- segnali per le uscite del cronometro
10 signal cronos_h_out : std_logic_vector(0 to 4) := (others => '0');
11 signal cronos_m_out : std_logic_vector(0 to 5) := (others => '0');
12 signal cronos_s_out : std_logic_vector(0 to 5) := (others => '0');
13
14 -- segnali per le uscite del cronometro convertite in decimale
15 signal cronos_h_dec : std_logic_vector(0 to 7) := (others => '0');
16 signal cronos_m_dec : std_logic_vector(0 to 7) := (others => '0');
17 signal cronos_s_dec : std_logic_vector(0 to 7) := (others => '0');

```

Nel caso venga caricato un orario, esso verrà messo in ingresso ai segnali per le uscite della control unit, tali uscite andranno in ingresso ai segnali set del cronometro. Nel caso non venga inserito nessun orario

i segnali rimarranno comunque a zero. Il cronometro a sua volta come uscite saranno mappati negli appositi segnali, questi ultimi andranno poi in ingresso al convertitore, che si occupa della rappresentazione in decimale sul display. Infine le uscite del convertitore andranno in ingresso al display a sette segmenti.

```

1 crinos : cronometro

2     Port map(

3         CLK      => CLK,
4         RST      => RST,
5         EN       => EN,
6         set_EN   => set_clrd, --il bottone funge da segnale di enable
7             , alla sua pressione vengono caricati i valori h, m ed s
8         set_h    => cu_h_out,
9         set_m    => cu_m_out,
10        set_s    => cu_s_out,
11        h       => crinos_h_out,
12        m       => crinos_m_out,
13        s       => crinos_s_out

14     );

```

Per il port mapping del converter si osserva che per l'ora si ha il primo bit impostato a zero, essendo che a differenza dei minuti e dei secondi necessita di un bit in meno.

```

1 conv_s : converter

2     Port map(

```

```

3      clk      => clk,
4      vIn      => cronos_s_out,
5      vOut     => cronos_s_dec
6      );
7
8 conv_m : converter
9
10 Port map(
11
12     clk      => clk,
13
14     vIn      => cronos_m_out,
15
16     vOut     => cronos_m_dec
17
18 );
19
20 conv_h : converter
21
22 Port map(
23
24     clk      => clk,
25
26     vIn(1 to 5) => cronos_h_out,
27
28     vIn(0)      => '0',
29
30     vOut      => cronos_h_dec
31
32 );

```

Nel display vengono accese solo 6 cifre a partire da destra, le ultime due rimangono spente. In ingresso al display, value_32, le uscite dei convertitori.

```

1 watch : display_seven_segments
2
3 Generic map(
4
5     clock_frequency_in  => 100000000,
6
7     clock_frequency_out => 500
8
9 );

```

```
6  Port map(  
7      clock                  => CLK,  
8      reset                  => RST,  
9      value32_in(7  downto 0)  => cronos_s_dec,  
10     value32_in(15 downto 8)  => cronos_m_dec,  
11     value32_in(23 downto 16) => cronos_h_dec,  
12     value32_in(31 downto 24) => (others => '0'),  
13     enable                 => "00111111", -- le prime due  
14          cifre vengono spente  
15     dots                   => "00000000",  
16     cathodes              => cathodes,  
17     anodes                 => anodes  
18 );
```

2.7 Esercizio 5.3 – Cronometro con intertempo

Per implementare l'intertempo è stato fatto uso di una memoria in modo che contenesse l'orario, il quale viene salvato alla pressione di un bottone apposito.

Memoria

La memoria è composta da 8 locazioni di memoria, ognuna di essa deve essere grande $3 \cdot 8 = 24$ bit per poter contenere ore, minuti e secondi. Viene utilizzato un contatore per incrementare l'indirizzo di memoria della nostra memoria. L'indirizzo viene incrementato ogni volta che viene premuto il bottone apposito, mandato a sua volta in ingresso come segnale di abilitazione del contatore

```
1 entity counter is
2     Port(
3         clock    : in std_logic;
4         reset    : in std_logic;
5         enable   : in std_logic;
6         counter : out std_logic_vector(0 to 2);
7         uscita   : out std_logic
8     );
9 end counter;
```

La memoria ha in ingresso dunque un vettore, `value`, di 24 bit. Il segnale per la scrittura è dato da `write`, il quale sarà abilitato dal bottone.

```

1 entity Mem is
2   Port (
3     clock    :  in std_logic;
4     reset    :  in std_logic;
5     write    :  in std_logic;
6     addr     :  in std_logic_vector(0 to 2);
7     value    :  in std_logic_vector(0 to 23)
8   );
9 end Mem;
```

All'atto del reset la memoria viene azzerata, mentre se il segnale `write` è alto allora scrive in memoria all'indirizzo dato da `addr`.

```

1 type mem_type is array (0 to 7) of std_logic_vector(0 to 2);
2 signal MEM : mem_type;
3
4 begin
5   process(clock)
6   begin
7     if rising_edge(clock) then
8       if(reset = '1') then
9         init: for i in 0 to 7 loop
10        MEM(i) <= (others => '0');
```

```

11      end loop init;
12
13      end if;
14
15      if(write = '1') then
16          MEM(to_integer(unsigned(addr))) <= value;
17
18  end process;
```

Nel top module è stato inserito un segnale intermedio aggiuntivo.

```

1 -- segnale per la memoria
2 signal addr_temp : std_logic_vector(0 to 2) := (others => '0');
```

L'uscita del contatore andrà nel seguente segnale, in quale a sua volta verrà dato alla memoria.

```

1 count : counter
2
3     Port map(
4         clock    => CLK,
5         reset    => RST,
6         enable   => show_clrd,
7         counter  => addr_temp
8     );
9
10
11 memory : Mem
12
13     Port map(
14         clock => CLK,
```

```
12     reset => RST,  
13     write => show_clrd,  
14     addr  => addr_temp,  
15     value(0 to 7) => cronos_s_dec,  
16     value(8 to 15) => cronos_m_dec,  
17     value(16 to 23) => cronos_h_dec  
18   );
```

2.8 Esercizio 6.1 – Sistema di testing

Per la realizzazione di questo progetto abbiamo realizzato un sistema in grado di testare in maniera automatica una semplice macchina combinatoria, confrontando i risultati attesi con i risultati forniti dalla macchina combinatoria tramite un comparatore e restituendo 1 se i due risultati coincidono, 0 altrimenti.

Data la natura dell'esercizio, abbiamo seguito un approccio strutturale, suddividendo il sistema in unità operativa e unità di controllo:

- L'**unità operativa** è costituita dall'insieme dei componenti architettonici che implementano in pratica le operazioni richieste
- L'**unità di controllo** è la parte del sistema che ne definisce il comportamento, impartendo opportuni comandi alla parte operativa.

Possiamo realizzare l'unità di controllo in due modi:

- **Logica cablata:** progettiamo l'unità di controllo in termini di porte logiche;
- **Logica microprogrammata:** progettiamo l'unità di controllo con una micro-ROM e una logica che la controlli.

Intuitivamente, *è necessario sviluppare prima la parte operativa e solo dopo l'unità di controllo*. Questo perché non si può controllare un qualcosa che non è stato ancora definito e implementato.

2.8.1 Unità operativa

Abbiamo realizzato l'unità operativa mediante un approccio strutturale. Abbiamo utilizzato diverse componenti per la progettazione, per la maggior parte sviluppati seguendo un approccio comportamentale.

Contatore

Il contatore è stato realizzato con un approccio comportamentale e risulta essere di fondamentale importanza nella nostra architettura in quanto ci permette di *indirizzare le tre memorie che costituiscono il sistema*. Il contatore conterà fino a 7 (modulo 8) poiché abbiamo supposto di avere memorie con 8 locazioni.

L'uscita del contatore permetterà all'unità operativa di comunicare all'unità di controllo che tutti i confronti sono stati effettivamente svolti.

```
1 entity counter is
2     Port ( clock : in STD_LOGIC;
3             reset : in STD_LOGIC;
4             enable : in STD_LOGIC;
5             counter : out STD_LOGIC_VECTOR (0 to 2);
6             uscita : out STD_LOGIC
7         );
8 end counter;
9
10 architecture Behavioral of counter is
11 begin
```

```
12
13 process(clock,reset,enable)
14 variable count: integer := 0;
15 begin
16     if(reset = '1') then
17         count := 0;
18         uscita <= '0';
19     end if;
20     if falling_edge(clock) then
21         if(enable = '1') then
22             if(count = 7) then
23                 count := 0;
24                 uscita <= '1';
25             else
26                 count := count +1;
27                 uscita <= '0';
28             end if;
29         end if;
30     end if;
31     counter <= std_logic_vector(to_unsigned(count, 3));
32 end process;
33 end Behavioral;
```

ROM degli input

Il componente ROM-in ha lo scopo di *memorizzare gli ingressi che vogliamo sottoporre alla macchina combinatoria per testarla*. Ogni locazione sarà composta da 4 bit poiché la macchina combinatoria ha 4 ingressi.

```
1 entity ROM_in is
2   port(
3     clock : in std_logic;
4     reset : in std_logic;
5     read  : in std_logic;
6     addr  : in std_logic_vector(0 to 2);
7     y      : out std_logic_vector(0 to 3)
8   );
9 end ROM_in;
```

La ROM leggerà un valore indicizzato dal valore *addr* quando il segnale di *read*, fornito dall'unità di controllo, sarà alto. L'uscita della ROM andrà direttamente in ingresso alla macchina combinatoria, essendo così pronta per essere testata. Inoltre, è fornita di un segnale di *reset*, che ha lo scopo di resettare il valore dell'uscita, ponendola al valore di default *ROM(0)*.

```
1 architecture Behavioral of ROM_in is
2
3   type rom_type is array (0 to 7) of std_logic_vector(0 to 3);
```

```
4 type init_rom is array (0 to 7) of std_logic_vector(0 to 3);
5
6 signal ROM : rom_type;
7 signal value : init_rom := (
8 "0000",  "1001",  "1100",  "1101",  "0011",  "0111",  "1111",  "1010"
9 );
10
11 begin
12 process(clock)
13 begin
14 if rising_edge(clock) then
15     init: for i in 0 to 7 loop
16         ROM(i) <= value(i);
17     end loop init;
18     if(reset = '1') then
19         y <= ROM(0);
20     elsif(READ = '1') then
21         y <= ROM(TO_INTEGER(unsigned(addr)));
22     end if;
23 end if;
24 end process;
25
26 end Behavioral;
```

Macchina combinatoria

Questo componente rappresenta l'unità da testare. E' stata realizzata al livello dataflow e fa *semplici operazioni booleane sugli input*.

```
1 entity m_comb is
2   port(
3     input : in std_logic_vector(0 to 3);
4     output : out std_logic_vector(0 to 2)
5   );
6 end m_comb;
7
8 architecture dataflow of m_comb is
9 begin
10
11   output(0) <= input(0) AND input(1) AND input(2) AND input(3);
12   output(1) <= input(0) OR  input(1) OR  input(2) OR  input(3);
13   output(2) <= input(0) XOR input(1) XOR input(2) XOR input(3);
14
15 end dataflow;
```

Memoria per gli output

Questo componente ha l'unico scopo di *memorizzare i risultati delle operazioni della macchina combinatoria*. La memoria scriverà nella locazione indicizzata da *addr* quando il segnale di *write* è alto, fornito dall'unità di controllo. Il reset permette di inizializzare tutte le locazioni al valore "000". Le 8 locazioni contengono ciascuna 3 bit poiché l'uscita della macchina combinatoria è mappata su 3 bit.

```
1 entity MEM_out is
```

```
2  port(
3      clock : in std_logic;
4      reset : in std_logic;
5      write : in std_logic;
6      addr  : in std_logic_vector(0 to 2);
7      value : in std_logic_vector(0 to 2)
8  );
9 end MEM_out;
10
11 architecture Behavioral of MEM_out is
12
13 type mem_type is array (0 to 7) of std_logic_vector(0 to 2);
14 signal MEM : mem_type;
15
16 begin
17 process(clock)
18 begin
19     if rising_edge(clock) then
20         if(reset = '1') then
21             init: for i in 0 to 7 loop
22                 MEM(i) <= "000";
23             end loop init;
24         end if;
25         if(write = '1') then
26             MEM(to_integer(unsigned(addr))) <= value;
27         end if;
28     end if;
29 
```

```
30 end process;  
31 end Behavioral;
```

ROM dei risultati attesi

Questo componente ha lo scopo di *memorizzare i risultati attesi dalla macchina combinatoria*. L'uscita di questa macchina andrà in ingresso al comparatore, che confronterà tale valore con l'uscita della macchina combinatoria.

```
1 entity ROM_results is  
2   port(  
3     clock : in std_logic;  
4     reset : in std_logic;  
5     read : in std_logic;  
6     addr : in std_logic_vector(0 to 2);  
7     y : out std_logic_vector(0 to 2)  
8   );  
9  
10 end ROM_results;  
11  
12 architecture Behavioral of ROM_results is  
13  
14   type rom_type is array (0 to 7) of std_logic_vector(0 to 2);  
15   type init_rom is array (0 to 7) of std_logic_vector(0 to 2);  
16  
17   signal ROM : rom_type;
```

```
18 signal value : init_rom := (
19   "000", "010", "010", "011", "010", "011", "110", "010"
20 );
21
22 begin
23 process(clock)
24 begin
25   if rising_edge(clock) then
26     init: for i in 0 to 7 loop
27       ROM(i) <= value(i);
28     end loop init;
29   if(reset= '1')
30     y <= ROM(0);
31   elsif(read = '1') then
32     y <= ROM(to_integer(unsigned(addr)));
33   end if;
34 end if;
35 end process;
36 end Behavioral;
```

Comparatore

Il comparatore permette di effettuare il test vero e proprio della macchina. E' dotato di una abilitazione e di un segnale di *load*, che ha lo scopo di *abilitare al confronto i due ingressi di cui il comparatore è fornito.*

```

1  entity comparatore is
2      port(
3          clock : in std_logic;
4          en     : in std_logic;
5          val1   : in std_logic_vector(0 to 2);
6          val2   : in std_logic_vector(0 to 2);
7          load   : in std_logic;
8          y       : out std_logic
9      );
10 end comparatore;
11
12 architecture Behavioral of comparatore is
13
14 signal temp1 : std_logic_vector(0 to 2) := "000";
15 signal temp2 : std_logic_vector(0 to 2) := "000";
16
17 begin
18 process(clock)
19 begin
20     if rising_edge(clock) then
21         if(load = '1') then
22             temp1 <= val1;
23             temp2 <= val2;
24         elsif (en = '1') then
25             if((temp1(0) = temp2(0)) AND (temp1(1) = temp2(1)) AND (
26                 temp1(2) = temp2(2))) then
27                 y <= '1';
28             else

```

```
28         y <= '0';
29
30     end if;
31
32 end if;
33
34 end process;
35
36 end Behavioral;
```

Realizzazione dell'unità operativa

Una volta progettati tutti i componenti, è possibile realizzare l'unità operativa con un approccio strutturale. E' importante notare che *gli ingressi di abilitazione dell'unità operativa corrispondono alle uscite dell'unità di controllo.*

```
1 entity Unita_Operativa is
2
3     port (
4         CLOCK          : in std_logic;
5         RESET          : in std_logic;
6         reset_mem      : in std_logic;
7         read_rom       : in std_logic;
8         read_rom_2     : in std_logic;
9         write_mem      : in std_logic;
10        enable         : in std_logic;
11        enable_counter : in std_logic;
12        load           : in std_logic;
13        fine           : out std_logic;
14        test            : out std_logic
```

```
14 );  
15 end Unità_Operativa;
```

Per quanto riguarda i segnali di uscita, il segnale *test* fornisce l'esito di un confronto (verrà gestito dal comparatore) mentre il segnale di *fine* permette di segnalare quando si sono testati tutti i valori (verrà gestito dal contatore).

```
1 begin  
2   cont : counter  
3  
4     port map(  
5       clock => CLOCK,  
6       reset => RESET,  
7       enable => enable_counter,  
8       counter => count_out_tmp,  
9       uscita => fine  
10    );  
11  
12   rom_input : ROM_in  
13  
14     port map(  
15       clock => CLOCK,  
16       reset => RESET,  
17       read  => read_rom,  
18       addr => count_out_tmp,  
19       y => data_in  
20    );  
21  
22   comb : m_comb
```

```
21 port map(
22     input => data_in,
23     output => data_out
24 );
25
26 mem_output : MEM_out
27 port map(
28     clock => CLOCK,
29     reset => reset_mem,
30     write => write_mem,
31     addr => count_out_tmp,
32     value => data_out
33 );
34
35 rom_expected_results : ROM_results
36 port map(
37     clock => CLOCK,
38     reset => RESET,
39     read => read_rom_2,
40     addr => count_out_tmp,
41     y => data_out_exp
42 );
43
44 comp : comparatore
45 port map(
46     clock => CLOCK,
47     en => enable,
48     val1 => data_out,
```

```

49      val2 => data_out_exp,
50
51      load => load,
52
53      y => test
54
55  );
56
57
58
59
60
61
62
63
64 end Structural;

```

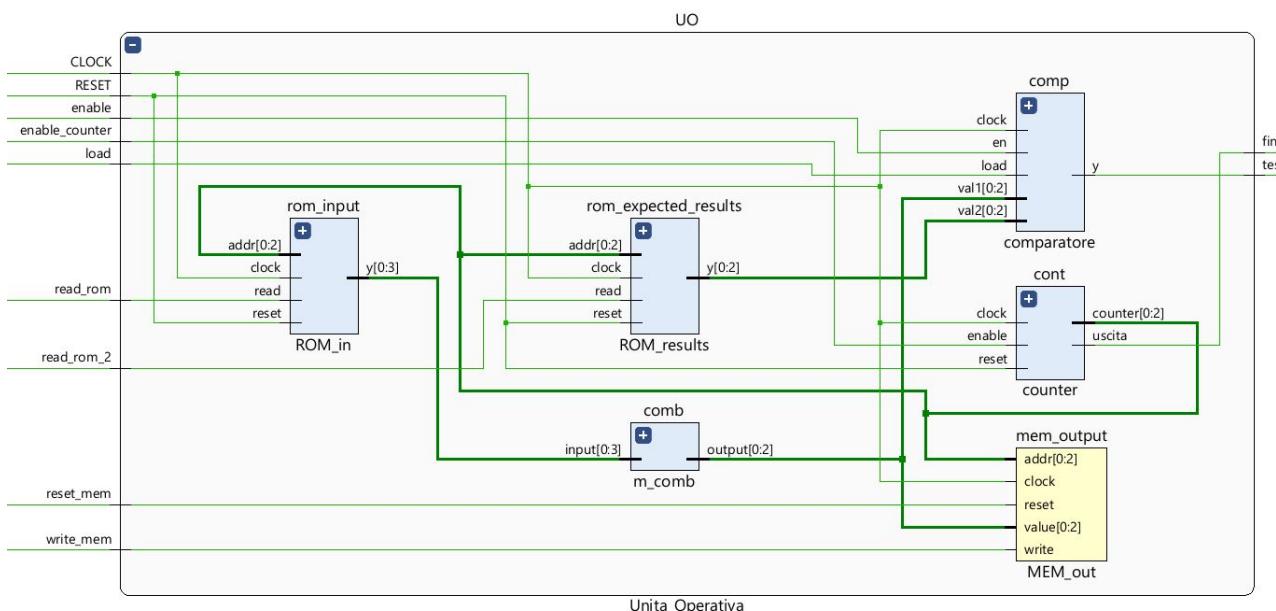


Figura 2.17: Schematic unità operativa

2.8.2 Unità di controllo

Dopo aver progettato la parte operativa, possiamo passare alla progettazione dell'unità di controllo. L'unità di controllo è stata realizzata mediante un'automa a stati finiti, quindi in logica cablata, *seguendo un approccio comportamentale con un process*.

La macchina può trovarsi in 7 stati diversi: *idle, lettura-valori, scrittura-*

ris, caricamento-comp, verifica, incrementa e fine-test.

La macchina parte dallo stato *IDLE* e rimarrà in questo stato fin quando l'unità di controllo non avrà in ingresso il segnale *START* pari a 1. In quel caso, la macchina andrà nello stato *lettura-valori*, dove verranno abilitati i segnali di lettura delle due ROM. Successivamente, la macchina va nello stato *scrittura-ris*, dove verrà scritto nella memoria il risultato dell'elaborazione della macchina combinatoria. Poi la macchina va nello stato *caricamento-comp* nel quale viene abilitato il segnale di *load* e dopo va nello stato di *verifica*, dove viene abilitato il comparatore che effettua il confronto. La macchina poi passa nello stato di *incrementa*, dove viene abilitato il contatore che passa alla prossima locazione di memoria e infine nello stato *fine-test*. Nell'ultimo stato, in funzione del valore di conteggio, la macchina continuerà a leggere i valori di test successivi dalla ROM oppure tornerà nello stato *IDLE*.

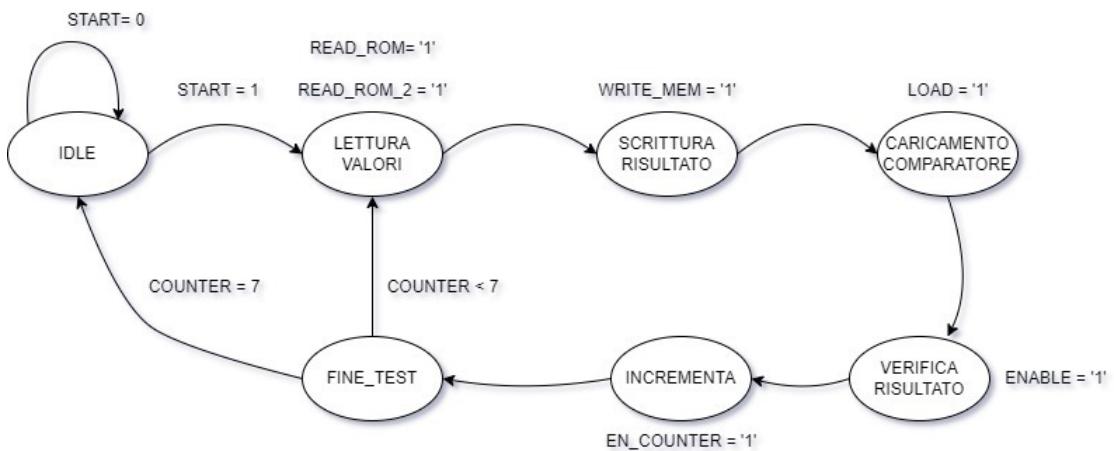


Figura 2.18: Automa rete di controllo

```
1 entity Unita_Controllo is
2     port(
3         start          : in std_logic;
4         clock          : in std_logic;
5         reset          : in std_logic;
6         reset_mem      : out std_logic;
7         read_rom       : out std_logic;
8         read_rom_2     : out std_logic;
9         write_mem      : out std_logic;
10        enable         : out std_logic;
11        enable_counter : out std_logic;
12        load           : out std_logic;
13        fine           : in std_logic
14    );
15 end Unita_Controllo;
16
17 architecture Behavioral of Unita_Controllo is
18
19 type stato is (idle,lettura_valori,scrittura_ris, caricamento_comp,
20                 verifica_ris, incrementa ,fine_test);
21
22 signal stato_corrente : stato := idle;
23
24 begin
25
26 process(clock, reset)
27 begin
28     if(reset = '1') then
29         stato_corrente <= idle;
```

```
28  end if;

29  if(rising_edge(clock) ) then
30
31      case stato_corrente is
32
33          when idle =>
34
35              reset_mem <= '1';
36
37              read_rom  <= '0';
38
39              read_rom_2 <= '0';
40
41              write_mem <= '0';
42
43              enable <= '0';
44
45              enable_counter <= '0';
46
47              load <= '0';
48
49              if(start = '1') then
50
51                  stato_corrente <= lettura_valori;
52
53              else
54
55                  stato_corrente <= idle;
56
57              end if;
58
59          when lettura_valori =>
60
61              reset_mem <= '0';
62
63              read_rom  <= '1';
64
65              read_rom_2 <= '1';
66
67              write_mem <= '0';
68
69              enable <= '0';
70
71              enable_counter <= '0';
72
73              load <= '0';
74
75              stato_corrente <= scrittura_ris;
76
77
78          when scrittura_ris =>
79
80              reset_mem <= '0';
81
```

```
56      read_rom <= '0';
57      read_rom_2 <= '0';
58      write_mem <= '1';
59      enable <= '0';
60      enable_counter <= '0';
61      load <= '0';
62      stato_corrente <= caricamento_comp;
63
64      when caricamento_comp =>
65          reset_mem <= '0';
66          read_rom <= '0';
67          read_rom_2 <= '0';
68          write_mem <= '0';
69          enable <= '0';
70          enable_counter <= '0';
71          load <= '1';
72          stato_corrente <= verifica_ris;
73
74      when verifica_ris =>
75          reset_mem <= '0';
76          read_rom <= '0';
77          read_rom_2 <= '0';
78          write_mem <= '0';
79          enable <= '1';
80          enable_counter <= '0';
81          load <= '0';
82          stato_corrente <= incrementa;
83
```

```
84      when incrementa =>
85          reset_mem <= '0';
86          read_rom <= '0';
87          read_rom_2 <= '0';
88          write_mem <= '0';
89          enable <= '0';
90          enable_counter <= '1';
91          load <= '0';
92          stato_corrente <= fine_test;
93
94      when fine_test =>
95          reset_mem <= '0';
96          read_rom <= '0';
97          read_rom_2 <= '0';
98          write_mem <= '0';
99          enable <= '0';
100         enable_counter <= '0';
101         load <= '0';
102         if(fine = '1') then
103             stato_corrente <= idle;
104         else
105             stato_corrente <= lettura_valori;
106         end if;
107     end case;
108 end if;
109
110 end process;
```

2.8.3 Sistema complessivo

Il sistema di testing complessivo avrà un segnale di ingresso *start*, che permette alla rete di controllo di abbandonare lo stato *IDLE*, un segnale di *RESET* che verrà associato al reset del contatore e un *reset_mem* che funzionerà come reset per le memorie. Come uscita abbiamo un segnale *test*, che rappresenta l'esito del confronto i-esimo.

```
1 entity Sistema_Testing is
2     port(
3         clock : in std_logic;
4         reset : in std_logic;
5         reset_mem : in std_logic;
6         start : in std_logic;
7         test : out std_logic
8     );
9 end Sistema_Testing;
```

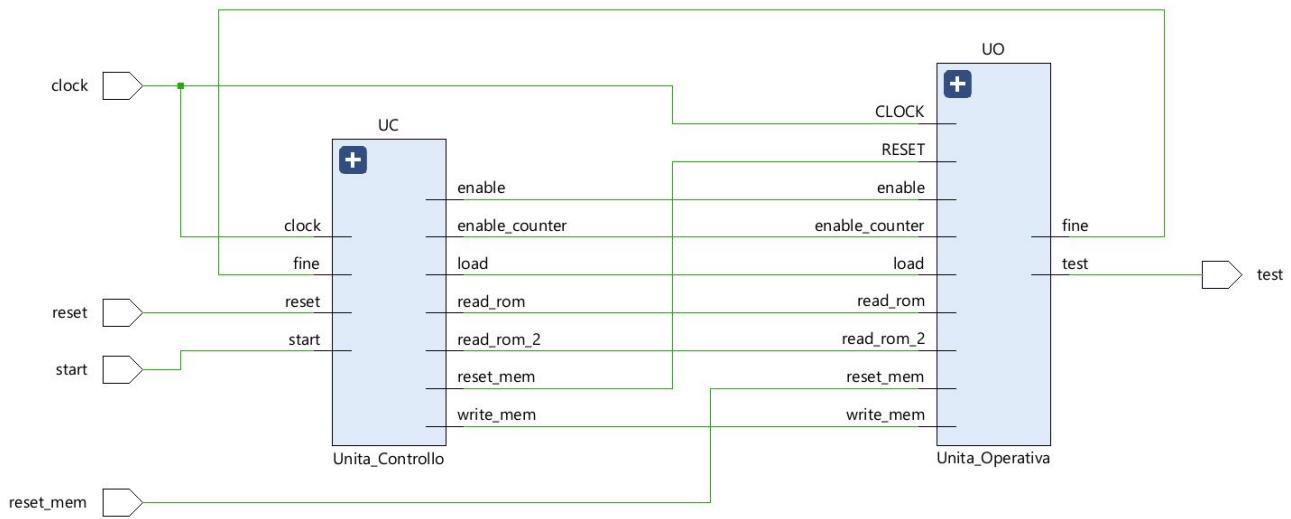


Figura 2.19: Schematic sistema di testing

2.8.4 Simulazione del sistema

Possiamo notare dalla simulazione che le uscite della macchina combinatoria sono corrette.

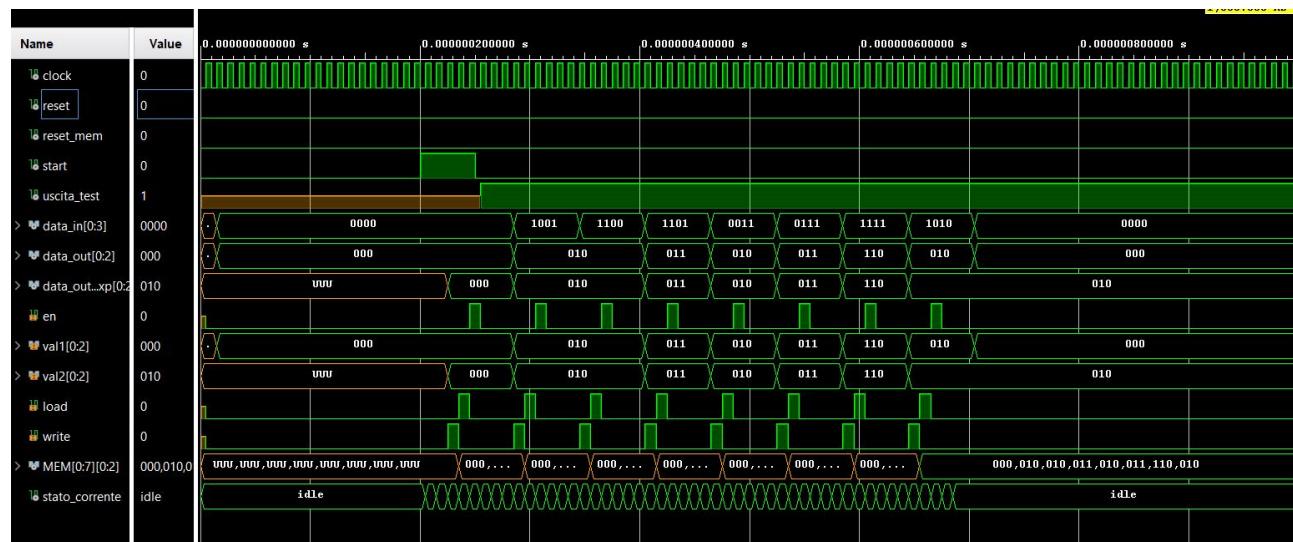


Figura 2.20: Simulazione sistema complessivo

Dalla figura in basso possiamo notare invece la corretta evoluzione degli stati della macchina in accordo con lo schema dell'automa precedentemente introdotto.



Figura 2.21: Evoluzione degli stati dell’unità di controllo

2.9 Esercizio 6.2 – Sistema di testing on board

Per la realizzazione di questo progetto è stato necessario estendere il comportamento del sistema di testing, aggiungendo due bottoni per permettere il reset della macchina e la lettura dei valori dalla ROM e un led per visualizzare l'uscita del comparatore.

```
1 entity Sistema_Testing is
2   port(
3     clock : in std_logic;
4     reset : in std_logic;
5     reset_mem : in std_logic;
6     start_sw : in std_logic;
7     read_btn : in std_logic;
8     test : out std_logic
9   );
10 end Sistema_Testing;
```

Alla entity del sistema originale, abbiamo aggiunto due segnali, *read_btn*, che permetterà di acquisire l'input tramite bottone e *start_sw*, che permetterà di avviare la macchina tramite uno switch. Successivamente, è stato necessario modificare anche l'unità di controllo, in quanto, una volta lasciato lo stato *IDLE*, non dovrà subito passare a leggere i valori dalla memoria, ma lo farà soltanto dopo la pressione del pulsante di *read*. Infatti, la macchina rimarrà ferma nel-

lo stato di *lettura_valori* fin quando non riceverà il segnale di lettura dall'esterno.

```
1 signal last_read : std_logic;
2 signal last_start : std_logic;
3 ...
4 case stato_corrente is
5     when idle =>
6         reset_mem      <= '1';
7         read_rom       <= '0';
8         read_rom_2    <= '0';
9         write_mem     <= '0';
10        enable        <= '0';
11        enable_counter <= '0';
12        load          <= '0';
13        if(start = '1' AND last_start = '0') then
14            last_start <= '1';
15            stato_corrente <= lettura_valori;
16        else
17            last_start <= '0';
18            stato_corrente <= idle;
19        end if;
20        when lettura_valori =>
21            reset_mem      <= '0';
22            write_mem     <= '0';
23            enable        <= '0';
24            enable_counter <= '0';
25            load          <= '0';
```

```
26      if( read_en = '1' AND last_read = '0') then
27          last_read <= '1';
28          read_rom           <= '1';
29          read_rom_2         <= '1';
30          stato_corrente    <= scrittura_ris;
31      else
32          last_read <= '0';
33          stato_corrente <= lettura_valori;
34      end if;
```

Per la gestione dei due buttoni, sono stati utilizzati dei componenti debouncer, precedentemente sviluppati, per eliminare le oscillazioni. Infine, il segnale *test*, in uscita dal sistema di testing, che è associato all'uscita del comparatore, verrà mostrato sul display. Per testare l'uscita, abbiamo volontariamente cambiato il valore di una locazione della ROM dei risultati attesi affinché il comparatore desse una uscita 0, facendo spegnere il led.

Capitolo 3

Comunicazione con handshaking

3.1 Traccia

3.1.1 Esercizio 7

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.

3.2 Esercizio 7 – Sistema di handshaking

3.2.1 Introduzione

Nei sistemi che richiedono una comunicazione tra diverse entità è necessario l'utilizzo di un **protocollo**, cioè di una serie di regole che permettono *il corretto scambio di informazioni tra le due entità*.

Possiamo classificare i protocolli in tre categorie:

- Protocolli sincroni;
- Protocolli asincroni;
- Protocolli semisincroni.

Nei protocolli **sincroni** le unità lavorano con *lo stesso riferimento temporale*, noto e condiviso.

I protocolli **asincroni** si basano sul concetto di evento. Le entità comunicano alle ricezione di eventi e *finché, non si verificano eventi, non trasmettono*.

I protocolli **semisincroni** funzionano allo stesso modo di quelli sincroni prevedendo però per lo slave della comunicazione la possibilità di *ritardare il messaggio tramite un apposito flag*.

Per la realizzazione del progetto abbiamo individuato, per ogni entità coinvolta, una parte di controllo e una parte operativa. Inoltre, abbiamo supposto di avere memorie ROM di 8 locazioni a 4 bit e una memoria per salvare i risultati delle somme con 8 locazioni da 5 bit.

3.2.2 Unità operativa - entità A

L'unità operativa è stata realizzata mediante un approccio strutturale. L'architettura prevede un contatore modulo 8 per indicizzare una memoria ROM da 8 locazioni.

```

1 entity Unita_Operativa_A is
2   Port (
3     CLK      : in std_logic;
4     RST      : in std_logic;
5     EN_COUNT : in std_logic; -- enable del contatore
6     READ     : in std_logic; -- read per la memoria
7     FINE     : out std_logic; -- segnale di fine; locazioni
8                   terminate
9     DATA_A    : out std_logic_vector(0 to 3)
10    );
11 end Unita_Operativa_A;
```

L'unità operativa ha come ingressi un segnale di abilitazione del contatore, EN_COUNT, e di lettura, READ. Quest'ultimi sono pilotati dall'unità di controllo. In uscita, abbiamo un segnale FINISH che permette di *notificare l'unità di controllo della fine delle locazioni da leggere*. Infine, in uscita abbiamo ancora un segnale DATA_A, che rappresenta il valore letto dalla memoria ROM di A, il quale andrà in ingresso all'unità operativa di B. Tale valore costituirà come primo operando del sommatore. Una scelta progettuale equivalente sarebbe potuta essere quella di *inviare il segnale DATA_A all'unità di controllo di A*, delegando a quest'ultima l'invio dei dati all'unità di controllo di B e, successivamente, all'unità operativa di B.

```

1 architecture Structural of Unita_Operativa_A is
2
3   signal count_addr_out : std_logic_vector(0 to 2) := (others => '0');
4   signal rom_a_out      : std_logic_vector(0 to 3) := (others => '0');
5
6 component ROM_in is
7   port (
8     clock      : in  std_logic;
9     reset      : in  std_logic;
10    read       : in  std_logic;
11    addr       : in  std_logic_vector(0 to 2);
12    value_out  : out std_logic_vector(0 to 3)
13   );
14 end component ROM_in;
```

```

15
16 component contatore is
17   Port (
18     clock    : in std_logic;
19     reset    : in std_logic;
20     enable   : in std_logic;
21     counter  : out std_logic_vector(0 to 2);
22     uscita   : out std_logic
23   );
24 end component contatore;
25
26 begin
27   Address_counter : contatore
28     Port map (
29       clock    => CLK,
30       reset    => RST,
31       enable   => EN_COUNT,
32       counter  => count_addr_out,
33       uscita   => FINE
34     );
35
36   ROM_A : ROM_in
37     Port map (
38       clock      => CLK,
39       reset      => RST,
40       read       => READ,
41       addr       => count_addr_out,
42       value_out  => DATA_A

```

```

43 );
44
45 end Structural;

```

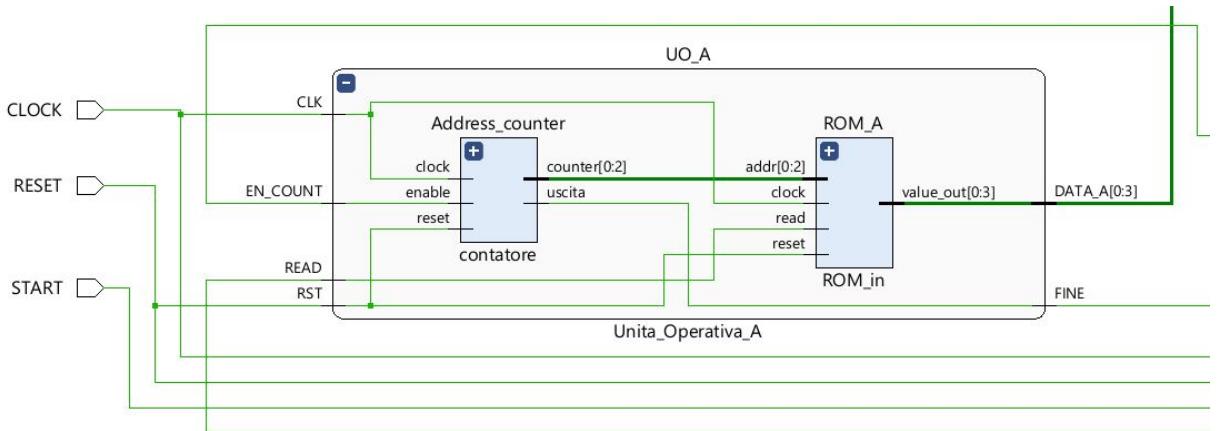


Figura 3.1: Schematic unità operativa entità A

3.2.3 Unità di controllo – entità A

L’unità di controllo dell’entità A è stata realizzata in logica cablata, mediante un automa a stati finiti.

L’automa rappresentante la rete di controllo di A evolve attraverso diversi stati: *idle*, *read*, *wait*, *increment* e *end_check*. L’automa parte dallo stato *idle* e vi rimane fin quando un segnale *start*, proveniente dall’esterno, non diventa alto. Una volta diventato alto, si va nello stato di *read*, dove viene inviato il segnale *req* all’unità di controllo di B e abilitato un segnale per leggere dalla memoria il valore che costituirà il primo operando del sommatore. Successivamente, la rete va in uno stato di *wait*, in attesa di un *ack* da parte dell’entità B. Una

volta ricevuto l'ack, la rete è pronta per leggere un altro valore dalla memoria e quindi alza l'abilitazione del contatore. Una volta fatto ciò, se si è arrivati a 7, quindi tutte le locazioni sono state lette, la rete torna nello stato *idle*, altrimenti leggerà un nuovo valore.

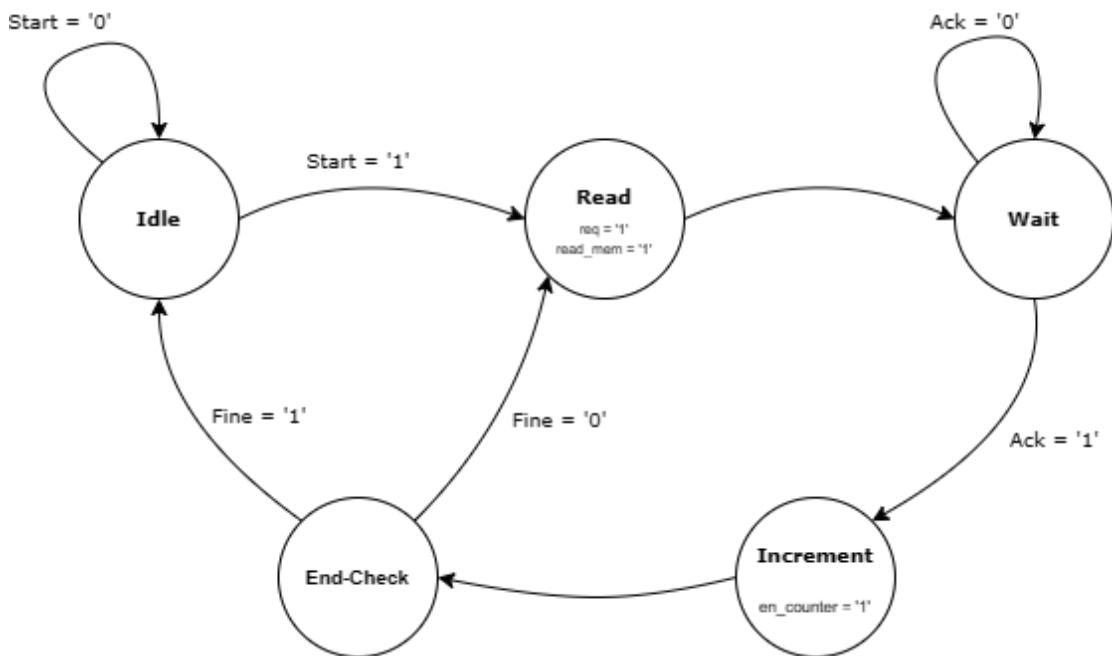


Figura 3.2: Automa Unità di controllo – entità A

Mostriamo l'entity dell'unità di controllo A, la quale fornisce all'unità operativa diversi segnali di controllo.

```

1 entity Unita_Controllo_A is
2   Port (
3     clock      : in std_logic;
4     reset      : in std_logic;
5     start      : in std_logic;
  
```

```

6      fine      :  in  std_logic;
7      ack       :  in  std_logic;
8      req       :  out std_logic; -- request to send
9      read_mem  :  out std_logic;
10     en_count   :  out std_logic
11   );
12 end Unità_Controllo_A;

```

Il segnale `req` è un segnale che dà il via alla comunicazione, tipico dei protocolli basati su handshake, inoltre abbiamo un segnale che abilita la lettura dalla memoria, `read_mem`, e un segnale per passare alla locazione successiva, incrementando il valore del contatore.

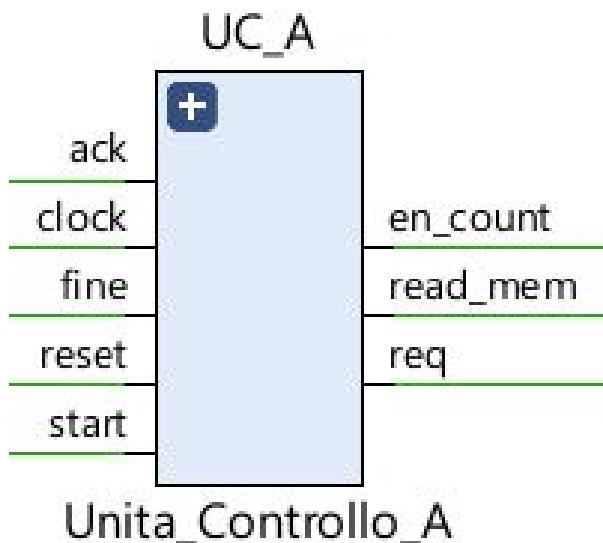


Figura 3.3: Schematic unità di controllo entità A

```

1 architecture Behavioral of Unità_Controllo_A is
2
3 type stato is (idle, read, wait_1, increment, end_check);
4 signal current : stato := idle;

```

```

5
6 begin
7 fsm_A : process(clock)
8 begin
9 if reset = '1' then
10     current <= idle;
11 end if;
12 if rising_edge(clock) then
13     case current is
14         when idle =>
15             req <= '0';
16             read_mem <= '0';
17             en_count <= '0';
18             if start = '1' then
19                 current <= read;
20             else
21                 current <= idle;
22             end if;
23         when read =>
24             read_mem    <= '1';
25             req        <= '1';
26             en_count   <= '0';
27             current <= wait_1;
28         when wait_1 =>
29             if ack = '1' then
30                 current <= increment;
31             else
32                 current <= wait_1;

```

```

33         end if;

34     when increment =>

35         en_count      <= '1';
36
37         req           <= '0';
38
39         read_mem      <= '0';
40
41         current       <= end_check;

42     when end_check =>

43         en_count      <= '0';
44
45         req           <= '0';
46
47         read_mem      <= '0';
48
49         if fine = '1' then
50
51             current       <= idle;
52
53         else
54
55             current       <= read;
56
57         end if;

58     end case;

59 end if;

60
61 end process fsm_A;

62 end Behavioral;

```

3.2.4 Unità operativa – entità B

L’unità operativa dell’entità B possiede la stessa struttura di quella dell’entità A, con l’aggiunta di un sommatore e di una memoria per contenere i risultati del sommatore.

```

1 entity Unita_Operativa_B is
2     Port (
3         CLK          : in  std_logic;
4         RST          : in  std_logic;
5         EN_COUNT    : in  std_logic; — enable del contatore
6         EN_ADD       : in  std_logic; — enable del sommatore
7         WRITE        : in  std_logic; — write per la memoria
8         READ         : in  std_logic; — read per la memoria
9         DATA_A        : in  std_logic_vector(0 to 3); — operando
10            — proveniente da A
11         FINE        : out std_logic — segnale di fine; locazioni
12            — terminate
13     );
14
15 end Unita_Operativa_B;

```

Il sommatore è stato un nuovo componente da implementare. Per l'esercizio, è stato implementato in modo comportamentale. Poiché i due operandi sono entrambi di 4 bit, il risultato sarà al massimo di 5 bit, pertanto la dimensione inserita basterà a contenere il risultato. Inoltre, è dotato di un'abilitazione e di un segnale di reset, che ha lo scopo di portare l'uscita a 0.

```

1 entity adder is
2     Port (
3         clock      : in  std_logic;
4         reset       : in  std_logic;
5         enable     : in  std_logic;

```

```

6      op1      :  in  std_logic_vector(0 to 3);
7      op2      :  in  std_logic_vector(0 to 3);
8      sum      :  out std_logic_vector(0 to 4)
9      );
10 end adder;
11
12 architecture Behavioral of adder is
13 begin
14 process(clock, reset)
15 variable a : integer := 0;
16 variable b : integer := 0;
17 variable z : integer := 0;
18 begin
19 if rising_edge(clock) then
20     if reset = '1' then
21         sum <= (others => '0');
22     end if;
23     if enable = '1' then
24         a := to_integer(unsigned(op1));
25         b := to_integer(unsigned(op2));
26         z := a+b;
27         sum <= std_logic_vector(to_unsigned(z, 5));
28     end if;
29 end if;
30
31 end process;
32 end Behavioral;

```

Di seguito, è riportato il continuo del codice dell'unità operativa,

mostrando l'architettura ed il port mapping, con il relativo schematic.

```

1 architecture Behavioral of Unita_Operativa_B is
2
3 signal count_addr_out : std_logic_vector(0 to 2) := (others => '0');
4 signal DATA_B          : std_logic_vector(0 to 3) := (others => '0');
5 signal sum_out         : std_logic_vector(0 to 4) := (others => '0');
6
7 component ROM_in is
8
9     port(
10         clock      : in  std_logic;
11         reset      : in  std_logic;
12         read       : in  std_logic;
13         addr       : in  std_logic_vector(0 to 2);
14         value_out  : out std_logic_vector(0 to 3)
15     );
16
17 end component ROM_in;
18
19 component memory is
20
21     Port(
22         clock      : in  std_logic;
23         reset      : in  std_logic;
24         write      : in  std_logic;
25         addr       : in  std_logic_vector(0 to 2);
26         value_in   : in  std_logic_vector(0 to 4)
27     );
28
29 end component memory;

```

```

27 component contatore is
28     Port(
29         clock : in std_logic;
30         reset : in std_logic;
31         enable : in std_logic;
32         counter : out std_logic_vector(0 to 2);
33         uscita : out std_logic
34     );
35 end component contatore;

36
37
38 component adder is
39     Port(
40         clock : in std_logic;
41         reset : in std_logic;
42         enable : in std_logic;
43         op1 : in std_logic_vector(0 to 3);
44         op2 : in std_logic_vector(0 to 3);
45         sum : out std_logic_vector(0 to 4)
46     );
47 end component adder;

48
49 begin
50     Address_counter : contatore
51         Port map(
52             clock => CLK,
53             reset => RST,
54             enable => EN_COUNT,

```

```

55     counter => count_addr_out,
56
57     uscita => FINE
58
59 Adder_B : adder
60
61     Port map(
62         clock => CLK,
63         reset => RST,
64         enable => EN_ADD,
65         op1 => DATA_A,
66         op2 => DATA_B,
67         sum => sum_out
68     );
69
70 ROM_B: ROM_in
71
72     Port map(
73         clock => CLK,
74         reset => RST,
75         read => READ,
76         addr => count_addr_out,
77         value_out => DATA_B
78     );
79
80 MEM_B : memory
81
82     Port map(
83         clock => CLK,
84         reset => RST,
85         write => WRITE,

```

```

83     addr      => count_addr_out,
84
85     value_in   => sum_out
86 );
87 end Behavioral;

```

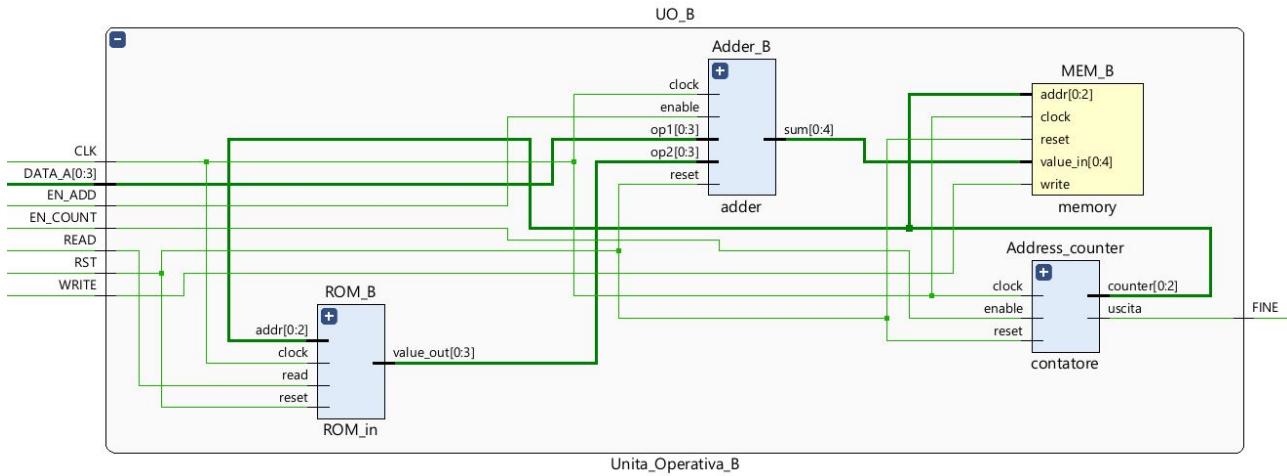


Figura 3.4: Schematic unità operativa entità B

Notiamo come l'unità operativa, in uscita, possiede un segnale `fine`, gestito dal contatore il cui scopo è quello di scandire la ricezione dei dati provenienti da A. Quando il segnale di fine sarà alto, questo segnale andrà in ingresso all'unità di controllo di B il quale, a sua volta, inoltrerà tale segnale in ingresso all'unità di controllo di A dove, una volta raggiunto il conteggio massimo, smetterà di trasmettere.

3.2.5 Unità di controllo – entità B

Anche in questo caso abbiamo usato un automa per rappresentare la rete di controllo.

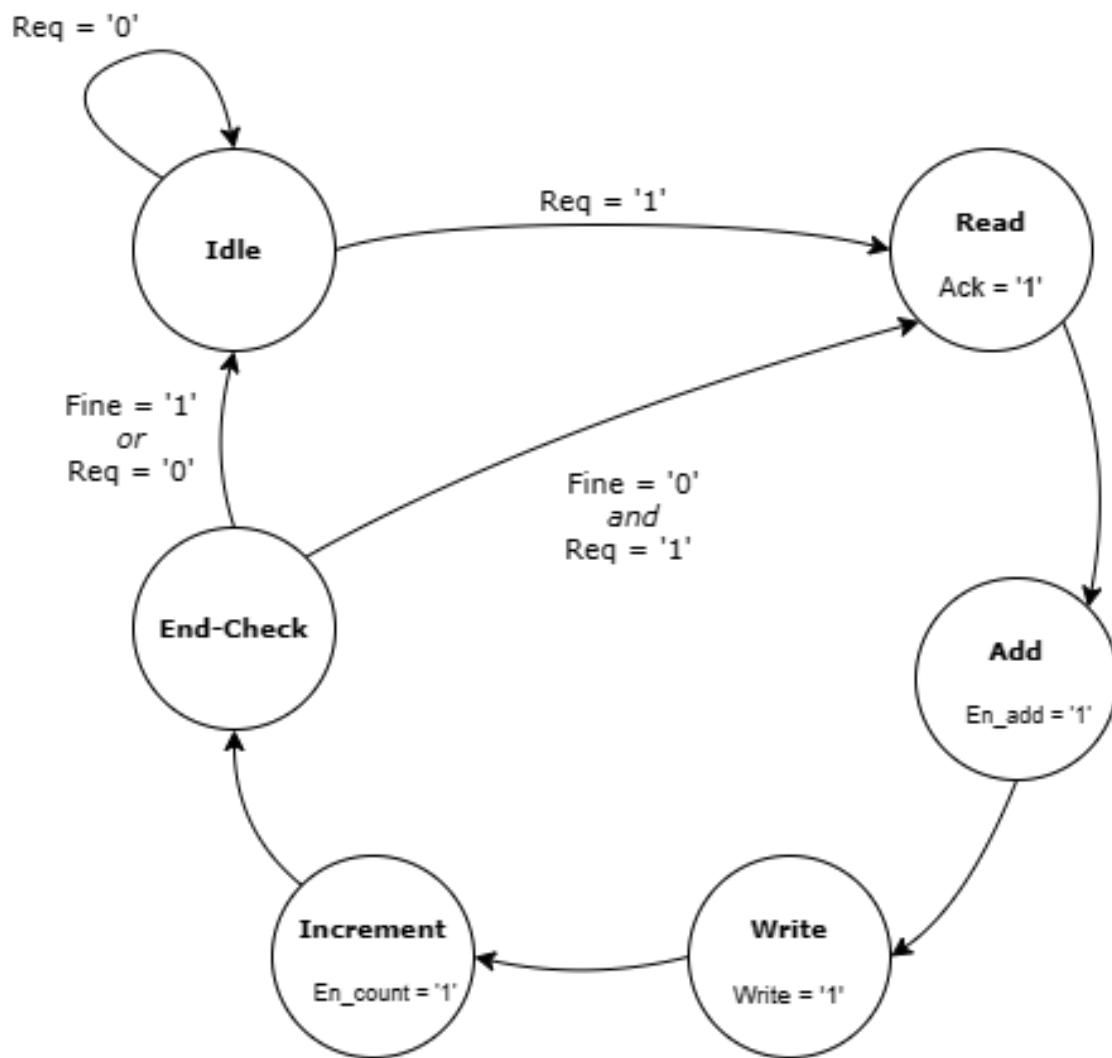


Figura 3.5: Automa Unita di Controllo B

L'automa rappresentante la rete di controllo di B evolve attraverso diversi stati: *idle*, *read*, *add*, *write*, *increment*, *end_check*.

L'automa parte dallo stato *idle*, e vi rimane fin quando il segnale di *req*, che permette di capire quando A sta trasmettendo un dato, non si alza. Una volta alzato, l'automa passa nello stato di *read*, in cui legge il valore della ROM da utilizzare come secondo operando del sommatore e invia l'*ack* per notificare l'unità di controllo di A dell'avvenuta ricezione del messaggio. Successivamente, nello stato *add* si abilita il sommatore alla somma dei due operandi in input e nello stato *write* si scrive il risultato della somma nella memoria. In *increment* si abilita il contatore per passare al prossimo dato da ricevere e indirizzare anche una nuova locazione. Infine, nello stato *end_check*, se sono finite le trasmissioni oppure il segnale di *req* è basso (non ci sono nuovi dati in ingresso) allora l'entità B andrà in *idle*, altrimenti se il segnale di *fine* è basso (non sono ancora finite le trasmissioni) e il segnale di *req* è alto (è disponibile un nuovo dato in input da parte dell'unità operativa di A) allora torna nello stato di *read*.

L'unità di controllo di B, come quella di A, è stata realizzata con un process (quindi a livello comportamentale), in logica cablata.

```

1 entity Unità_Controllo_B is
2   Port(
3     clock      :  in  std_logic;
4     reset      :  in  std_logic;
```

```
5      req          :  in  std_logic;  -- request to send
6      fine_in      :  in  std_logic;
7      fine_out      :  out std_logic;
8      ack           :  out std_logic;
9      write_mem    :  out std_logic;
10     read_mem     :  out std_logic;
11     en_count     :  out std_logic;
12     en_add        :  out std_logic
13   );
14 end Unita_Controllo_B;
```

Dalla entity notiamo in ingresso il segnale `req` proveniente dall'unità i controllo di A, utilizzato per realizzare l'handshaking. Il segnale `fine_in` è quello ricevuto dall'unità operativa, mentre `fine_out` quello da inviare all'unità di controllo di A per terminare la trasmissione. Successivamente, abbiamo un segnale di `ack` in uscita, per notificare A dell'avvenuta ricezione del dato e vari segnali di abilitazione per la memoria, il contatore e l'adder.

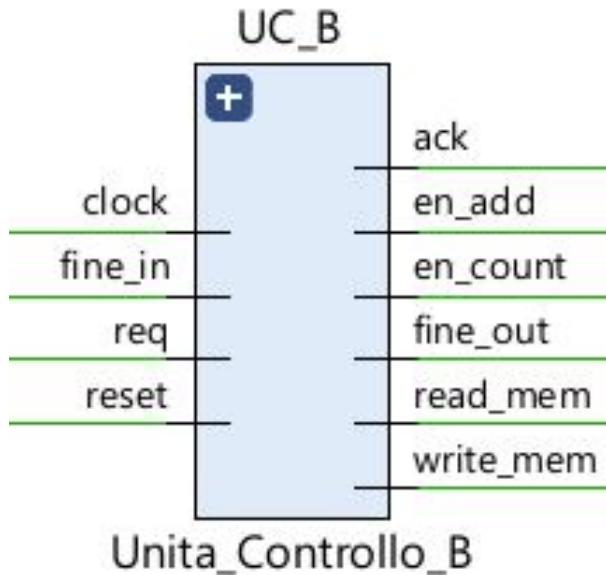


Figura 3.6: Schematic unità di controllo entità B

```

1 architecture Behavioral of Unita_Controllo_B is
2
3 type stato is (idle, read, add, write, increment, end_check);
4 signal current : stato := idle;
5
6 begin
7 fsm_B : process(clock)
8 begin
9 if reset = '1' then
10     current <= idle;
11 end if;
12 if rising_edge(clock) then
13     case current is
14         when idle =>
15             ack      <= '0';
16             write_mem <= '0';

```

```

17      read_mem    <= '0';
18      en_count   <= '0';
19      en_add     <= '0';
20      fine_out   <= '0';

21      if req = '1' then
22          current <= read;
23      else
24          current <= idle;
25      end if;

26

27      when read =>
28          ack         <= '1';
29          read_mem   <= '1';
30          write_mem  <= '0';
31          en_count   <= '0';
32          en_add     <= '0';
33          fine_out   <= '0';
34          current    <= add;

35

36      when add =>
37          ack         <= '0';
38          en_add     <= '1';
39          read_mem   <= '0';
40          write_mem  <= '0';
41          en_count   <= '0';
42          fine_out   <= '0';
43          current    <= write;

44

```

```

45      when write =>
46          ack          <= '0';
47          read_mem    <= '0';
48          write_mem   <= '1';
49          en_count    <= '0';
50          en_add       <= '0';
51          fine_out    <= '0';
52          current     <= increment;
53
54
55      when increment =>
56          ack          <= '0';
57          write_mem    <= '0';
58          read_mem    <= '0';
59          en_count    <= '1';
60          en_add       <= '0';
61          fine_out    <= '0';
62          current     <= end_check;
63
64      when end_check =>
65          ack          <= '0';
66          write_mem    <= '0';
67          read_mem    <= '0';
68          en_count    <= '0';
69          en_add       <= '0';
70
71          if fine_in = '1' OR req = '0' then
72              fine_out <= '1';
73
74              current <= idle;
75
76          elsif (fine_in = '0' AND req = '1') then

```

```
73          fine_out <= '0';
74          current <= read;
75      end if;
76  end case;
77 end if;
78
79 end process fsm_B;
80 end Behavioral;
```

3.2.6 Sistema complessivo

Infine, è riportato lo schematico del sistema di handshake complessivo e il testbench per verificarne il corretto funzionamento. Il sistema complessivo ha solo 3 ingressi, il clock, il reset e il segnale di start, che andrà in ingresso alla rete di controllo dell'entità A e che permette sostanzialmente al sistema di iniziare ad evolvere.

Per semplicità, abbiamo usato per entrambe le ROM gli stessi valori da sommare.

CAPITOLO 3. COMUNICAZIONE CON HANDSHAKING

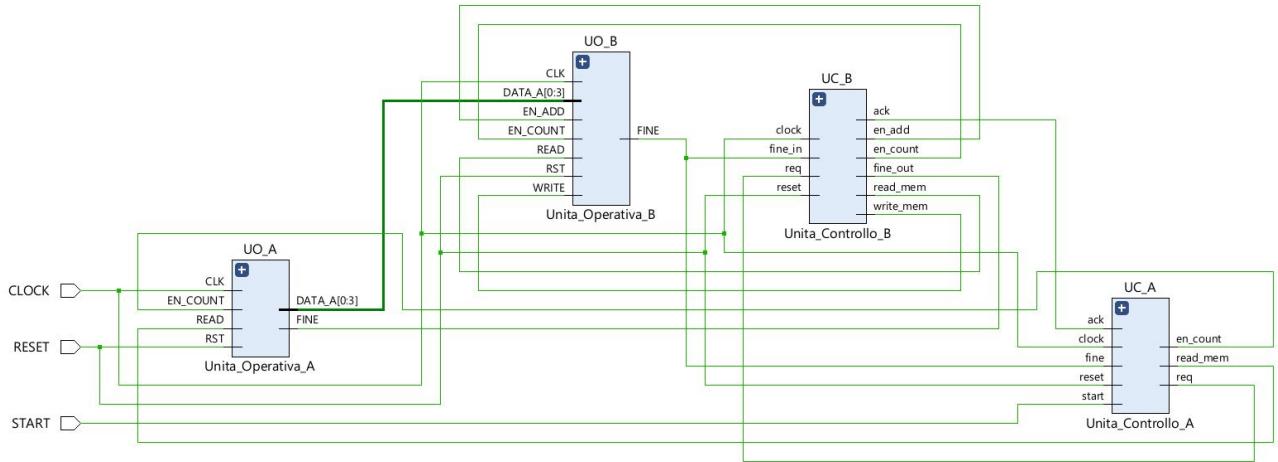


Figura 3.7: Schematic sistema complessivo

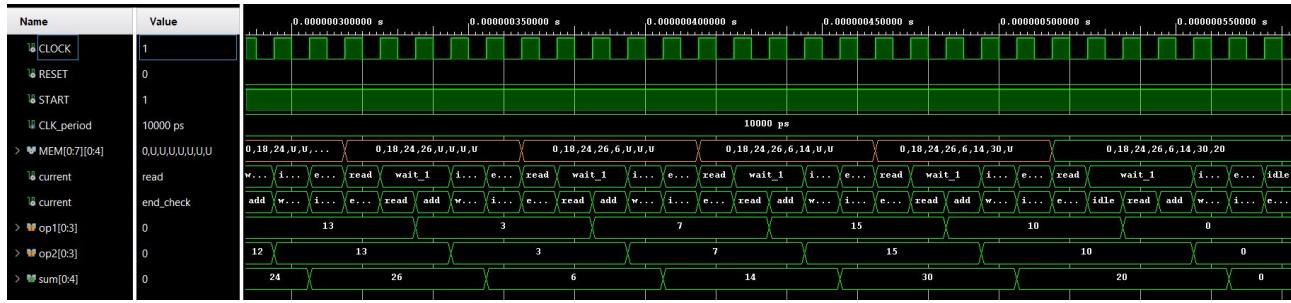


Figura 3.8: Testbench del sistema complessivo

Capitolo 4

Processore

4.1 Traccia

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

4.2 Esercizio 8 – Processore

4.2.1 Introduzione

Il MIC-1 è un processore che utilizza un modello a stack, e quindi, a differenza dell’architettura a registri generali, supporta esclusivamente istruzioni provenienti dallo stack. In questo modo, la lunghezza delle istruzioni tende ad essere inferiore rispetto a quelle del modello a registri generali, in quanto in molte istruzioni gli operandi sono impliciti. Dall’altro lato invece osserviamo che, con il modello a stack, ci sono maggiori inefficienze causate dai continui accessi in memoria.

Il MIC-1 è un processore la cui parte di controllo è stata sviluppata in logica microprogrammata.

4.2.2 Unità operativa

L’unità operativa del MIC-1 comprende l’ALU, i suoi ingressi e le sue uscite. I registri hanno una dimensione di 32 bit e non sono accessibili al programmatore, ma solo al microprogramma.

Il datapath di questo processore è costituito da due bus, il bus C che permette la scrittura all’interno dei registri, e il bus B, che permette invece la lettura dai registri. Il bus B è collegato al secondo ingresso dell’ALU mentre il bus C è collegato all’uscita dell’ALU.

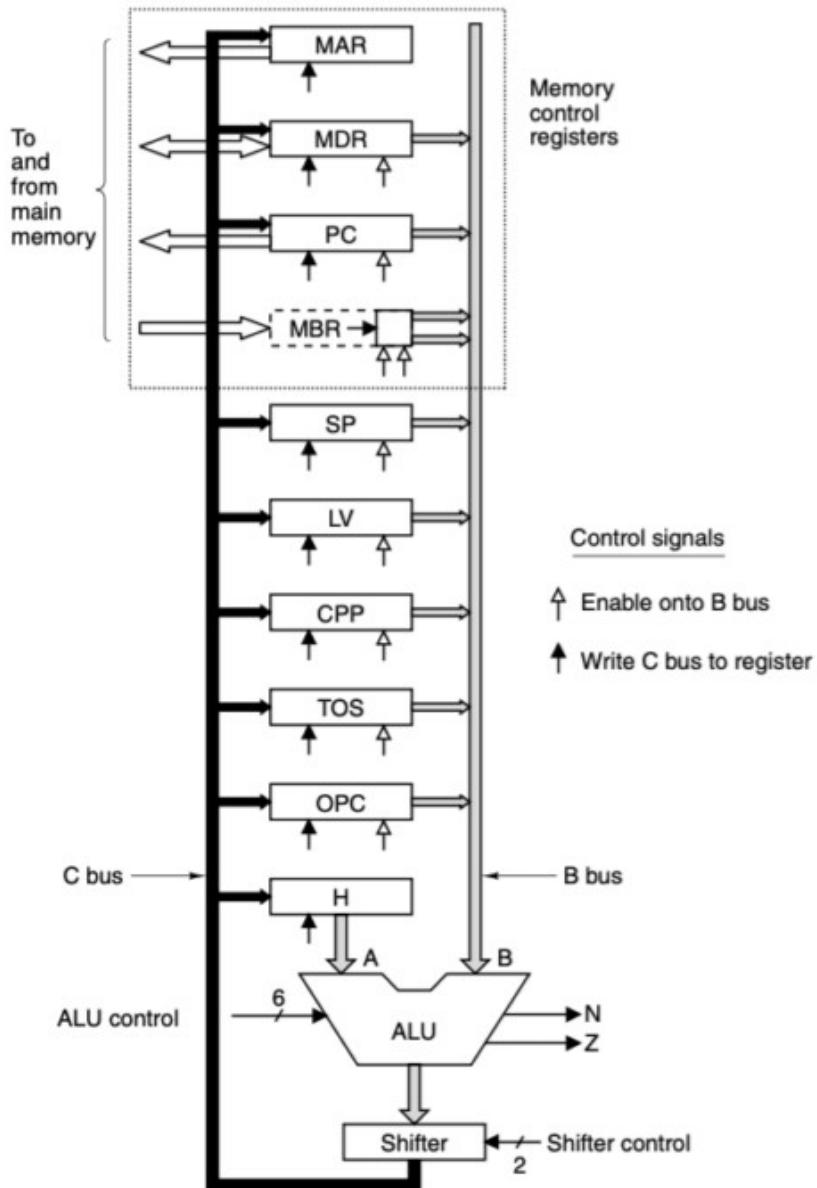


Figura 4.1: Unità operativa del MIC-1

I registri, non essendo generali, assolvono a funzioni specifiche. In particolare abbiamo:

- MAR (Memory Address Register): contiene l'indirizzo di memoria della locazione da cui leggere o scrivere;

- MDR (Memory Data Register): contiene il dato letto dalla memoria o da scrivere in memoria;
- PC (Program Counter): contiene l'indirizzo di memoria della prossima istruzione da eseguire;
- MBR (Memory Buffer Register): contiene l'istruzione prelevata dalla memoria;
- SP (Stack Pointer): contiene un puntatore alla testa dello stack;
- TOS (Top of stack): contiene il valore in testa allo stack;
- LV (Local Variables): punta alla base dell'area di memoria in cui si trovano le variabili locali di un metodo;
- H (Holding): contiene il primo operando dell'ALU;
- CPP (Constant Pool Pointer): è un puntatore a valori costanti, utilizzato per semplificare la gestione della macchina virtuale;
- OPC (Scratch Register): permette di "appoggiare" qualcosa di utile nella micro-istruzione.

Dallo schema dell'unità operativa, si può notare come si impone, ad ogni ciclo del datapath, il passaggio attraverso l'ALU, per semplificare i circuiti, non dovendo implementare un clock a lunghezza variabile, con una lunghezza a seconda dell'istruzione considerata.

4.2.3 Unità di controllo

L'unità di controllo del processore MIC-1 è realizzata in logica micro-programmata: le sequenze di controllo sono memorizzate all'interno di una ROM 512x36 bit.

Le microistruzioni contenute nella control store, oltre a specificare i segnali di controllo necessari a impartire i giusti comandi alla parte operativa, contengono il prossimo indirizzo da inserire all'interno del micro-program counter, alcuni bit utilizzati per gestire le condizioni di salto condizionato in corrispondenza dei segnali di stato Z e N forniti dall'ALU e il prelievo delle istruzioni dal MBR, che viene utilizzato, in questa architettura, come instruction register.

I codici operativi utilizzati per identificare le specifiche istruzioni a livello architetturali sono gli entry-point all'interno della control store della sequenza di microistruzioni che le implementano a livello micro-architetturale. Sfruttando la proprietà secondo la quale solo un registori alla volta può scrivere sul bus B, si può ridurre il numero di bit necessari per memorizzare i segnali di controllo relativi alla scrittura sul bus, tramite l'utilizzo di un decoder. In generale, si parla di architettura orizzontale (senza una codifica) rispetto ad un'architettura orizzontale (con un maggior grado di codifica e con dell'hardware aggiuntivo per risolverlo).

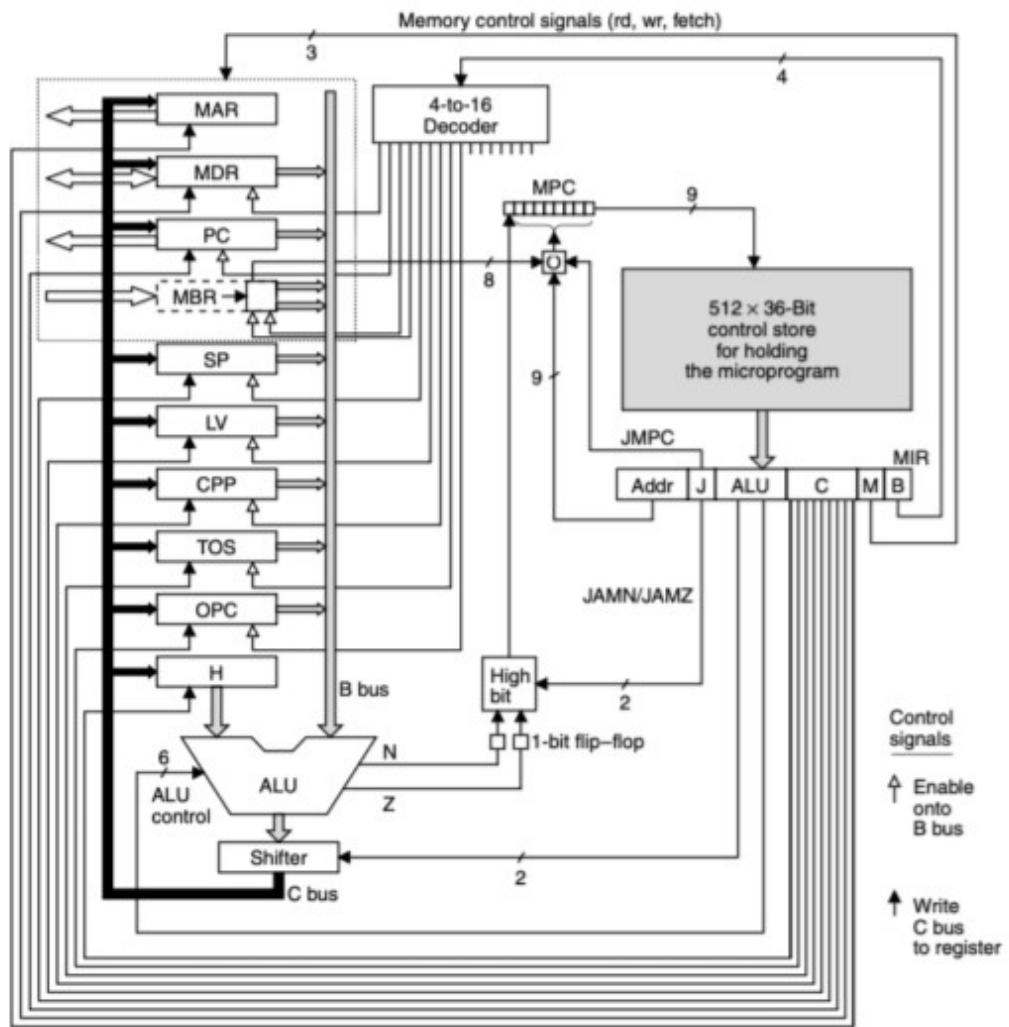


Figura 4.2: Schema complessivo del MIC-1

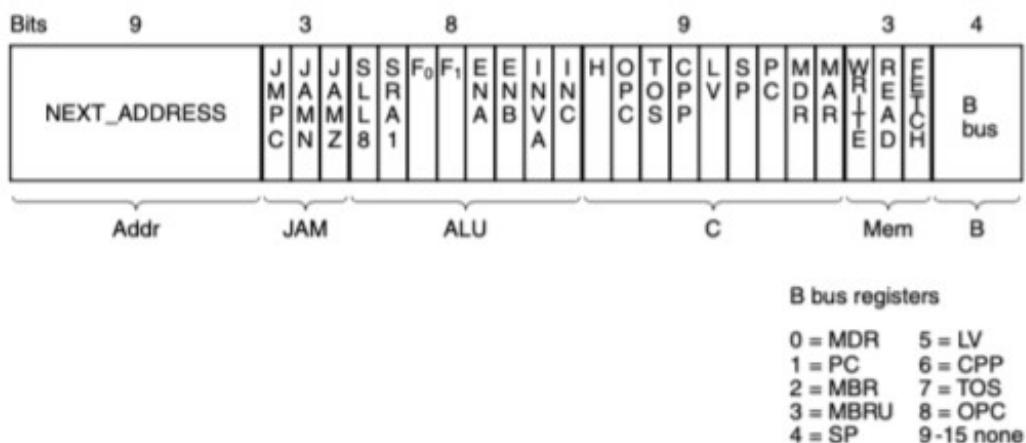


Figura 4.3: Formato della microistruzione

4.2.4 Approfondimento di alcune istruzioni

Bipush

L'operazione di bipush permette di caricare un byte sullo stack. La prima microistruzione per la realizzazione della bipush è memorizzata nella locazione con indirizzo 0x10 della control store. L'operazione di bipush, essendo il MIC-1 un processore basato su un modello a stack, risulta essere fondamentale in quanto molte operazioni prevedono il caricamento di qualche valore sullo stack. L'istruzione ha una lunghezza complessiva di due byte, uno per il codice operativo, uno per specificare il valore da caricare sullo stack.

```
bipush = 0x10:  
    SP = MAR = SP + 1  
    PC = PC + 1; fetch  
    MDR = TOS = MBR; wr; goto main
```

Figura 4.4: Sequenza microistruzioni bipush

Illustriamo adesso il funzionamento dell'istruzione, analizzando la sequenza di microistruzioni ad essa associata:

- La prima microistruzione ha lo scopo di incrementare il valore dello stack pointer e inserire il nuovo valore incrementato all'interno del MAR, predisponendo il successivo effettivo caricamento sullo stack;
- La seconda microistruzione ha lo scopo di incrementare il program counter. Prima dell'incremento, il program counter pun-

tava all'indirizzo del valore da caricare nello stack (operando dell'istruzione). In questa fase viene anche predisposta la fase di fetch per la prossima istruzione;

- La terza microistruzione ha lo scopo di memorizzare nel TOS e nel MDR il valore da caricare sullo stack, attualmente memorizzato nel MBR in quanto operando dell'istruzione bipush. Con l'operazione di write, il valore verrà memorizzato all'interno di $SP + 1$, che è caricato all'interno del MAR, dopo la prima microistruzione. L'ultimo passo è il salto al microprogramma *main*.

Il microprogramma *main* garantisce il corretto proseguimento del ciclo del processore e quindi la corretta esecuzione del programma caricato in memoria. Infatti, il microprogramma *main* incrementa il program counter, esegue il fetch di un'istruzione e imposta come prossima microistruzione quella "puntata" dal registro *MBR*.

```
main:  
    PC = PC + 1; fetch; goto (MBR)
```

Figura 4.5: Sequenza microistruzioni main

Notiamo inoltre che il valore da caricare sullo stack sarà già disponibile nel MDR all'atto dell'esecuzione della bipush, in quanto il MIC-1 effettua una fase di fetch "anticipata".

Di seguito è riportato un esempio di simulazione dell'istruzione, caricando sullo stack il valore 0x0A.

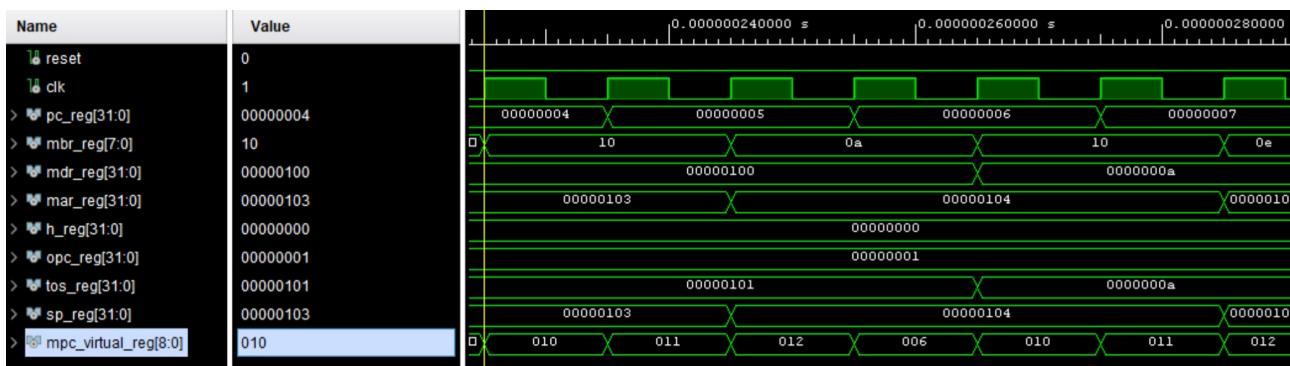


Figura 4.6: Esempio esecuzione bipush

Osserviamo come il valore dello stack pointer venga incrementato e il valore del registro *TOS* sia assegnato al valore desiderato 0xA.

Isub

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

Figura 4.7: Sequenza microistruzioni isub

Illustriamo adesso il funzionamento dell'istruzione, analizzando la sequenza di microistruzioni ad essa associata:

- Ricordando che vale il seguente invarianto, ovvero quando viene eseguita la prima istruzione della ISUB, il *valore* in testa allo stack è già presente nel registro TOS. Dunque si comincia a predisporre il registro MAR a contenere l'elemento subito sotto alla testa dello stack, ovvero *il valore dello stack pointer decrementato di uno*. In questo ciclo viene attivato il segnale di controllo

che attiva l'operazione di read. Si noti che se rd è alto non è detto che è possibile utilizzare il valore al ciclo successivo, ma bisogna lasciare un ciclo di latenza per far sì che il registro MAR si aggiorni, aspettando il prossimo colpo di clock.

- Durante questo ciclo si sfrutta la latenza presente nel primo ciclo per portare il contenuto di TOS nel registro H, in modo da predisporlo come primo operando dell'operazione; dunque $H = TOS$, ovvero TOS deve alimentare il bus B, l'ALU è configurata in modo tale da non alterarne il valore, arrivando sul bus C, andando ad alimentare il registro H, in modo tale da poterlo utilizzare come operando al prossimo ciclo. Durante questo ciclo stanno anche avvenendo quella serie di operazioni verso la memoria richieste durante il ciclo precedente. Dunque adesso MAR contiene l'attuale valore di stack pointer, e il registro rd è andato alto. Al prossimo ciclo di clock MDR sarà aggiornato.
- Durante il terzo ciclo sia MDR che H adesso contengono il valore dell'operando. Quindi con i due operandi possiamo operare sui bus in ingresso, A e B, dell'ALU, dunque $MDR = TOS = MDR - H$, ovvero siamo interessati a due cose, la prima di sottrarre i due operandi e dunque avere l'ALU configurata per una somma e quindi alimentare il bus C con tale somma. Dopodiché bisogna mantenere vera l'invariante del valore riguardo TOS, assicurandoci che il valore della somma vi sia presente, inoltre

bisogna anche scriverlo in memoria assicurandoci che sia effettivamente in testa allo stack, dunque la somma deve finire anche in MDR in modo da scriverlo in memoria.

4.2.5 Modifica di una microistruzione

Per questo esercizio abbiamo deciso di modificare l'istruzione IAND e trasformarla in una IOR. Il funzionamento della IAND standard è simile a quello della ISUB, con l'unica differenza che, nell'ultima microistruzione, non viene effettuata la sottrazione ma l'operazione di AND.

```
iand = 0x7E:  
    MAR = SP = SP - 1; rd  
    H = TOS  
    MDR = TOS = MDR AND H; wr; go to main
```

Figura 4.8: Sequenza microistruzioni IAND

```
iand = 0x7E:  
    MAR = SP = SP - 1; rd  
    H = TOS  
    MDR = TOS = MDR OR H; wr; goto main
```

Figura 4.9: Sequenza microistruzioni IAND modificata

Di seguito è riportato un esempio di esecuzione della IAND modificata con il relativo codice MAL del programma di test. L'operazione viene effettuata tra i valori 0xC e 0x6, restituendo 0xE (comportandosi quindi come se fosse una OR), e non 0x4.

```
.main
.var
res
.endvar
BIPUSH 0xC
BIPUSH 0x6
IAND
ISTORE res
HALT
.endmethod
```

Figura 4.10: Programma di test

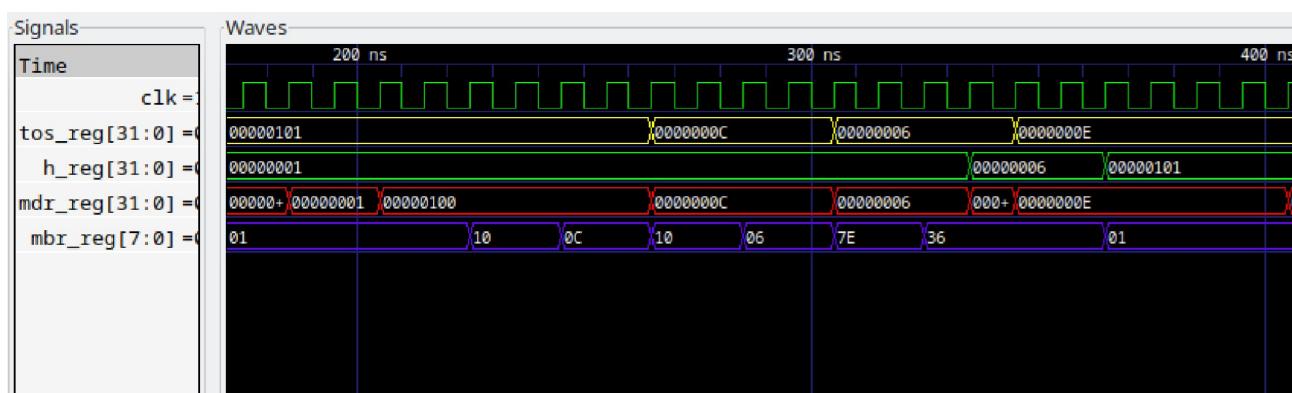


Figura 4.11: Simulazione IAND modificata

Capitolo 5

Interfaccia seriale

5.1 Traccia

5.1.1 Esercizio 9.1

Sfruttando l'implementazione fornita dalla Digilent di un dispositivo UART (componente RS232RefComp.vhd), progettare e implementare in VHDL un sistema costituito da 2 nodi A e B collegati tra loro mediante una interfaccia seriale. Il sistema A acquisisce una stringa di 8 bit dall'utente (mediante gli switch della board di sviluppo) e la invia mediante la seriale al sistema B, che la manda in output sui led della board di sviluppo.

5.1.2 Esercizio 9.2

Come variante dell'esercizio 8.1, il sistema A invia al sistema B tramite l'interfaccia seriale N stringhe di 8 bit contenute all'interno di una memoria ROM. Le stringhe ricevute vengono memorizzate in una memoria locale a B. Il progetto deve prevedere che A utilizzi un componente contatore per scandire le N stringhe da inviare.

5.2 Esercizio 9.1

Per la realizzazione è stato utilizzato l'implementazione del componente RS232 fornito da Digilent. In particolare, gli switch sono utilizzati per scegliere gli 8 bit acquisiti dal trasmettitore, mappati da pin J15 al pin R13. Inoltre sono stati utilizzati altri due switch per fornire l'abilitazione di scrittura per il trasmettitore, per ottenere i dati in parallelo in ingresso da poi inviare in serie, e l'abilitazione di lettura del ricevitore per ricevere i dati in serie, rispettivamente mappati su V10 e U11. Infine i led sono stati configurati in modo da accendersi nello posizione omologhe degli switch alzati, dal pin H17 al pin U16.

5.2.1 Nodo A

Il nodo A prende in ingresso la stringa di bit in parallelo, `data_p_in`, e la manda in uscita in seriale tramite il trasmettitore su `data_s_out`.

Vi è presente il segnale di abilitazione per la scrittura en_wr.

```
1 entity SysA is
2     Port(
3         clock      :  in  std_logic;
4         reset      :  in  std_logic;
5         data_p_in  :  in  std_logic_vector(7 downto 0);
6         en_wr      :  in  std_logic;
7         data_s_out :  out std_logic
8     );
9 end SysA;
```

Nel port mapping con il componente implementativo dell'UART è stato mappato DBIN con il segnale parallelo in ingresso data_p_in, mentre TXD con il segnale di uscita seriale data_s_out.

5.2.2 Nodo B

Il nodo B riceve in ingresso la stringa di bit in modalità seriale su data_s_in, e l'uscita in parallelo su data_p_out.

```
1 entity SysB is
2     Port(
3         clock      :  in  std_logic;
4         reset      :  in  std_logic;
5         data_s_in  :  in  std_logic;
6         en_rd      :  in  std_logic;
7         data_p_out :  out std_logic_vector(7 downto 0)
8     );
```

```
9 end SysB;
```

Nel port mapping la porta RXD della UART è collegata con l'ingresso seriale data_s_in, mentre la porta DBOUT è collegata all'uscita in parallelo data_p_out. Vi è presente il segnale di enable per la lettura, en_rd.

Top module

```
1 entity System is
2   Port (
3     CLK          :  in  std_logic;
4     RST          :  in  std_logic;
5     EN_WR        :  in  std_logic;
6     EN_RD        :  in  std_logic;
7     DATA_IN      :  in  std_logic_vector(7 downto 0);
8     DATA_OUT     :  out std_logic_vector(7 downto 0)
9   );
10 end System;
```

Riportiamo ora il port mapping

```
1 nodeA : SysA
2   Port map(
3     clock      => CLK,
4     reset      => RST_N,
5     data_p_in  => DATA_IN,
6     en_wr      => EN_WR,
```

```
7      data_s_out  => data_s_out_a  
8  );  
9  
10 nodeB : SysB  
11 Port map(  
12     clock      => CLK,  
13     reset      => RST_N,  
14     data_s_in   => data_s_out_a,  
15     en_rd      => EN_RD_N,  
16     data_p_out  => DATA_OUT  
17 );
```

L'ingresso DATAIN del top module è mappato tramite gli switch della board, il quale rappresenta il dato in ingresso parallelo, mappato infatti con il nodo A, il quale lo riceve in ingresso su data_p_in. Il nodo A presenta l'uscita seriale data_s_out_a, mappato a sua volta in ingresso al nodo B su data_s_in. Infine l'uscita del nodo B, ovvero data_p_out, viene mappata sull'uscita del sistema complessivo, DATAOUT, dove quest'ultimo è mappato tramite i led della board per mostrare il dato.

5.3 Esercizio 9.2

Per la comunicazione tra i due nodi A e B si è proceduti in maniera analoga all'esercizio precedente.

5.3.1 Unità operativa – entità A

L’unità operativa è stata realizzata mediante un approccio strutturale.

```
1 entity SysA is
2     Port(
3         clock      :  in  std_logic;
4         reset      :  in  std_logic;
5         startA     :  in  std_logic;
6         btn_load   :  in  std_logic;
7         txd_out    :  out std_logic; -- serial data out
8     );
9 end SysA;
```

L’unità operativa ha come ingressi un segnale `startA` il quale servirà per inizializzare il sistema, il quale è stato mappato tramite un bottone della board. Ulteriormente si ha un segnale di abilitazione per la lettura dalla memoria ROM dei dati da inviare, `btn_load`, anch’esso mappato con un bottone. Il segnale `txd_out` è l’uscita della porta UART dei dati trasmessi in uscita in modalità seriale.

5.3.2 Unità di controllo – entità A

L’unità di controllo di A è stata realizzata in logica cablata, mediante un’automa a stati finiti. L’unità di controllo si occupa di far leggere i valori dalla memoria ROM e di far avviare la trasmissione.

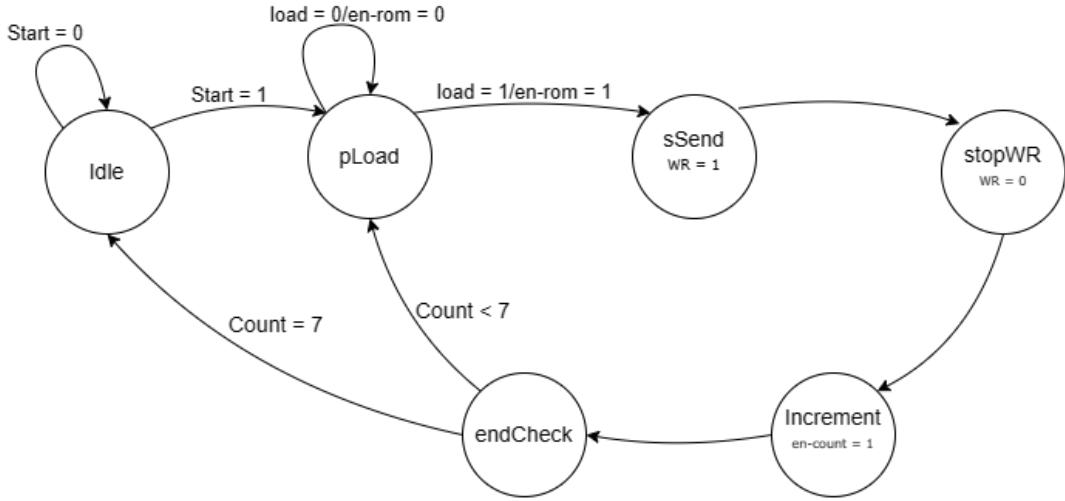


Figura 5.1: Automa unità di controllo entità A

L'automa rappresentato la rete di controllo di A evolve attraverso i seguenti stati: *idle*, *pLoad*, *sSend*, *stopWR*, *increment*, *endCheck*. L'automa parte dallo stato di *idle* e vi permane fin quando un segnale di *start*, proveniente dall'esterno, non diventa alto. Una volta diventato alto, si va nello stato di *pLoad*, nel quale viene fatta la lettura dei dati dalla memoria ROM, andando ad alzare l'opportuna abilitazione, tuttavia è stato scelto di far in modo di attivare la lettura dalla ROM solo qualora si premesse il bottone sulla board dedicato, questo viene controllato tramite un segnale in ingresso all'unità di controllo *load*, il quale quando è alto allora viene alzato anche il segnale in uscita *en_rom* così da iniziare la lettura. Successivamente si entra nello stato *sSend* dove vengono inviati i dati in seriale, e dunque viene alzato il segnale di write strobe della UART, tramite il segnale in uscita della control unit *wr*. Nello stato successivo si entra in *stopWR* dove semplicemente viene abbassato il segnale di write strobe, e dun-

que una volta basso allora viene alzato il segnale interno della UART per segnalare che il buffer è vuoto, ovvero TBE – Text Buffer Empty, successivamente viene alzato il segnale RDA – Read Data Available per indicare al ricevitore che i dati sono stati trasmessi e sono pronti per essere letti. Il prossimo stato, *increment*, server per alzare l'abilitazione del contatore, tramite `en_count`, per aggiornare l'indirizzo di lettura dalla memoria. Infine si entra nello stato *end-Check* dove si verifica qualora il contatore sia arrivato al conteggio massimo, e dunque ritornare nello stato di *idle*, oppure nel caso contrario ritornare nello stato di *pLoad*, dove vi si permane fino a quando non verrà premuto il bottone per alzare il segnale `load`, come citato prima.

Mostriamo l'entity dell'unità di controllo di A, la quale fornisce all'unità di controllo i diversi segnali di controllo discussi prima.

```
1 entity Transmitter is
2   Port (
3     clock      : in std_logic;
4     reset      : in std_logic;
5     start      : in std_logic;
6     count      : in std_logic_vector(0 to 2);
7     load       : in std_logic;
8     en_rom    : out std_logic;
9     en_count  : out std_logic;
10    wr        : out std_logic -- uart write strobe signal
```

```

11 );
12 end Transmitter;

```

Il segnale `start` è il segnale che fa uscire dallo stato di *idle*. Il segnale in ingresso `load` è mappato con un bottone sulla scheda, il quale viene controllato se alto in modo da alzare l'abilitazione della memoria ROM, `en_rom`, per la lettura da essa. Il segnale in ingresso `count` è il valore del conteggio per l'indirizzo in memoria, mentre `en_count` è il segnale in uscita per abilitare il contatore. Il segnale in uscita `wr` è il write strobe dello UART per far iniziare la trasmissione.

```

1 type state is (idle, pLoad, sSend, stopWR, increment, endCheck) ;
2
3 signal curr : state := idle;
4 signal count_val : integer := 0;
5
6 begin
7
8   count_val <= to_integer(unsigned(count));
9
10 fsmA : process(clock)
11 begin
12   if reset = '1' then
13     en_rom      <= '0';
14     en_count    <= '0';
15     wr          <= '0';
16     curr        <= idle;
17   end if;

```

```

18
19   if rising_edge(clock) then
20
21     case curr is
22
23       when idle =>
24
25         en_rom      <= '0';
26
27         en_count    <= '0';
28
29         wr          <= '0';
30
31         if start = '1' then
32
33           curr <= pLoad;
34
35         else
36
37           curr <= idle;
38
39         end if;
40
41       when pLoad =>  -- read (parallel) data from ROM
42
43         en_rom      <= '0';
44
45         en_count    <= '0';
46
47         wr          <= '0'; -- uart write strobe
48
49         if load = '1' then
50
51           en_rom <= '1';
52
53           curr <= sSend;
54
55         else
56
57           en_rom <= '0';
58
59           curr <= pLoad;
60
61         end if;
62
63       when sSend =>
64
65         en_rom      <= '0';
66
67         en_count    <= '0';
68
69         wr          <= '1'; -- begin trasmission of (serial)
70
71         data

```

```
45      curr      <= stopWR;
46      when stopWR =>
47          en_rom    <= '0';
48          en_count  <= '0';
49          wr        <= '0';
50          curr      <= increment;
51      when increment =>
52          en_rom    <= '0';
53          en_count  <= '1';
54          wr        <= '0';
55          curr      <= endCheck;
56      when endCheck =>
57          en_rom    <= '0';
58          en_count  <= '0';
59          wr        <= '0';
60          if count_val = 7 then
61              curr <= idle;
62          else
63              curr <= pLoad;
64          end if;
65      end case;
66  end if;
67 end process;
```

5.3.3 Unità operativa – entità B

L'unità operativa è stata realizzata mediante un approccio strutturale.

```
1 entity SysB is
2     Port (
3         clock      : in std_logic;
4         reset      : in std_logic;
5         rx_in      : in std_logic;
6         data_p_out : out std_logic_vector(7 downto 0)
7     );3
8 end SysB;
```

L'unità operativa ha come ingresso `rx`_in, per ricevere i dati trasmessi in seriale. In uscita si ha il valore in parallelo `data_p_out`.

5.3.4 Unità di controllo – entità B

L'unità di controllo di A è stata realizzata in logica cablata, mediante un'automa a stati finiti. L'unità di controllo ha lo scopo di ricevere i dati in seriale e di scriverli sulla memoria del nodo B, per poi successivamente leggere da quest'ultima e accendere i led sulla board.

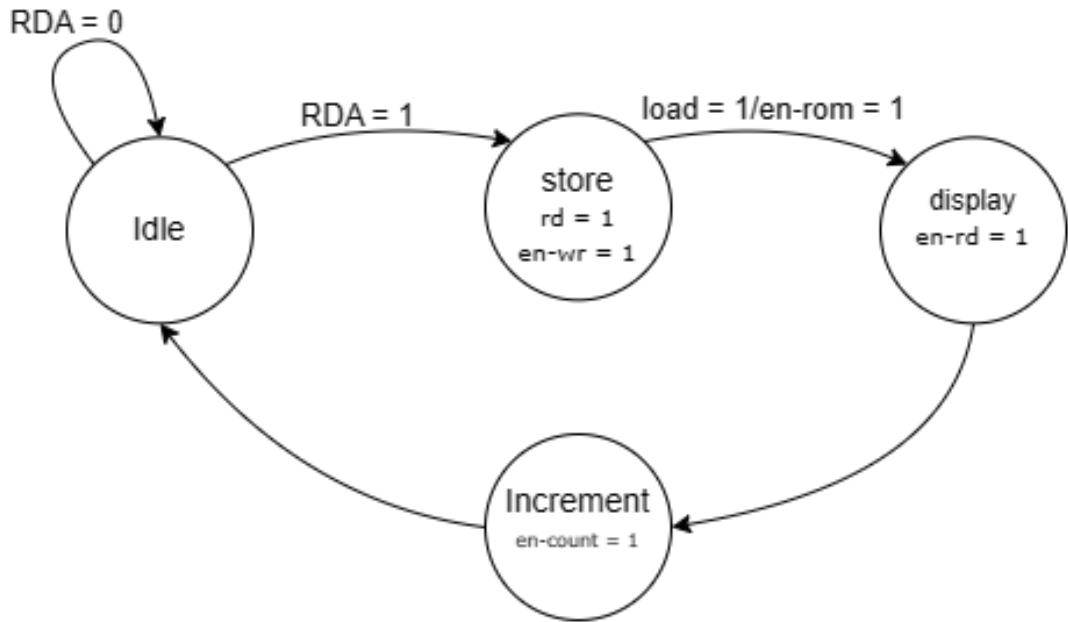


Figura 5.2: Automa unità di controllo entità B

L'automa che rappresenta la rete di controllo di B evolve attraverso i seguenti stati: *idle*, *store*, *display*, *increment*. L'automa parte dallo stato di *idle* e vi permane fin quando non si alza il segnale di RDA, il quale ricordiamo sarà alto solo qualora la trasmissione seriale sarà terminata e dunque sarà possibile la lettura in parallelo. Successivamente si avanza nello stato nello stato di *store*, dove si alza il segnale d'uscita *rd*, ovvero il read strobe dell'UART, viene anche alzato il segnale per l'abilitazione della memoria, *en_wr*. Una volta memorizzato il valore in memoria lo si vuole mostrare accendendo i LED sulla board, avanziamo dunque nello stato *display*, dove in questo caso dalla memoria effettuiamo una lettura, alzando il segnale di abilitazione per la stessa memoria *en_rd*. Infine si arriva nello stato di *increment*, dove si alza l'abilitazione del contatore, *en_count*, per aggiornare l'indirizzo

della memoria, per poi tornare nello stato di *idle*.

Mostriamo l'entity dell'unità di controllo B.

```
1 entity Receiver is
2
3     Port(
4         clock      : in std_logic;
5         reset      : in std_logic;
6         en_wr      : out std_logic;
7         en_rd      : out std_logic;
8         en_count   : out std_logic;
9         rda        : in std_logic;
10        rd         : out std_logic
11    );
12 end Receiver;
```

Il segnale RDA è il segnale che ci permette di uscire dal segnale di *idle*, il quale ricordiamo sarà alto solo qualora saranno arrivati tutti i bit al ricevitore, e dunque è possibile iniziare la lettura. Si hanno i segnali di abilitazione in uscita, quali `en_wr` e `en_rd` che sono rispettivamente per la scrittura e la lettura della memoria del nodo B. Vi è in fine il segnale read strobe, `rd`, per poter leggere i dati ricevuti.

```
1 type state is (idle, store, display, increment);
2
3 signal curr : state := idle;
4
5 begin
```

```

7  fsmB : process(clock)
8 begin
9   if reset = '1' then
10    rd      <= '0';
11    en_wr   <= '0';
12    en_rd   <= '0';
13    en_count <= '0';
14    curr     <= idle;
15  end if;
16
17  if rising_edge(clock) then
18    case curr is
19      when idle =>
20        rd      <= '0';
21        en_wr   <= '0';
22        en_rd   <= '0';
23        en_count <= '0';
24        if rda = '1' then
25          curr <= store;
26        else
27          curr <= idle;
28        end if;
29      when store =>
30        rd      <= '1';
31        en_wr   <= '1'; — scrivi in memoria
32        en_rd   <= '0';
33        en_count <= '0';
34        curr     <= display;

```

```

35      when display =>
36          rd          <= '0';
37          en_wr       <= '0';
38          en_rd       <= '1'; — leggi dalla memoria
39          en_count    <= '0';
40          curr        <= increment;
41
42      when increment =>
43          rd          <= '0';
44          en_wr       <= '0';
45          en_rd       <= '0';
46          en_count    <= '1';
47          curr        <= idle;
48
49      end case;
50
51  end if;

```

5.3.5 Sistema complessivo

Infine, è riportato lo schematico del sistema complessivo, il quale oltre al segnale di clock e reset prevede in ingresso il segnale START, per poter far partire il sistema, il quale è mappato tramite un bottone della board, e il segnale LOAD mappato con un altro bottone, per poter dare l'abilitazione della lettura dalla ROM del nodo A, come è stato discusso nella sezione precedente. Infine si ha il segnale DATAOUT il quale viene mappato con gli appositi LED sulla scheda per mostrare il dato salvato nella memoria del nodo B.

Mostriamo dunque l'entity del sistema complessivo e i componen-

ti da esso utilizzati, quali sono i due nodi A e B, e il Button Debouncer.

```
1 entity System is
2     Port(
3         CLOCK      :  in  std_logic;
4         RESET      :  in  std_logic;
5         START      :  in  std_logic;
6         LOAD       :  in  std_logic;
7         DATA_OUT   :  out std_logic_vector(7 downto 0)
8     );
9 end System;
10
11 architecture Structural of System is
12
13 component SysA is
14     Port(
15         clock      :  in  std_logic;
16         reset      :  in  std_logic;
17         startA    :  in  std_logic;
18         btn_load  :  in  std_logic;
19         txd_out   :  out std_logic -- dati seriali in uscita dal
20             trasmettitore
21     );
22
23 component SysB is
24     Port(
25         clock      :  in  std_logic;
```

```

26      reset      :  in  std_logic;
27      rxd_in     :  in  std_logic;
28      data_p_out :  out std_logic_vector(7 downto 0) -- dati
29          letti dal ricevitore, in uscita per essere mappati con
30          DATA_OUT
31      );
32 end component SysB;
33
34 component ButtonDebouncer is
35     Generic(
36         CLK_period      : integer := 10;
37         btn_noise_time : integer := 10000000
38     );
39     Port(
40         rst      :  in  std_logic;
41         clk      :  in  std_logic;
42         btn      :  in  std_logic;
43         clrd_btn :  out std_logic
44     );
45 end component ButtonDebouncer;

```

Riportiamo i segnali intermedi utilizzati.

```

1  -- segnale temporaneo mappato con l'uscita del nodo A, e posto in
2  -- ingresso rxd al nodo B
3  -- in modo che a sua volta venga mappato con la porta RXD della UART
4  -- usata dal nodo B
5  signal txd_tmp      : std_logic;

```

```

4
5 -- segnali ripuliti dei buttoni mappati per lo START e il LOAD
6 signal clrd_start : std_logic := '0';
7 signal clrd_load : std_logic := '0';

```

Mostriamo infine il port mapping effettuato.

```

1 begin
2
3 nodeA : SysA
4   Port map(
5     clock      => CLOCK,
6     reset       => RESET,
7     startA     => clrd_start,
8     btn_load   => clrd_load,
9     txd_out    => txd_tmp
10   );
11
12 nodeB : SysB
13   Port map(
14     clock      => CLOCK,
15     reset       => RESET,
16     rxd_in    => txd_tmp,
17     data_p_out => DATA_OUT
18   );
19
20 start_btn : ButtonDebouncer
21   Generic map(

```

```
22      CLK_period      => 10,
23      btn_noise_time  => 650000000
24  )
25  Port map(
26      rst      => RESET,
27      clk      => CLOCK,
28      btn      => START,
29      clrd_btn => clrd_start
30  );
31
32 load_btn : ButtonDebouncer
33 Generic map(
34      CLK_period      => 10,
35      btn_noise_time  => 650000000
36  )
37 Port map(
38      rst      => RESET,
39      clk      => CLOCK,
40      btn      => LOAD,
41      clrd_btn => clrd_load
42  );
43
44 end Structural;
```

Capitolo 6

Switch multistadio

6.1 Traccia

Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network. Lo switch progettato deve operare come segue:

- a) Lo switch deve consentire lo scambio di messaggi di 2 bit ciascuno da un nodo sorgente a un nodo destinazione in un rete con 4 nodi, implementando uno schema a priorità fissa fra i nodi (ed. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).

6.2 Esercizio 10 – Omega Network

6.2.1 Introduzione

Una rete di interconnessione è un sistema che permette di smistare le comunicazioni tra un insieme di sorgenti e un insieme di destinazioni. La rete di interconnessione viene realizzata mediante switch e si suddividono in due categorie:

- a singolo stadio (diretti);
- a N stadi (indiretti).

Switch a connessione diretta

Per la realizzazione di uno switch a connessione diretta sono necessarie due informazioni: l'indirizzo del nodo sorgente e l'indirizzo del nodo destinazione. Questo meccanismo, per come è progettato, realizza una mutua esclusione tra i nodi, dovuta al fatto che solo un collegamento è attivo contemporaneamente. Affinché si possa comunicare in parallelo, bisogna replicare necessariamente l'hardware, creando più collegamenti. Questa tipologia di rete viene realizzata attraverso un mux e un demux.

Questa soluzione non è esente dal problema dei conflitti, poiché aumentando il parallelismo in hardware si va semplicemente a rendere possibile la comunicazione parallela, ma se i nodi di ingresso vogliono

no comunicare con lo stesso nodo di uscita è inevitabile che si creano collisioni.

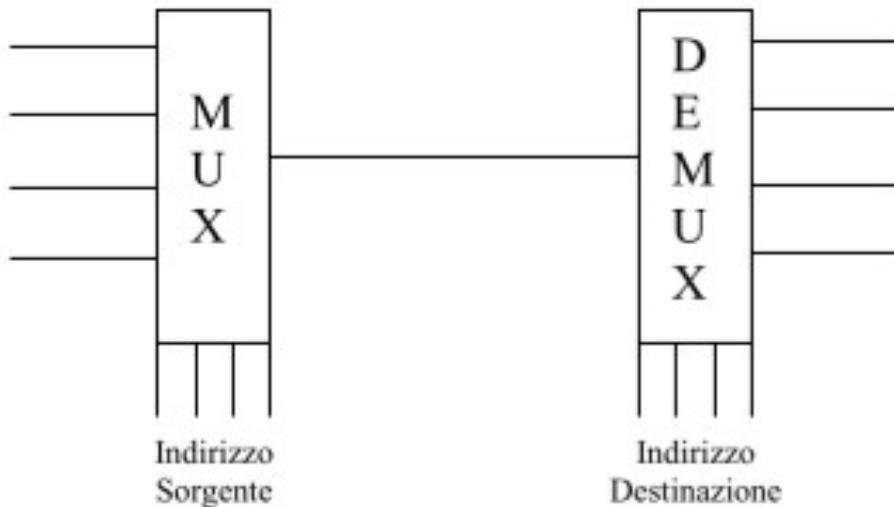


Figura 6.1: Switch a singolo stadio

Switch a connessione indiretta

Per la realizzazione di uno switch a connessione indiretta, si rende necessario scomporre gli indirizzi in tante parti quanti sono gli stadi, in modo da comandare opportunamente i blocchi. In questo modo, non esiste una logica di controllo centralizzata ma la logica di controllo è distribuita negli stadi. Il nodo a cui arriva il messaggio analizza l'indirizzo e si comporta di conseguenza.

Una tecnica molto utilizzata per l'implementazione di questo tipo di switch è la tecnica del perfect shuffling, che fa riferimento al modo con cui le carte di un mazzo vengono mischiate. Infatti, si divide il mazzo di carte in due parti uguali e si mischiano perfettamente e, dopo e

$\log_2 n$ volte, si ritorna all'ordine iniziale. Questa tecnica ci permette di ottimizzare i percorsi da seguire indicando i modi di collegamento tra i diversi stadi. Mischiare perfettamente significa che la prima carta viene accoppiata con la prima della seconda metà, la seconda della prima metà con la seconda della seconda metà, e così via.

In base al bit dell'indirizzo di destinazione associato ad ogni stadio, viene selezionato il collegamento da percorrere, utilizzando il seguente criterio:

- Se il valore del bit è 0, si procede con il ramo di uscita superiore del blocco considerato;
- Se il valore del bit è 1, si procede con il ramo di uscita inferiore del blocco considerato.

Esempio: Dal nodo 2 si vuole andare al nodo 4; indirizzo 100, quindi:

I stadio Giù; II stadio Su; III stadio Su → **N.B. L'indirizzo vede considerato dal bit più significativo a quello meno significativo**

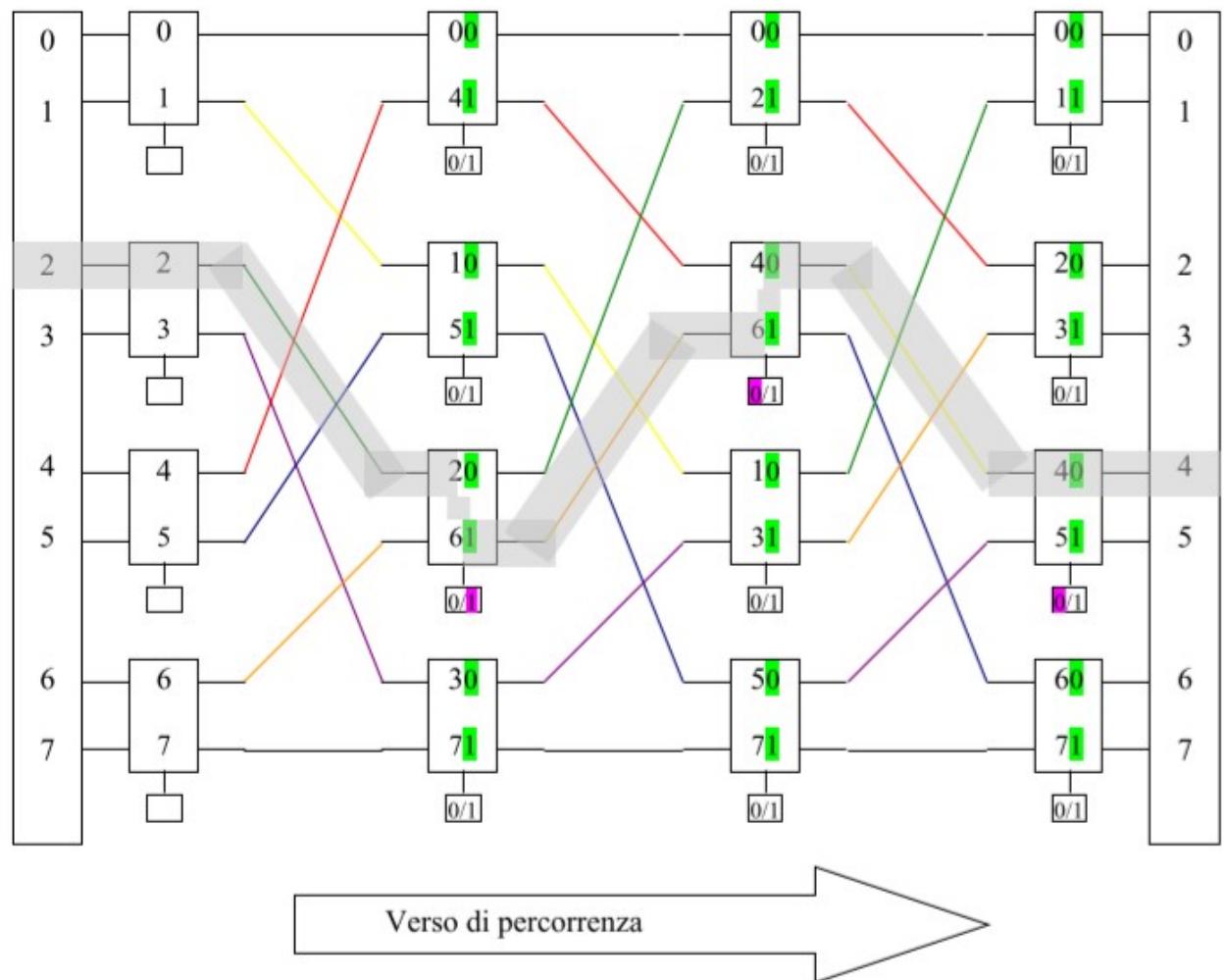


Figura 6.2: Funzionamento del perfect shuffling

6.2.2 Unità operativa

L'unità operativa è stata realizzata seguendo un approccio strutturale.

Multiplexer 2:1

Il multiplexer 2:1 è stato realizzato mediante un approccio dataflow

```
1 entity mux_2_1 is
2 port(
3     x0 : in std_logic_vector(0 to 1);
4     x1 : in std_logic_vector(0 to 1);
5     s : in std_logic;
6     y : out std_logic_vector(0 to 1)
7 );
8 end mux_2_1;
9
10 architecture Dataflow of mux_2_1 is
11 begin
12
13     with s select
14         Y <= x0 when '0',
15             x1 when '1',
16             "00" when others;
17 end Dataflow;
```

Demultiplexer 1:2

Anche il demultiplexer è stato realizzato mediante un approccio data-flow.

```
1 entity demux_1_2 is
2   port(
3     x : in std_logic_vector(0 to 1);
4     s : in std_logic;
5     y0 : out std_logic_vector(0 to 1);
6     y1 : out std_logic_vector(0 to 1)
7   );
8 end demux_1_2;
9
10 architecture Dataflow of demux_1_2 is
11 begin
12   y0 <= x when s = '0' else "00";
13   y1 <= x when s = '1' else "00" ;
14
15 end Dataflow;
```

Switch

Il singolo switch è stato realizzato in modo strutturale, con un mux e un demux. In ingresso allo switch le linee di ingresso, con l'abilitazione del multiplexer gestita dal bit della sorgente e in uscita allo switch abbiamo le linee di uscita, con la selezione del demultiplexer associata al bit di destinazione.

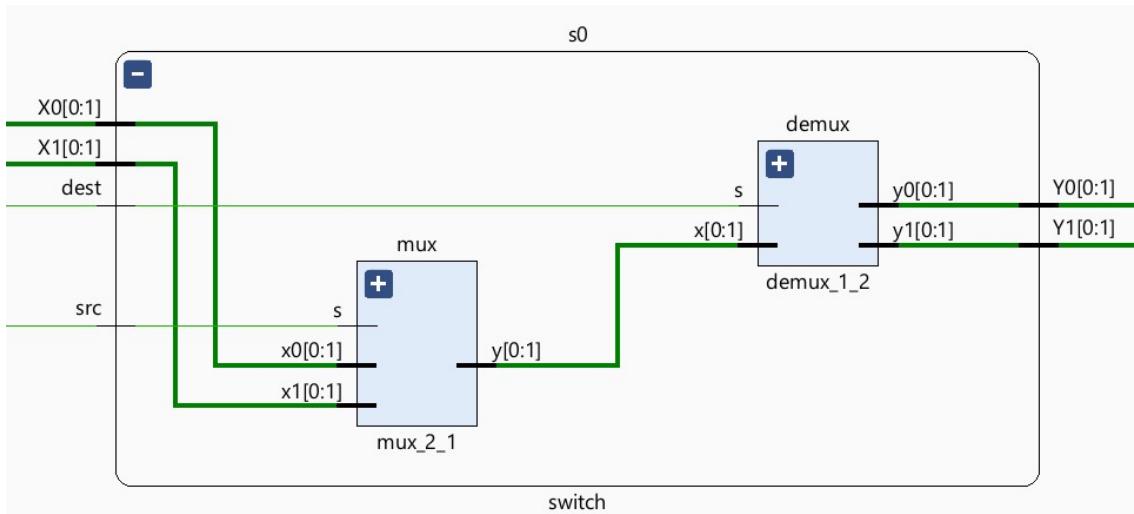


Figura 6.3: Schematic dello switch

Di seguito è riportata l'implementazione in VHDL.

```

1  entity switch is
2
3      port (
4          x0      : in  std_logic_vector(0 to 1);
5          x1      : in  std_logic_vector(0 to 1);
6          src     : in  std_logic;
7          dest    : in  std_logic;
8          y0      : out std_logic_vector(0 to 1);
9          y1      : out std_logic_vector(0 to 1)
10         );
11
12 end switch;
13
14
15 component mux_2_1 is
16
17     port (
18         x0 : in  std_logic_vector(0 to 1);
19
20         dest: in  std_logic;
21
22         s   : out std_logic;
23
24         y0 : out std_logic;
25
26         y1 : out std_logic;
27
28     );
29
30 end component;
31
32
33 architecture Structural of switch is
34
35     begin
36
37         u1: mux_2_1
38             port map (x0=>x0, dest=>dest, s=>s, y0=>y0, y1=>y1);
39
40         dest <= x0;
41
42         dest <= x1;
43
44         dest <= src;
45
46         dest <= s;
47
48         dest <= y0;
49
50         dest <= y1;
51
52     end;
53
54
55 end;

```

```
17      x1 : in std_logic_vector(0 to 1);  
18      s : in std_logic;  
19      y : out std_logic_vector(0 to 1)  
20  );  
21 end component mux_2_1;  
22  
23 component demux_1_2 is  
24  port(  
25      x : in std_logic_vector(0 to 1);  
26      s : in std_logic;  
27      y0 : out std_logic_vector(0 to 1);  
28      y1 : out std_logic_vector(0 to 1)  
29  );  
30 end component demux_1_2;  
31  
32 signal temp : std_logic_vector(0 to 1) := "00";  
33  
34 begin  
35  
36   mux : mux_2_1  
37   port map(  
38     x0 => x0,  
39     x1 => x1,  
40     s  => src,  
41     y  => temp  
42  );  
43  
44   demux : demux_1_2
```

```
45  port map(
46      x  => temp,
47      s  => dest,
48      y0 => Y0,
49      y1 => Y1
50  );
51
52 end Structural;
```

Implementazione unità operativa

L'unità operativa è costituita da 4 switch. Notiamo che, in fase di port map, è necessario gestire bene le uscite degli switch al primo stadio, in modo da inoltrarle ai giusti ingressi degli switch del secondo livello, per implementare la tecnica del perfect shuffling.

```
1 entity UnitOperativa is
2   port(
3     x0  : in std_logic_vector(0 to 1);
4     x1  : in std_logic_vector(0 to 1);
5     x2  : in std_logic_vector(0 to 1);
6     x3  : in std_logic_vector(0 to 1);
7     src : in std_logic_vector(0 to 1);
8     dest : in std_logic_vector(0 to 1);
9     y0  : out std_logic_vector(0 to 1);
10    y1  : out std_logic_vector(0 to 1);
11    y2  : out std_logic_vector(0 to 1);
```

```
12      y3    : out std_logic_vector(0 to 1)
13  );
14 end Unita_Operativa;
15
16 architecture Structural of Unita_Operativa is
17
18 component switch is
19   port(
20     x0    : in  std_logic_vector(0 to 1);
21     x1    : in  std_logic_vector(0 to 1);
22     src   : in  std_logic;
23     dest  : in  std_logic;
24     y0    : out std_logic_vector(0 to 1);
25     y1    : out std_logic_vector(0 to 1)
26   );
27 end component switch;
28
29 --uscite degli switch del primo stadio
30 signal temp0 : std_logic_vector(0 to 1) := "00";
31 signal temp1 : std_logic_vector(0 to 1) := "00";
32 signal temp2 : std_logic_vector(0 to 1) := "00";
33 signal temp3 : std_logic_vector(0 to 1) := "00";
34
35 begin
36
37   s0 : switch
38   port map(
39     x0    => x0,
```

```
40      x1    => x1,
41      src   => src(1),
42      dest  => dest(0),
43      y0    => temp0,
44      y1    => temp1
45  );
46
47  s1 : switch
48  port map(
49      x0    => x2,
50      x1    => x3,
51      src   => src(1),
52      dest  => dest(0),
53      y0    => temp2,
54      y1    => temp3
55  );
56
57  s2 : switch
58  port map(
59      x0    => temp0,
60      x1    => temp2,
61      src   => src(0),
62      dest  => dest(1),
63      y0    => y0,
64      y1    => y1
65  );
66
67  s3 : switch
```

```

68     port map(
69         X0      => temp1,
70         X1      => temp3,
71         src     => src(0),
72         dest    => dest(1),
73         Y0      => y2,
74         Y1      => y3
75     );
76
77 end Structural;

```

Inoltre, per gestire correttamente l'indirizzo, gli switch del primo livello hanno come sorgente il primo bit del segnale *src* e come destinazione il secondo bit del segnale *dest*, per il secondo stadio avviene l'inverso. I segnali *src* e *dest* provengono dall'unità di controllo, dopo aver gestito la priorità dei nodi.

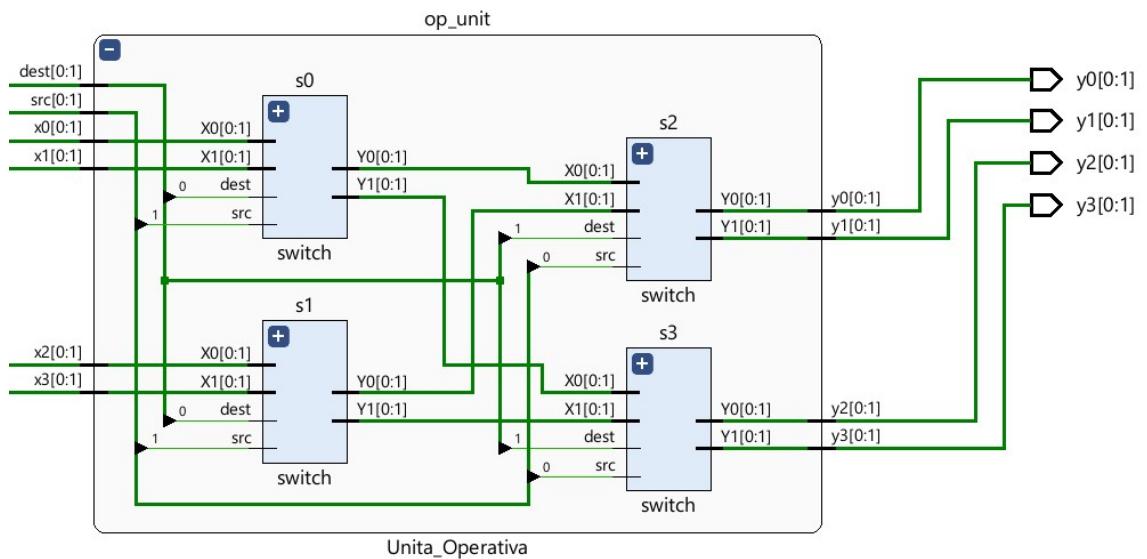


Figura 6.4: Schematic unità operativa

6.2.3 Unità di controllo

L'unità di controllo è stata progettata mediante un approccio strutturale. E' costituita da un multiplexer, un demultiplexer e una rete di priorità.

L'unità di controllo ha in ingresso un vettore di 6 bit, di cui i primi due indicano la sorgente, i secondi due bit indicano la destinazione e i restanti due bit indicano il dato effettivo da trasmettere. L'unità di controllo, in funzione dei segnali enable in ingresso alla rete di priorità, capisce quali nodi vogliono trasmettere. La rete di priorità, ha lo scopo di dare l'abilitazione ai mux/demux per permettere la trasmissione del nodo a maggiore priorità verso la destinazione preposta.

Multiplexer 4:1

Il multiplexer 4:1 è stato realizzato secondo un approccio dataflow.

```
1 entity mux_4_1 is
2     port(
3         x0 : in std_logic_vector(0 to 1);
4         x1 : in std_logic_vector(0 to 1);
5         x2 : in std_logic_vector(0 to 1);
6         x3 : in std_logic_vector(0 to 1);
7         s : in std_logic_vector(0 to 1);
8         Y : out std_logic_vector(0 to 1)
9     );
10 end mux_4_1;
```

11

```
12 architecture dataflow of mux_4_1 is
13
14 begin
15     with s select
16         Y <= X0 when "00",
17             X1 when "01",
18             X2 when "10",
19             X3 when "11",
20             "00" when others;
21
22 end dataflow;
```

Demultiplexer 1:4

Il demultiplexer 1:4 è stato realizzato secondo un approccio dataflow.

```
1 entity demux_1_4 is
2     port (
3         x : in std_logic_vector(0 to 1);
4         s : in std_logic_vector(0 to 1);
5         y0 : out std_logic_vector(0 to 1);
6         y1 : out std_logic_vector(0 to 1);
7         y2 : out std_logic_vector(0 to 1);
8         y3 : out std_logic_vector(0 to 1)
9     );
10 end demux_1_4;
11
12 architecture dataflow of demux_1_4 is
13
```

```
14 begin
15
16     y0 <= x when s = "00" else (others => '-');
17     y1 <= x when s = "01" else (others => '-');
18     y2 <= x when s = "10" else (others => '-');
19     y3 <= x when s = "11" else (others => '-');
20 end architecture dataflow;
```

Arbitro di priorità

L'arbitro di priorità è stato anch'esso realizzato mediante un approccio dataflow.

```
1 entity arbitro is
2     port(
3         enable00 : in std_logic;
4         enable01 : in std_logic;
5         enable10 : in std_logic;
6         enable11 : in std_logic;
7         y        : out std_logic_vector(0 to 1)
8     );
9 end arbitro;
10
11 architecture Dataflow of arbitro is
12
13 begin
14     y <= "00" when enable00 = '1' else
15         "01" when enable01 = '1' else
16             "10" when enable10 = '1' else
```

```

17      "11" when enable11 = '1' else
18      "___";
19 end Dataflow;

```

Implementazione unità di controllo

Di seguito è riportato lo schematic e l'implementazione in VHDL dell'unità di controllo.

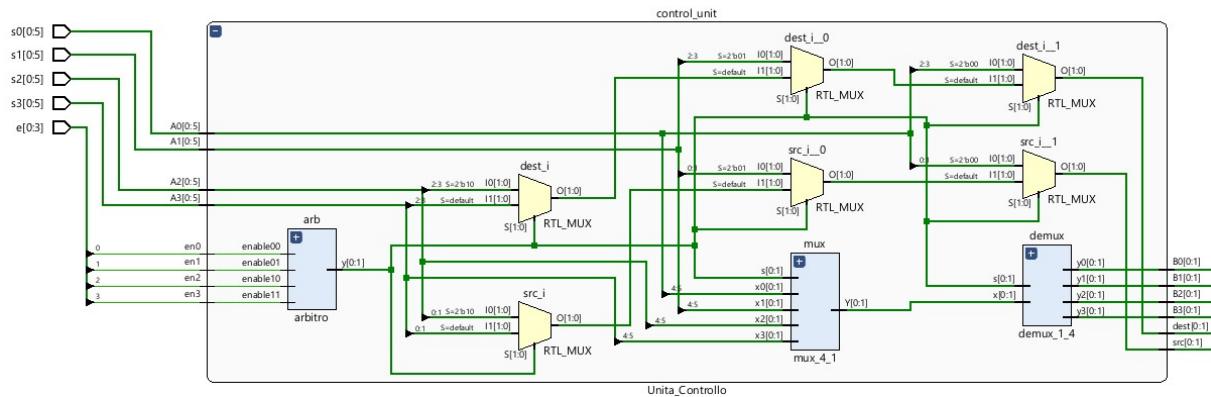


Figura 6.5: Schematic unità di controllo

```

1 entity Unita_Controllo is
2 Port (
3     A0 : in std_logic_vector(0 to 5);
4     A1 : in std_logic_vector(0 to 5);
5     A2 : in std_logic_vector(0 to 5);
6     A3 : in std_logic_vector(0 to 5);
7     en0 : in std_logic;
8     en1 : in std_logic;
9     en2 : in std_logic;
10    en3 : in std_logic;

```

```
11      B0  : out std_logic_vector(0 to 1);
12      B1  : out std_logic_vector(0 to 1);
13      B2  : out std_logic_vector(0 to 1);
14      B3  : out std_logic_vector(0 to 1);
15      src : out std_logic_vector(0 to 1);
16      dest : out std_logic_vector(0 to 1)
17  );
18 end Unita_Controllo;
19
20 architecture structural of Unita_Controllo is
21
22 component mux_4_1 is
23   port(
24     x0 : in std_logic_vector(0 to 1);
25     x1 : in std_logic_vector(0 to 1);
26     x2 : in std_logic_vector(0 to 1);
27     x3 : in std_logic_vector(0 to 1);
28     s  : in std_logic_vector(0 to 1);
29     Y  : out std_logic_vector(0 to 1)
30  );
31 end component;
32
33 component demux_1_4 is
34   port (
35     x  : in std_logic_vector(0 to 1);
36     s  : in std_logic_vector(0 to 1);
37     y0 : out std_logic_vector(0 to 1);
38     y1 : out std_logic_vector(0 to 1);
```

```
39      y2 : out std_logic_vector(0 to 1);
40
41      y3 : out std_logic_vector(0 to 1)
42  );
43
44 component arbitro is
45
46   port(
47     enable00 : in std_logic;
48     enable01 : in std_logic;
49     enable10 : in std_logic;
50     enable11 : in std_logic;
51     y        : out std_logic_vector(0 to 1)
52  );
53
54 end component;
55
56 signal t_sel: std_logic_vector(0 to 1);
57 signal t_y: std_logic_vector(0 to 1);
58
59 begin
60   mux: mux_4_1
61   --DATO input
62   port map(
63     x0 => A0(4 to 5),
64     x1 => A1(4 to 5),
65     x2 => A2(4 to 5),
66     x3 => A3(4 to 5),
67     s => t_sel,
68     Y => t_y
69  );
70
```

```

67
68     demux: demux_1_4
69     --B DATO output
70     port map(
71         x => t_y,
72         s => t_sel,
73         y0 => B0,
74         y1 => B1,
75         y2 => B2,
76         y3 => B3
77     );
78
79     arb: arbitro
80     port map(
81         enable00 => en0,
82         enable01 => en1,
83         enable10 => en2,
84         enable11 => en3,
85         y      => t_sel
86     );
87
88     src  <= A0(0 to 1) when t_sel = "00" else
89             A1(0 to 1) when t_sel = "01" else
90             A2(0 to 1) when t_sel = "10" else
91             A3(0 to 1) when t_sel = "11" else
92             "--";
93
94     dest <= A0(2 to 3) when t_sel = "00" else

```

```

95      A1(2 to 3) when t_sel = "01" else
96      A2(2 to 3) when t_sel = "10" else
97      A3(2 to 3) when t_sel = "11" else
98      "__";
99
100 end structural;

```

6.2.4 Testing del sistema complessivo

Di seguito è riportato il testbench del sistema complessivo.

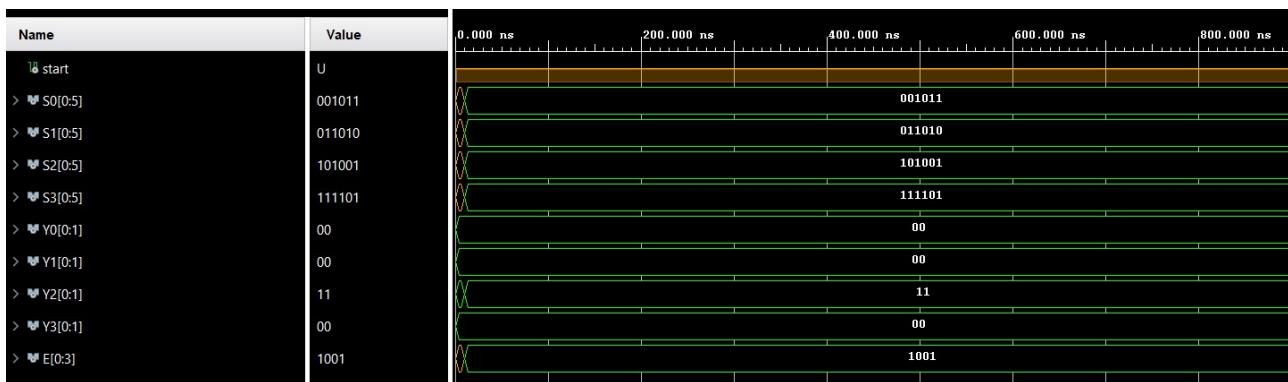


Figura 6.6: Testbench del sistema complessivo

Notiamo che, quando vogliono trasmettere la prima e la quarta linea (1001), sarà la prima linea a trasmettere il dato 11 al rispettivo destinatario (10, la terza linea), grazie alla rete di priorità dell'unità di controllo, che assegna una maggiore priorità alla prima linea.

Capitolo 7

Macchine aritmetiche

7.1 Traccia

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

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

7.2 Esercizio 11 – Moltiplicatore di Robertson

7.2.1 Introduzione

I moltiplicatori sono delle macchine fondamentali nello sviluppo di sistemi digitali in quanto utilizzati in tantissimi contesti differenti.

Distinguiamo due grandi categorie di moltiplicatori:

- moltiplicatori paralleli;
- moltiplicatori seriali.

Nei moltiplicatori paralleli, si ha una replicazione spaziale dell'operazione somma per costruire l'operazione di moltiplicazione.

Nei moltiplicatori seriali, si effettuano operazioni identiche in tutte le iterazioni, attraverso un algoritmo, realizzando quindi una replicazione temporale.

In sostanza, le operazioni aritmetiche possono essere implementate attraverso due approcci distinti:

- combinatorio: realizziamo la macchina aritmetica come macchina combinatoria, ottenendo ottime prestazioni, a discapito del notevole costo dei componenti utilizzati;
- sequenziale: si realizza la macchina aritmetica decomponendola in unità operativa e unità di controllo. Quest'ultima forni-

sce all'unità operativa tutti i segnali di controllo necessari alla realizzazione dell'operazione aritmetica.

Per il nostro progetto, abbiamo deciso di implementare il moltiplicatore di Robertson.

Il moltiplicatore di Robertson è un moltiplicatore seriale, realizzato attraverso un approccio sequenziale, seguendo un algoritmo di calcolo ispirato al calcolo manuale. Normalmente, dati due operandi, detti moltiplicatore e moltiplicando, andiamo a moltiplicare ogni singolo bit del moltiplicatore per il moltiplicando. Successivamente, andiamo a sommare correttamente i risultati pesati. Questo approccio può essere espresso in formule come:

$$P_0 = 0$$

$$P_{i+1} = P_i + x_i 2^i Y$$

In sostanza, andiamo a shiftare di una posizione variabile verso sinistra. Il prodotto tra Y e x può essere 0 o 1, in funzione del valore di x .

Tuttavia, modificando l'ordine delle operazioni, è possibile ottenere una procedura che permette di shiftare sempre di un bit verso destra, semplificando il circuito.

$$P_{i+1} = (P_i + x_i Y) 2^{-1}$$

Questo algoritmo può essere esteso per gestire anche la moltiplicazione con numeri negativi. Per farlo, si sfruttano la proprietà della rappresentazione in complementi a due che prevede che il bit più significativo ha un peso negativo.

$$X = 2^0x_0 + 2^1x_1 + \cdots + 2^{n-2}x_{n-2} - 2^{n-1}x_{n-1}$$

E' quindi possibile effettuare una moltiplicazione tra numeri relativi moltiplicando le prime n-1 cifre ignorando il segno e procedendo con un passo di correzione in cui si effettua una sottrazione all'ultimo step di calcolo.

Per la progettazione, abbiamo suddiviso il moltiplicatore in parte operativa e parte di controllo.

7.2.2 Unità operativa

L'unità operativa è stata realizzata mediante un approccio strutturale.

Registro

Il registro è stato realizzato con un approccio comportamentale. Il componente ha lo scopo di memorizzare il moltiplicando.

```
1 entity registro is
2   port (
3     clk : in std_logic;
4     reset : in std_logic;
5     load : in std_logic;
```

```
6     parallel_in : in std_logic_vector(7 downto 0);
7     parallel_out : out std_logic_vector(7 downto 0)
8 );
9 end registro;
10
11 architecture Behavioral of registro is
12
13 signal state : std_logic_vector(7 downto 0);
14
15 begin
16     parallel_out <= state;
17
18     process(clk) begin
19         if(rising_edge(clk)) then
20             if(reset = '1') then
21                 state <= (others => '0');
22             elsif(load = '1') then
23                 state <= parallel_in;
24             end if;
25         end if;
26     end process;
27 end Behavioral;
```

Sommatore

Il sommatore è un ripple carry adder, realizzato strutturalmente a partire da full adder. Per l'implementazione e la spiegazione del componente vedere gli esercizi precedenti. Notiamo che il carry in in-

gresso permette, insieme al segnale subtract proveniente dall'unità di controllo, di effettuare una sottrazione, per eseguire il passo di correzione.

Contatore

Il contatore serve all'unità di controllo per capire in che fase si trova. In particolare, l'algoritmo di calcolo verrà ripetuto tante volte quante sono le cifre del moltiplicatore.

```
1 entity counter is
2     generic(
3         M : integer := 8
4     );
5     port(
6         clk, rst, count, load : in std_logic;
7         parallel_input : in std_logic_vector(integer(ceil(log2(real(M
8             ))))-1 downto 0);
9         clk_out : out std_logic;
10        Y : out std_logic_vector(integer(ceil(log2(real(M))))-1
11            downto 0)
12    );
13 end counter;
14
15 architecture Behavioral of counter is
16
17 signal TY : std_logic_vector(integer(ceil(log2(real(M))))-1 downto 0)
18 ;
```

```
17 begin
18     Y <= TY;
19
20     output : process(TY) begin
21         if(conv_integer(TY) = M-1) then
22             clk_out <= '1';
23         else
24             clk_out <= '0';
25         end if;
26     end process;
27
28     process(clk) begin
29         if(rising_edge(clk)) then
30             if(rst = '1') then
31                 TY <= (others => '0');
32             elsif(load = '1') then
33                 TY <= parallel_input;
34             elsif(count = '1') then
35                 if(conv_integer(TY) >= M-1) then
36                     TY <= (others => '0');
37                 else
38                     TY <= TY + "1";
39                 end if;
40             end if;
41         end if;
42     end process;
43 end Behavioral;
```

Multiplexer

Il multiplexer ha lo scopo di inviare all'adder il valore corretto da sommare. Il valore può essere il moltiplicando se il bit del moltiplicatore che stiamo considerando è 1 oppure può essere 0 se il bit del moltiplicatore che stiamo considerando è 0.

```
1 entity mux is
2   port (
3     a : in std_logic_vector(7 downto 0);
4     b : in std_logic_vector(7 downto 0);
5     s : in std_logic;
6     out_d : out std_logic_vector(7 downto 0)
7   );
8 end mux;
9
10 architecture Dataflow of mux is
11 begin
12
13   with s select
14     out_d <= a when '0',
15       b when '1',
16       (others => '0') when others;
17
18 end Dataflow;
```

Flip flop

Il flip flop, realizzato mediante un approccio comportamentale, viene utilizzato per gestire il segno nelle operazioni. In particolare, se il bit più significativo del moltiplicando è 1 e voglio moltiplicare per una cifra alta, allora il flip flop conterrà 1, in modo tale che ogni volta che faccio lo shift viene introdotto un 1.

```
1 entity ff_sign is
2   port(
3     q0 : in std_logic;
4     m7 : in std_logic;
5     clk : in std_logic;
6     rst : in std_logic;
7     data : out std_logic
8   );
9 end ff_sign;
10
11 architecture Behavioral of ff_sign is
12
13 signal state : std_logic := '0';
14
15 begin
16   data <= state;
17   process(clk) begin
18     if(rising_edge(clk)) then
19       if(rst = '1') then
20         state <= '0';
21       else
```

```
22         state <= (q0 and m7) or state;
23
24     end if;
25
26 end process;
27
28 end Behavioral;
```

Shift register

Nel nostro progetto abbiamo realizzato due shift register, A e Q, rispettivamente usati come addendo dell'adder e come contenitore dei valori "shiftati" ad ogni passo da A (anche se Q è inizializzato con il valore del moltiplicatore).

Il segnale di shift è fornito dall'unità di controllo.

```
1 entity shift_register is
2
3     port(
4         clk : in std_logic;
5         shift : in std_logic;
6         reset : in std_logic;
7         load : in std_logic;
8         serial_in : in std_logic;
9         parallel_in : in std_logic_vector(7 downto 0);
10        serial_out : out std_logic;
11        parallel_out : out std_logic_vector(7 downto 0)
12    );
13
14 end shift_register;
```

```
15
16 architecture Behavioral of shift_register is
17
18     type state_type is record
19         q0 : std_logic;
20         m7 : std_logic;
21         state : std_logic;
22     end record;
23
24     variable state : state_type;
25
26 begin
27
28     process(clk, reset, load, shift, serial_in, parallel_in)
29     begin
30
31         if reset = '1' then
32             state.q0 := '0';
33             state.m7 := '0';
34             state.state := '0';
35         elsif rising_edge(clk) then
36             if load = '1' then
37                 state.q0 := serial_in;
38                 state.m7 := parallel_in;
39                 state.state := '0';
40             else
41                 state.q0 := (state.state and m7) or state.m7;
42                 state.m7 := state.q0;
43                 state.state := shift;
44             end if;
45         end if;
46     end process;
47
48     serial_out <= state.q0;
49     parallel_out <= state.m7;
50
51 end Behavioral;
```

```
15
16 signal state : std_logic_vector(7 downto 0);
17
18 begin
19     parallel_out <= state;
20     serial_out <= state(0);
21
22 process(clk) begin
23     if(rising_edge(clk)) then
24         if(reset = '1') then
25             state <= (others => '0');
26         elsif(shift = '1') then
27             state <= serial_in & state(7 downto 1);
28         elsif(load = '1') then
29             state <= parallel_in;
30         else
31             state <= state;
32         end if;
33     end if;
34 end process;
35 end Behavioral;
```

Implementazione dell'unità operativa

L'unità operativa presenta due shift register, A e Q. Inizialmente, A contiene proprio il moltiplicatore, e successivamente verrà shiftato di una posizione a destra ad ogni passo. I valori shiftati non devono essere persi, poiché sono necessari per la costruzione del risultato finale.

Pertanto, i due shift register saranno collegati in cascata, con Q che conterrà man mano i valori "buttati fuori" dallo shift register A. Ad ogni shift, il nuovo valore introdotto in A dipenderà dall'uscita del flip flop, che lo determina in base al segno del bit più significativo di M. A conterrà man mano l'operando dell'adder (quindi i risultati delle somme parziali).

Poiché Q è vuoto all'inizio, può essere utilizzato per memorizzare inizialmente il moltiplicatore. Inoltre, ad ogni passo mi interessa solo un singolo bit del moltiplicatore. Questa osservazione mi permette di non utilizzare un registro apposito per salvare il valore del moltiplicatore.

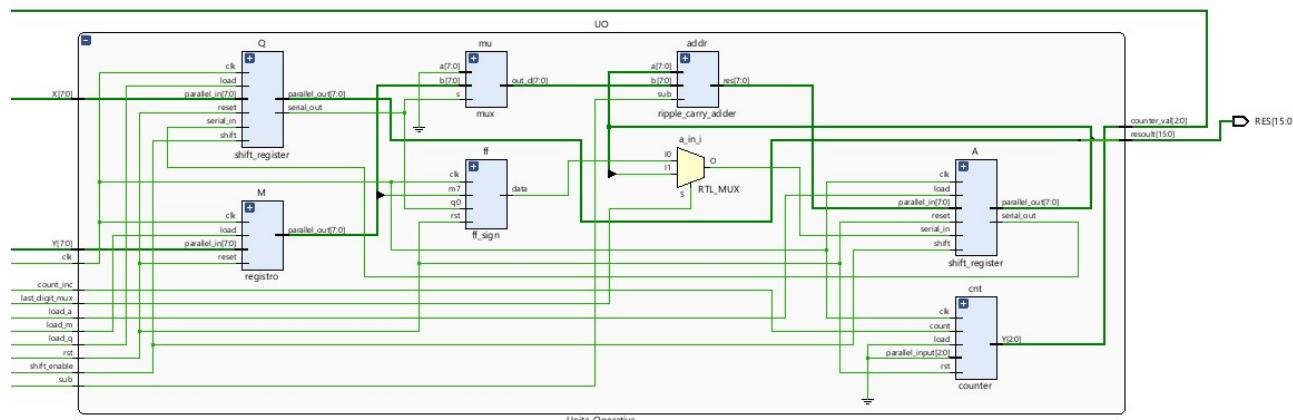


Figura 7.1: Schematic unità operativa

Di seguito è riportata l'implementazione in VHDL.

```

1  entity Unità_Operativa is
2
3      port(
4          clk           : in std_logic;
5          rst           : in std_logic;
6          x             : in std_logic_vector(7 downto 0);
7
8          y             : out std_logic_vector(7 downto 0);
9          res           : out std_logic_vector(15 downto 0);
10         v             : out std_logic;
11         );
12
13     end Unità_Operativa;
14
15
16  architecture Behavioral of Unità_Operativa is
17
18      type state is record
19          M : std_logic_vector(7 downto 0);
20          A : std_logic_vector(7 downto 0);
21          Q : std_logic;
22          R : std_logic;
23          Cnt : integer range 0 to 16;
24      end record state;
25
26      signal s : state;
27
28      begin
29
30          process(clk, rst)
31          begin
32              if(rst = '1') then
33                  s.M <= "00000000";
34                  s.A <= "00000000";
35                  s.Q <= '0';
36                  s.R <= '0';
37                  s.Cnt <= 0;
38              elsif(clk'event and clk = '1') then
39                  if(s.R = '1') then
40                      s.Q <= '0';
41                      s.R <= '0';
42                  else
43                      s.Q <= s.M(7);
44                      s.R <= s.M(7);
45                  end if;
46
47                  if(s.Cnt = 15) then
48                      s.res <= s.A;
49                      s.v <= '1';
50                  else
51                      s.v <= '0';
52                  end if;
53
54                  if(s.Q = '1') then
55                      s.A <= s.A + s.M;
56                  end if;
57
58                  if(s.Q = '0') then
59                      s.A <= s.A - s.M;
60                  end if;
61
62                  if(s.Q = '0') then
63                      s.M <= s.M(6 downto 0) & "0";
64                  end if;
65
66                  if(s.Q = '1') then
67                      s.M <= s.M(6 downto 0) & "1";
68                  end if;
69
70                  if(s.Q = '0') then
71                      s.M <= s.M(6 downto 0) & "0";
72                  end if;
73
74                  if(s.Q = '1') then
75                      s.M <= s.M(6 downto 0) & "1";
76                  end if;
77
78                  if(s.Q = '0') then
79                      s.M <= s.M(6 downto 0) & "0";
80                  end if;
81
82                  if(s.Q = '1') then
83                      s.M <= s.M(6 downto 0) & "1";
84                  end if;
85
86                  if(s.Q = '0') then
87                      s.M <= s.M(6 downto 0) & "0";
88                  end if;
89
90                  if(s.Q = '1') then
91                      s.M <= s.M(6 downto 0) & "1";
92                  end if;
93
94                  if(s.Q = '0') then
95                      s.M <= s.M(6 downto 0) & "0";
96                  end if;
97
98                  if(s.Q = '1') then
99                      s.M <= s.M(6 downto 0) & "1";
100                 end if;
101
102                 if(s.Q = '0') then
103                     s.M <= s.M(6 downto 0) & "0";
104                 end if;
105
106                 if(s.Q = '1') then
107                     s.M <= s.M(6 downto 0) & "1";
108                 end if;
109
110                 if(s.Q = '0') then
111                     s.M <= s.M(6 downto 0) & "0";
112                 end if;
113
114                 if(s.Q = '1') then
115                     s.M <= s.M(6 downto 0) & "1";
116                 end if;
117
118                 if(s.Q = '0') then
119                     s.M <= s.M(6 downto 0) & "0";
120                 end if;
121
122                 if(s.Q = '1') then
123                     s.M <= s.M(6 downto 0) & "1";
124                 end if;
125
126                 if(s.Q = '0') then
127                     s.M <= s.M(6 downto 0) & "0";
128                 end if;
129
130                 if(s.Q = '1') then
131                     s.M <= s.M(6 downto 0) & "1";
132                 end if;
133
134                 if(s.Q = '0') then
135                     s.M <= s.M(6 downto 0) & "0";
136                 end if;
137
138                 if(s.Q = '1') then
139                     s.M <= s.M(6 downto 0) & "1";
140                 end if;
141
142                 if(s.Q = '0') then
143                     s.M <= s.M(6 downto 0) & "0";
144                 end if;
145
146                 if(s.Q = '1') then
147                     s.M <= s.M(6 downto 0) & "1";
148                 end if;
149
150                 if(s.Q = '0') then
151                     s.M <= s.M(6 downto 0) & "0";
152                 end if;
153
154                 if(s.Q = '1') then
155                     s.M <= s.M(6 downto 0) & "1";
156                 end if;
157
158                 if(s.Q = '0') then
159                     s.M <= s.M(6 downto 0) & "0";
160                 end if;
161
162                 if(s.Q = '1') then
163                     s.M <= s.M(6 downto 0) & "1";
164                 end if;
165
166                 if(s.Q = '0') then
167                     s.M <= s.M(6 downto 0) & "0";
168                 end if;
169
170                 if(s.Q = '1') then
171                     s.M <= s.M(6 downto 0) & "1";
172                 end if;
173
174                 if(s.Q = '0') then
175                     s.M <= s.M(6 downto 0) & "0";
176                 end if;
177
178                 if(s.Q = '1') then
179                     s.M <= s.M(6 downto 0) & "1";
180                 end if;
181
182                 if(s.Q = '0') then
183                     s.M <= s.M(6 downto 0) & "0";
184                 end if;
185
186                 if(s.Q = '1') then
187                     s.M <= s.M(6 downto 0) & "1";
188                 end if;
189
190                 if(s.Q = '0') then
191                     s.M <= s.M(6 downto 0) & "0";
192                 end if;
193
194                 if(s.Q = '1') then
195                     s.M <= s.M(6 downto 0) & "1";
196                 end if;
197
198                 if(s.Q = '0') then
199                     s.M <= s.M(6 downto 0) & "0";
200                 end if;
201
202                 if(s.Q = '1') then
203                     s.M <= s.M(6 downto 0) & "1";
204                 end if;
205
206                 if(s.Q = '0') then
207                     s.M <= s.M(6 downto 0) & "0";
208                 end if;
209
210                 if(s.Q = '1') then
211                     s.M <= s.M(6 downto 0) & "1";
212                 end if;
213
214                 if(s.Q = '0') then
215                     s.M <= s.M(6 downto 0) & "0";
216                 end if;
217
218                 if(s.Q = '1') then
219                     s.M <= s.M(6 downto 0) & "1";
220                 end if;
221
222                 if(s.Q = '0') then
223                     s.M <= s.M(6 downto 0) & "0";
224                 end if;
225
226                 if(s.Q = '1') then
227                     s.M <= s.M(6 downto 0) & "1";
228                 end if;
229
230                 if(s.Q = '0') then
231                     s.M <= s.M(6 downto 0) & "0";
232                 end if;
233
234                 if(s.Q = '1') then
235                     s.M <= s.M(6 downto 0) & "1";
236                 end if;
237
238                 if(s.Q = '0') then
239                     s.M <= s.M(6 downto 0) & "0";
240                 end if;
241
242                 if(s.Q = '1') then
243                     s.M <= s.M(6 downto 0) & "1";
244                 end if;
245
246                 if(s.Q = '0') then
247                     s.M <= s.M(6 downto 0) & "0";
248                 end if;
249
250                 if(s.Q = '1') then
251                     s.M <= s.M(6 downto 0) & "1";
252                 end if;
253
254                 if(s.Q = '0') then
255                     s.M <= s.M(6 downto 0) & "0";
256                 end if;
257
258                 if(s.Q = '1') then
259                     s.M <= s.M(6 downto 0) & "1";
260                 end if;
261
262                 if(s.Q = '0') then
263                     s.M <= s.M(6 downto 0) & "0";
264                 end if;
265
266                 if(s.Q = '1') then
267                     s.M <= s.M(6 downto 0) & "1";
268                 end if;
269
270                 if(s.Q = '0') then
271                     s.M <= s.M(6 downto 0) & "0";
272                 end if;
273
274                 if(s.Q = '1') then
275                     s.M <= s.M(6 downto 0) & "1";
276                 end if;
277
278                 if(s.Q = '0') then
279                     s.M <= s.M(6 downto 0) & "0";
280                 end if;
281
282                 if(s.Q = '1') then
283                     s.M <= s.M(6 downto 0) & "1";
284                 end if;
285
286                 if(s.Q = '0') then
287                     s.M <= s.M(6 downto 0) & "0";
288                 end if;
289
290                 if(s.Q = '1') then
291                     s.M <= s.M(6 downto 0) & "1";
292                 end if;
293
294                 if(s.Q = '0') then
295                     s.M <= s.M(6 downto 0) & "0";
296                 end if;
297
298                 if(s.Q = '1') then
299                     s.M <= s.M(6 downto 0) & "1";
300                 end if;
301
302                 if(s.Q = '0') then
303                     s.M <= s.M(6 downto 0) & "0";
304                 end if;
305
306                 if(s.Q = '1') then
307                     s.M <= s.M(6 downto 0) & "1";
308                 end if;
309
310                 if(s.Q = '0') then
311                     s.M <= s.M(6 downto 0) & "0";
312                 end if;
313
314                 if(s.Q = '1') then
315                     s.M <= s.M(6 downto 0) & "1";
316                 end if;
317
318                 if(s.Q = '0') then
319                     s.M <= s.M(6 downto 0) & "0";
320                 end if;
321
322                 if(s.Q = '1') then
323                     s.M <= s.M(6 downto 0) & "1";
324                 end if;
325
326                 if(s.Q = '0') then
327                     s.M <= s.M(6 downto 0) & "0";
328                 end if;
329
330                 if(s.Q = '1') then
331                     s.M <= s.M(6 downto 0) & "1";
332                 end if;
333
334                 if(s.Q = '0') then
335                     s.M <= s.M(6 downto 0) & "0";
336                 end if;
337
338                 if(s.Q = '1') then
339                     s.M <= s.M(6 downto 0) & "1";
340                 end if;
341
342                 if(s.Q = '0') then
343                     s.M <= s.M(6 downto 0) & "0";
344                 end if;
345
346                 if(s.Q = '1') then
347                     s.M <= s.M(6 downto 0) & "1";
348                 end if;
349
350                 if(s.Q = '0') then
351                     s.M <= s.M(6 downto 0) & "0";
352                 end if;
353
354                 if(s.Q = '1') then
355                     s.M <= s.M(6 downto 0) & "1";
356                 end if;
357
358                 if(s.Q = '0') then
359                     s.M <= s.M(6 downto 0) & "0";
360                 end if;
361
362                 if(s.Q = '1') then
363                     s.M <= s.M(6 downto 0) & "1";
364                 end if;
365
366                 if(s.Q = '0') then
367                     s.M <= s.M(6 downto 0) & "0";
368                 end if;
369
370                 if(s.Q = '1') then
371                     s.M <= s.M(6 downto 0) & "1";
372                 end if;
373
374                 if(s.Q = '0') then
375                     s.M <= s.M(6 downto 0) & "0";
376                 end if;
377
378                 if(s.Q = '1') then
379                     s.M <= s.M(6 downto 0) & "1";
380                 end if;
381
382                 if(s.Q = '0') then
383                     s.M <= s.M(6 downto 0) & "0";
384                 end if;
385
386                 if(s.Q = '1') then
387                     s.M <= s.M(6 downto 0) & "1";
388                 end if;
389
390                 if(s.Q = '0') then
391                     s.M <= s.M(6 downto 0) & "0";
392                 end if;
393
394                 if(s.Q = '1') then
395                     s.M <= s.M(6 downto 0) & "1";
396                 end if;
397
398                 if(s.Q = '0') then
399                     s.M <= s.M(6 downto 0) & "0";
400                 end if;
401
402                 if(s.Q = '1') then
403                     s.M <= s.M(6 downto 0) & "1";
404                 end if;
405
406                 if(s.Q = '0') then
407                     s.M <= s.M(6 downto 0) & "0";
408                 end if;
409
410                 if(s.Q = '1') then
411                     s.M <= s.M(6 downto 0) & "1";
412                 end if;
413
414                 if(s.Q = '0') then
415                     s.M <= s.M(6 downto 0) & "0";
416                 end if;
417
418                 if(s.Q = '1') then
419                     s.M <= s.M(6 downto 0) & "1";
420                 end if;
421
422                 if(s.Q = '0') then
423                     s.M <= s.M(6 downto 0) & "0";
424                 end if;
425
426                 if(s.Q = '1') then
427                     s.M <= s.M(6 downto 0) & "1";
428                 end if;
429
430                 if(s.Q = '0') then
431                     s.M <= s.M(6 downto 0) & "0";
432                 end if;
433
434                 if(s.Q = '1') then
435                     s.M <= s.M(6 downto 0) & "1";
436                 end if;
437
438                 if(s.Q = '0') then
439                     s.M <= s.M(6 downto 0) & "0";
440                 end if;
441
442                 if(s.Q = '1') then
443                     s.M <= s.M(6 downto 0) & "1";
444                 end if;
445
446                 if(s.Q = '0') then
447                     s.M <= s.M(6 downto 0) & "0";
448                 end if;
449
450                 if(s.Q = '1') then
451                     s.M <= s.M(6 downto 0) & "1";
452                 end if;
453
454                 if(s.Q = '0') then
455                     s.M <= s.M(6 downto 0) & "0";
456                 end if;
457
458                 if(s.Q = '1') then
459                     s.M <= s.M(6 downto 0) & "1";
460                 end if;
461
462                 if(s.Q = '0') then
463                     s.M <= s.M(6 downto 0) & "0";
464                 end if;
465
466                 if(s.Q = '1') then
467                     s.M <= s.M(6 downto 0) & "1";
468                 end if;
469
470                 if(s.Q = '0') then
471                     s.M <= s.M(6 downto 0) & "0";
472                 end if;
473
474                 if(s.Q = '1') then
475                     s.M <= s.M(6 downto 0) & "1";
476                 end if;
477
478                 if(s.Q = '0') then
479                     s.M <= s.M(6 downto 0) & "0";
480                 end if;
481
482                 if(s.Q = '1') then
483                     s.M <= s.M(6 downto 0) & "1";
484                 end if;
485
486                 if(s.Q = '0') then
487                     s.M <= s.M(6 downto 0) & "0";
488                 end if;
489
490                 if(s.Q = '1') then
491                     s.M <= s.M(6 downto 0) & "1";
492                 end if;
493
494                 if(s.Q = '0') then
495                     s.M <= s.M(6 downto 0) & "0";
496                 end if;
497
498                 if(s.Q = '1') then
499                     s.M <= s.M(6 downto 0) & "1";
500                 end if;
501
502                 if(s.Q = '0') then
503                     s.M <= s.M(6 downto 0) & "0";
504                 end if;
505
506                 if(s.Q = '1') then
507                     s.M <= s.M(6 downto 0) & "1";
508                 end if;
509
510                 if(s.Q = '0') then
511                     s.M <= s.M(6 downto 0) & "0";
512                 end if;
513
514                 if(s.Q = '1') then
515                     s.M <= s.M(6 downto 0) & "1";
516                 end if;
517
518                 if(s.Q = '0') then
519                     s.M <= s.M(6 downto 0) & "0";
520                 end if;
521
522                 if(s.Q = '1') then
523                     s.M <= s.M(6 downto 0) & "1";
524                 end if;
525
526                 if(s.Q = '0') then
527                     s.M <= s.M(6 downto 0) & "0";
528                 end if;
529
530                 if(s.Q = '1') then
531                     s.M <= s.M(6 downto 0) & "1";
532                 end if;
533
534                 if(s.Q = '0') then
535                     s.M <= s.M(6 downto 0) & "0";
536                 end if;
537
538                 if(s.Q = '1') then
539                     s.M <= s.M(6 downto 0) & "1";
540                 end if;
541
542                 if(s.Q = '0') then
543                     s.M <= s.M(6 downto 0) & "0";
544                 end if;
545
546                 if(s.Q = '1') then
547                     s.M <= s.M(6 downto 0) & "1";
548                 end if;
549
550                 if(s.Q = '0') then
551                     s.M <= s.M(6 downto 0) & "0";
552                 end if;
553
554                 if(s.Q = '1') then
555                     s.M <= s.M(6 downto 0) & "1";
556                 end if;
557
558                 if(s.Q = '0') then
559                     s.M <= s.M(6 downto 0) & "0";
560                 end if;
561
562                 if(s.Q = '1') then
563                     s.M <= s.M(6 downto 0) & "1";
564                 end if;
565
566                 if(s.Q = '0') then
567                     s.M <= s.M(6 downto 0) & "0";
568                 end if;
569
570                 if(s.Q = '1') then
571                     s.M <= s.M(6 downto 0) & "1";
572                 end if;
573
574                 if(s.Q = '0') then
575                     s.M <= s.M(6 downto 0) & "0";
576                 end if;
577
578                 if(s.Q = '1') then
579                     s.M <= s.M(6 downto 0) & "1";
580                 end if;
581
582                 if(s.Q = '0') then
583                     s.M <= s.M(6 downto 0) & "0";
584                 end if;
585
586                 if(s.Q = '1') then
587                     s.M <= s.M(6 downto 0) & "1";
588                 end if;
589
590                 if(s.Q = '0') then
591                     s.M <= s.M(6 downto 0) & "0";
592                 end if;
593
594                 if(s.Q = '1') then
595                     s.M <= s.M(6 downto 0) & "1";
596                 end if;
597
598                 if(s.Q = '0') then
599                     s.M <= s.M(6 downto 0) & "0";
600                 end if;
601
602                 if(s.Q = '1') then
603                     s.M <= s.M(6 downto 0) & "1";
604                 end if;
605
606                 if(s.Q = '0') then
607                     s.M <= s.M(6 downto 0) & "0";
608                 end if;
609
610                 if(s.Q = '1') then
611                     s.M <= s.M(6 downto 0) & "1";
612                 end if;
613
614                 if(s.Q = '0') then
615                     s.M <= s.M(6 downto 0) & "0";
616                 end if;
617
618                 if(s.Q = '1') then
619                     s.M <= s.M(6 downto 0) & "1";
620                 end if;
621
622                 if(s.Q = '0') then
623                     s.M <= s.M(6 downto 0) & "0";
624                 end if;
625
626                 if(s.Q = '1') then
627                     s.M <= s.M(6 downto 0) & "1";
628                 end if;
629
630                 if(s.Q = '0') then
631                     s.M <= s.M(6 downto 0) & "0";
632                 end if;
633
634                 if(s.Q = '1') then
635                     s.M <= s.M(6 downto 0) & "1";
636                 end if;
637
638                 if(s.Q = '0') then
639                     s.M <= s.M(6 downto 0) & "0";
640                 end if;
641
642                 if(s.Q = '1') then
643                     s.M <= s.M(6 downto 0) & "1";
644                 end if;
645
646                 if(s.Q = '0') then
647                     s.M <= s.M(6 downto 0) & "0";
648                 end if;
649
650                 if(s.Q = '1') then
651                     s.M <= s.M(6 downto 0) & "1";
652                 end if;
653
654                 if(s.Q = '0') then
655                     s.M <= s.M(6 downto 0) & "0";
656                 end if;
657
658                 if(s.Q = '1') then
659                     s.M <= s.M(6 downto 0) & "1";
660                 end if;
661
662                 if(s.Q = '0') then
663                     s.M <= s.M(6 downto 0) & "0";
664                 end if;
665
666                 if(s.Q = '1') then
667                     s.M <= s.M(6 downto 0) & "1";
668                 end if;
669
670                 if(s.Q = '0') then
671                     s.M <= s.M(6 downto 0) & "0";
672                 end if;
673
674                 if(s.Q = '1') then
675                     s.M <= s.M(6 downto 0) & "1";
676                 end if;
677
678                 if(s.Q = '0') then
679                     s.M <= s.M(6 downto 0) & "0";
680                 end if;
681
682                 if(s.Q = '1') then
683                     s.M <= s.M(6 downto 0) & "1";
684                 end if;
685
686                 if(s.Q = '0') then
687                     s.M <= s.M(6 downto 0) & "0";
688                 end if;
689
690                 if(s.Q = '1') then
691                     s.M <= s.M(6 downto 0) & "1";
692                 end if;
693
694                 if(s.Q = '0') then
695                     s.M <= s.M(6 downto 0) & "0";
696                 end if;
697
698                 if(s.Q = '1') then
699                     s.M <= s.M(6 downto 0) & "1";
700                 end if;
701
702                 if(s.Q = '0') then
703                     s.M <= s.M(6 downto 0) & "0";
704                 end if;
705
706                 if(s.Q = '1') then
707                     s.M <= s.M(6 downto 0) & "1";
708                 end if;
709
710                 if(s.Q = '0') then
711                     s.M <= s.M(6 downto 0) & "0";
712                 end if;
713
714                 if(s.Q = '1') then
715                     s.M <= s.M(6 downto 0) & "1";
716                 end if;
717
718                 if(s.Q = '0') then
719                     s.M <= s.M(6 downto 0) & "0";
720                 end if;
721
722                 if(s.Q = '1') then
723                     s.M <= s.M(6 downto 0) & "1";
724                 end if;
725
726                 if(s.Q = '0') then
727                     s.M <= s.M(6 downto 0) & "0";
728                 end if;
729
730                 if(s.Q = '1') then
731                     s.M <= s.M(6 downto 0) & "1";
732                 end if;
733
734                 if(s.Q = '0') then
735                     s.M <= s.M(6 downto 0) & "0";
736                 end if;
737
738                 if(s.Q = '1') then
739                     s.M <= s.M(6 downto 0) & "1";
740                 end if;
741
742                 if(s.Q = '0') then
743                     s.M <= s.M(6 downto 0) & "0";
744                 end if;
745
746                 if(s.Q = '1') then
747                     s.M <= s.M(6 downto 0) & "1";
748                 end if;
749
750                 if(s.Q = '0') then
751                     s.M <= s.M(6 downto 0) & "0";
752                 end if;
753
754                 if(s.Q = '1') then
755                     s.M <= s.M(6 downto 0) & "1";
756                 end if;
757
758                 if(s.Q = '0') then
759                     s.M <= s.M(6 downto 0) & "0";
760                 end if;
761
762                 if(s.Q = '1') then
763                     s.M <= s.M(6 downto 0) & "1";
764                 end if;
765
766                 if(s.Q = '0') then
767                     s.M <= s.M(6 downto 0) & "0";
768                 end if;
769
770                 if(s.Q = '1') then
771                     s.M <= s.M(6 downto 0) & "1";
772                 end if;
773
774                 if(s.Q = '0') then
775                     s.M <= s.M(6 downto 0) & "0";
776                 end if;
777
778                 if(s.Q = '1') then
779                     s.M <= s.M(6 downto 0) & "1";
780                 end if;
781
782                 if(s.Q = '0') then
783                     s.M <= s.M(6 downto 0) & "0";
784                 end if;
785
786                 if(s.Q = '1') then
787                     s.M <= s.M(6 downto 0) & "1";
788                 end if;
789
790                 if(s.Q = '0') then
791                     s.M <= s.M(6 downto 0) & "0";
792                 end if;
793
794                 if(s.Q = '1') then
795                     s.M <= s.M(6 downto 0) & "1";
796                 end if;
797
798                 if(s.Q = '0') then
799                     s.M <= s.M(6 downto 0) & "0";
800                 end if;
801
802                 if(s.Q = '1') then
803                     s.M <= s.M(6 downto 0) & "1";
804                 end if;
805
806                 if(s.Q = '0') then
807                     s.M <= s.M(6 downto 0) & "0";
808                 end if;
809
810                 if(s.Q = '1') then
811                     s.M <= s.M(6 downto 0) & "1";
812                 end if;
813
814                 if(s.Q = '0') then
815                     s.M <= s.M(6 downto 0) & "0";
816                 end if;
817
818                 if(s.Q = '1') then
819                     s.M <= s.M(6 downto 0) & "1";
820                 end if;
821
822                 if(s.Q = '0') then
823                     s.M <= s.M(6 downto 0) & "0";
824                 end if;
825
826                 if(s.Q = '1') then
827                     s.M <= s.M(6 downto 0) & "1";
828                 end if;
829
830                 if(s.Q = '0') then
831                     s.M <= s.M(6 downto 0) & "0";
832                 end if;
833
834                 if(s.Q = '1') then
835                     s.M <= s.M(6 downto 0) & "1";
836                 end if;
837
838                 if(s.Q = '0') then
839                     s.M <= s.M(6 downto 0) & "0";
840                 end if;
841
842                 if(s.Q = '1') then
843                     s.M <= s.M(6 downto 0) & "1";
844                 end if;
845
846                 if(s.Q = '0') then
847                     s.M <= s.M(6 downto 0) & "0";
848                 end if;
849
850                 if(s.Q = '1') then
851                     s.M <= s.M(6 downto 0) & "1";
852                 end if;
853
854                 if(s.Q = '0') then
855                     s.M <= s.M(6 downto 0) & "0";
856                 end if;
857
858                 if(s.Q = '1') then
859                     s.M <= s.M(6 downto 0) & "1";
860                 end if;
861
862                 if(s.Q = '0') then
863                     s.M <= s.M(6 downto 0) & "0";
864                 end if;
865
866                 if(s.Q = '1') then
867                     s.M <= s.M(6 downto 0) & "1";
868                 end if;
869
870                 if(s.Q = '0') then
871                     s.M <= s.M(6 downto 0) & "0";
872                 end if;
873
874                 if(s.Q = '1') then
875                     s.M <= s.M(6 downto 0) & "1";
876                 end if;
877
878                 if(s.Q = '0') then
879                     s.M <= s.M(6 downto 0) & "0";
880                 end if;
881
882                 if(s.Q = '1') then
883                     s.M <= s.M(6 downto 0) & "1";

```

```

6      Y          : in std_logic_vector(7 downto 0);
7      load_a     : in std_logic;
8      load_q     : in std_logic;
9      load_m     : in std_logic;
10     count_inc  : in std_logic;
11     shift_enable: in std_logic;
12     sub         : in std_logic;
13     last_digit_mux: in std_logic;
14     counter_val : out std_logic_vector(2 downto 0);
15     result       : out std_logic_vector(15 downto 0)
16   );
17 end Unita_Operativa;
18
19 architecture Structural of Unita_Operativa is
20
21 component ripple_carry_adder is
22   port(
23     a      : in std_logic_vector(7 downto 0);
24     b      : in std_logic_vector(7 downto 0);
25     sub   : in std_logic;
26     res   : out std_logic_vector(7 downto 0);
27     cout  : out std_logic
28   );
29 end component;
30
31 component counter is
32   generic(
33     M : integer := 8

```

```
34 );
35 port(
36     clk : in std_logic;
37     rst : in std_logic;
38     count : in std_logic;
39     load : in std_logic;
40     parallel_input : in std_logic_vector(integer(ceil(log2(real(M
41         ))))-1 downto 0);
42     clk_out : out std_logic;
43     Y : out std_logic_vector(integer(ceil(log2(real(M)))-1
44         ) downto 0)
45 );
46 end component;
47
48
49
50
51
52
53
54
55
56 component ff_sign is
57     port(
58         q0 : in std_logic;
59         m7 : in std_logic;
60         clk : in std_logic;
61         rst : in std_logic;
62         data : out std_logic
63     );
64 end component;
65
66
67
68
69
70
71
72
73
74
75
76 component mux is
77     port(
78         a : in std_logic_vector(7 downto 0);
79         b : in std_logic_vector(7 downto 0);
```

```
60      s : in std_logic;
61      out_d : out std_logic_vector(7 downto 0)
62  );
63 end component;
64
65 component registro is
66   port(
67     clk : in std_logic;
68     reset : in std_logic;
69     load : in std_logic;
70     parallel_in : in std_logic_vector(7 downto 0);
71     parallel_out : out std_logic_vector(7 downto 0)
72  );
73 end component;
74
75 component shift_register is
76   port(
77     clk : in std_logic;
78     shift : in std_logic;
79     reset : in std_logic;
80     load : in std_logic;
81
82     serial_in : in std_logic;
83     parallel_in : in std_logic_vector(7 downto 0);
84
85     serial_out : out std_logic;
86     parallel_out : out std_logic_vector(7 downto 0)
87  );
```

```
88 end component;
89
90 signal ff_out : std_logic;
91 signal adder_out : std_logic_vector(7 downto 0);
92 signal a_parallel_out : std_logic_vector(7 downto 0);
93 signal a_serial_out : std_logic;
94
95 signal q_parallel_out : std_logic_vector(7 downto 0);
96 signal q_serial_out : std_logic;
97
98 signal m_parallel_out : std_logic_vector(7 downto 0);
99
100 signal mux_out : std_logic_vector(7 downto 0);
101
102 signal a_in : std_logic;
103
104 begin
105
106     result <= a_parallel_out & q_parallel_out; — concatenazione
107         registri A e Q
108
109     a_in <= ff_out when last_digit_mux = '0' else
110         a_parallel_out(7) when last_digit_mux = '1';
111
112     A : shift_register
113         port map(
114             clk => clk,
115             shift => shift_enable,
```

```
115      reset => rst,
116      load  => load_a,
117      serial_in => a_in,
118      parallel_in => adder_out,
119      parallel_out => a_parallel_out,
120      serial_out => a_serial_out
121  );
122
123 Q : shift_register
124 port map(
125     clk => clk,
126     shift => shift_enable,
127     reset => rst,
128     load => load_q,
129     serial_in => a_serial_out,
130     parallel_in => X,
131     parallel_out => q_parallel_out,
132     serial_out => q_serial_out
133 );
134
135 M : registro
136 port map(
137     clk => clk,
138     reset => rst,
139     load => load_m,
140     parallel_in => Y,
141     parallel_out => m_parallel_out
142 );
```

```
143
144     addr : ripple_carry_adder
145     port map(
146         a => a_parallel_out,
147         b => mux_out,
148         sub => sub,
149         res => adder_out
150     );
151
152     cnt : counter generic map(
153         M => 8
154     ) port map(
155         clk => clk,
156         rst => rst,
157         count => count_inc,
158         load => '0',
159         y => counter_val,
160         parallel_input => (others => '0')
161     );
162
163     mu : mux port map(
164         b => m_parallel_out,
165         a => (others=>'0'),
166         s => q_serial_out,
167         out_d => mux_out
168     );
169
170     ff : ff_sign port map(
```

```
171     q0 => q_serial_out,  
172     m7 => m_parallel_out(7),  
173     clk => clk,  
174     rst => rst,  
175     data => ff_out  
176 );  
177  
178 end Structural;
```

7.2.3 Unità di controllo

L'unità di controllo fornisce all'unità operativa tutti i segnali necessari all'implementazione dell'algoritmo, tra cui il segnale *shift_enable* per lo shift register, i *load* per il caricamento dei valori dai registri, *reset* per riportare la macchina allo stato iniziale, il segnale *count_inc* per incrementare il valore del conteggio, il segnale *subtract* per effettuare il passo di correzione.

```
1 entity Unita_Controllo is
2   port (
3     clk : in std_logic;
4     counter_val : in std_logic_vector(2 downto 0);
5     start : in std_logic;
6     reset : in std_logic;
7     load_a : out std_logic;
8     load_q : out std_logic;
9     load_m : out std_logic;
10    count_inc : out std_logic;
11    shift_enable : out std_logic;
12    sub : out std_logic;
13    last_digit_mux : out std_logic;
14    rst : out std_logic
15  );
16 end Unita_Controllo;
```

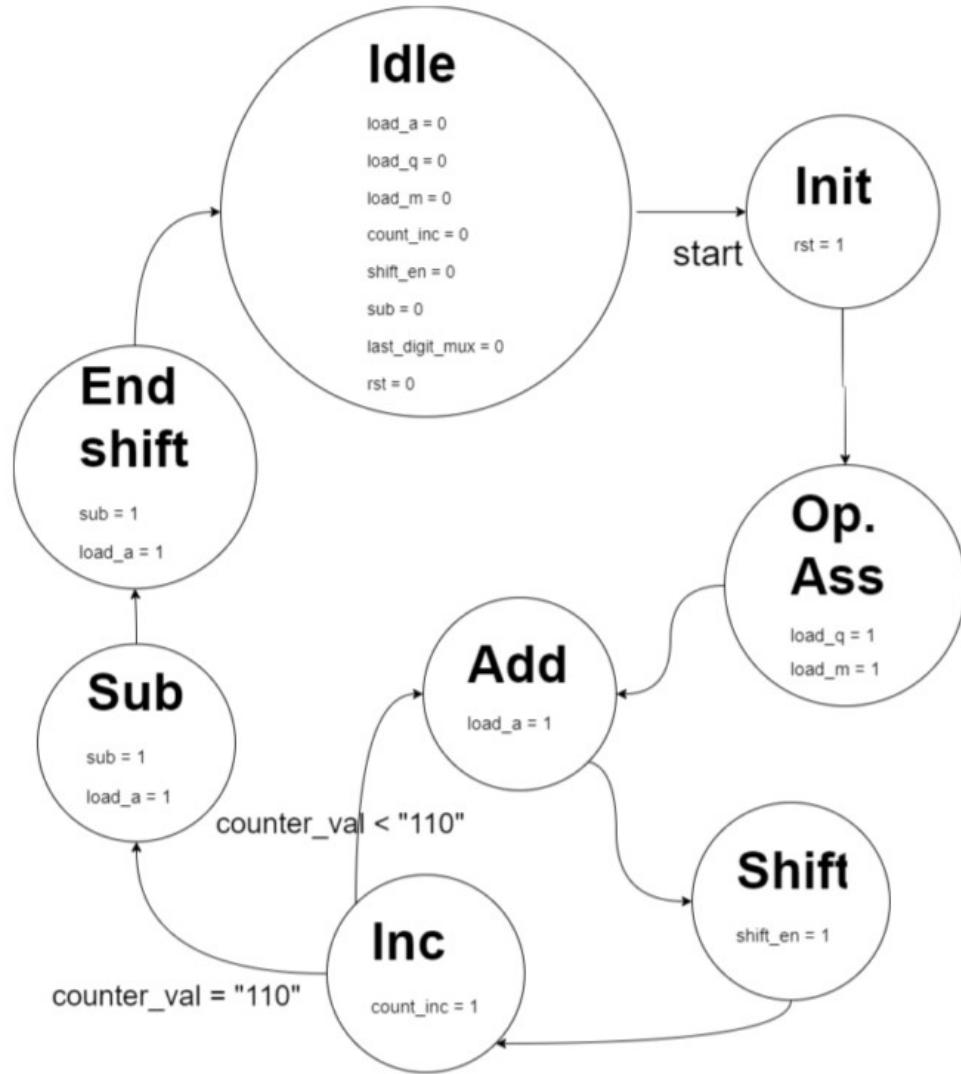


Figura 7.2: Automa unità di controllo

L'unità di controllo evolve attraverso i seguenti stati: *idle*, *init*, *operands assignment*, *add*, *shift*, *increment*, *sub* e *end_shift*. La macchina parte nello stato *idle* e vi permane fin quando non riceve un segnale di *start* dall'esterno. Una volta ricevuto transita nello stato *init*, dove il contenuto dei tre registri e del flip flop viene azzerato. Poi nella fase di *operands assignment* i registri Q ed M vengono caricati rispettivamente con il valore del moltiplicatore e del moltiplicando. A

questo punto, iniziano le somme da parte della macchina. In particolare, ogni volta che la macchina passa nello stato *add* verrà sommato il valore $x_i Y$, cioè il bit del moltiplicatore che stiamo considerando per il valore del moltiplicando. Successivamente avviene lo *shift* a destra di una posizione. La sequenza *add-shift* viene ripetuta per 7 volte, dopodiché viene effettuato il passo di correzione nello stato *sub* e nello stato *end_shift* viene alzato il segnale di selezione del multiplexer in modo da preservare il segno del risultato. Notiamo che, una volta che il contatore avrà raggiunto il valore 6, la macchina effettuerà a prescindere il passo di correzione. Tuttavia, nel caso in cui la cifra più significativa del moltiplicatore (Q_0) fosse 0, non vedrei gli effetti della sottrazione, perché sto sottraendo uno 0.

L'unità di controllo, in funzione del segno degli operandi, si ritroverà a gestire 4 casi distinti:

- Se X e Y (moltiplicatore e moltiplicando) sono entrambi positivi e facciamo una moltiplicazione, le somme sono sempre positive, opportunamente shiftate, dunque non bisogna correggere nulla;
- Se X è positivo ma Y è negativo lo sappiamo subito, ogni volta che prendiamo la Y e la moltiplichiamo per una cifra non nulla il prodotto parziale è negativo, scriviamo un 1 nel flip flop;
- Se X è negativo e Y è positivo, fino all'ultima cifra non ce ne accorgiamo, facciamo le somme, all'ultima cifra ci accorgiamo che

ha un peso negativo e dunque al posto di sommare sottraiamo, effettuando il passo di correzione;

- Se X e Y sono entrambi negativi, dall'inizio per noi il prodotto parziale era negativo, solo alla fine ci accorgiamo che in realtà è positivo per cui invece di fare la somma facciamo la sottrazione, passo di correzione.

Implementazione dell'unità di controllo

L'unità di controllo è stata progettata seguendo un approccio comportamentale, con 2 process. Un process ha il compito di processare lo stato prossimo, l'altro ha lo scopo di stabilire, per ogni stato, le uscite da alzare.

L'unità di controllo è stata realizzata in logica cablata.

```
1 architecture Behavioral of Unita_Controllo is
2
3 type stato is (idle, init, op_ass, add, shift, inc, subtract,
4   end_shift);
5
6 signal state : stato := idle;
7
8 begin
9   state_prc : process(clk) begin
10     if(rising_edge(clk)) then
11       if(reset = '1') then
12         state <= idle;
13       elsif(state = idle and start = '1') then
```

```

12         state <= init;
13
14         elsif(state = init) then
15
16             state <= op_ass;
17
18             elsif(state = op_ass) then
19
20                 state <= add;
21
22                 elsif(state = add) then
23
24                     state <= shift;
25
26                     elsif(state = shift) then
27
28                         state <= inc;
29
30                         elsif(state = inc and counter_val = "110") then
31
32                             state <= subtract;
33
34                             elsif(state = inc) then
35
36                                 state <= add;
37
38                                 elsif(state = subtract) then
39
40                                     state <= end_shift;
41
42                                     elsif(state = end_shift) then
43
44                                         state <= idle;
45
46                                         end if;
47
48                                         end if;
49
50         end process;
51
52
53         out_prc : process(state) begin
54
55             --default values
56
57             load_a <= '0';
58
59             load_q <= '0';
60
61             load_m <= '0';
62
63             count_inc <= '0';

```

```
40      shift_enable <= '0';
41      sub <= '0';
42      rst <= '0';
43      last_digit_mux <= '0';
44
45      if(state = init) then
46          rst <= '1';
47      elsif(state = op_ass) then
48          load_q <= '1';
49          load_m <= '1';
50      elsif(state = add) then
51          load_a <= '1';
52      elsif(state = shift) then
53          shift_enable <= '1';
54      elsif(state = inc) then
55          count_inc <= '1';
56      elsif(state = subtract) then
57          sub <= '1';
58          load_a <= '1';
59      elsif(state = end_shift) then
60          shift_enable <= '1';
61          last_digit_mux <= '1';
62      end if;
63
64  end process;
65 end Behavioral;
```

7.3 Sistema complessivo

Di seguito è riportato lo schematic del sistema complessivo, l'implementazione in VHDL e il testbench per verificare la correttezza del funzionamento del sistema.

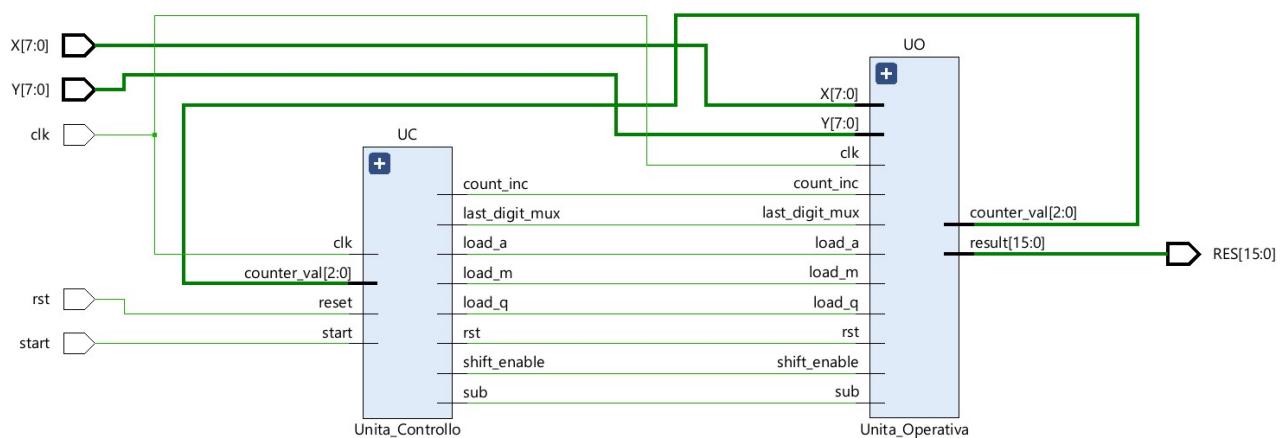


Figura 7.3: Schematic sistema complessivo

```

1 entity moltiplicatore_robertson is
2   port(
3     X : in std_logic_vector(7 downto 0);
4     Y : in std_logic_vector(7 downto 0);
5     rst : in std_logic;
6     start : in std_logic;
7     clk : in std_logic;
8     RES : out std_logic_vector(15 downto 0)
9   );
10 end moltiplicatore_robertson;
11
12 architecture structural of moltiplicatore_robertson is
13

```

```
14 component Unita_Operativa is
15   port(
16     clk : in std_logic;
17
18     X : in std_logic_vector(7 downto 0);
19     Y : in std_logic_vector(7 downto 0);
20
21     load_a : in std_logic;
22     load_q : in std_logic;
23     load_m : in std_logic;
24     count_inc : in std_logic;
25     shift_enable : in std_logic;
26     sub : in std_logic;
27     rst : in std_logic;
28     last_digit_mux : in std_logic;
29
30     counter_val : out std_logic_vector(2 downto 0);
31     result : out std_logic_vector(15 downto 0)
32   );
33 end component;
34
35 component Unita_Controllo is
36   port(
37     clk : in std_logic;
38     counter_val : in std_logic_vector(2 downto 0);
39     start : in std_logic;
40     reset : in std_logic;
41
```

```
42      load_a : out std_logic;
43      load_q : out std_logic;
44      load_m : out std_logic;
45      count_inc : out std_logic;
46      shift_enable : out std_logic;
47      sub : out std_logic;
48      rst : out std_logic;
49
50      last_digit_mux : out std_logic
51  );
52 end component;
53
54 signal counter_val : std_logic_vector(2 downto 0);
55 signal load_a : std_logic;
56 signal load_q : std_logic;
57 signal load_m : std_logic;
58 signal count_inc : std_logic;
59 signal shift_enable : std_logic;
60 signal sub : std_logic;
61 signal reset_datapath : std_logic;
62 signal last_digit_mux : std_logic;
63
64 begin
65
66   UC : Unita_Controllo port map(
67     clk => clk,
68     counter_val => counter_val,
69     start => start,
```

```
70      reset => rst,
71
72      sub => sub,
73
74      load_a => load_a,
75
76      load_q => load_q,
77
78      load_m => load_m,
79
80      count_inc => count_inc,
81
82      shift_enable => shift_enable,
83
84      rst => reset_datapath,
85
86      last_digit_mux => last_digit_mux
87
88  );
89
90
91  UO : Unita_Operativa port map(
92
93      clk => clk,
94
95      X => X,
96
97      Y => Y,
98
99      load_a => load_a,
100
101      load_q => load_q,
102
103      load_m => load_m,
104
105      count_inc => count_inc,
106
107      shift_enable => shift_enable,
108
109      sub => sub,
110
111      rst => reset_datapath,
112
113      counter_val => counter_val,
114
115      result => res,
116
117      last_digit_mux => last_digit_mux
118
119  );
120
121
122  end structural;
```

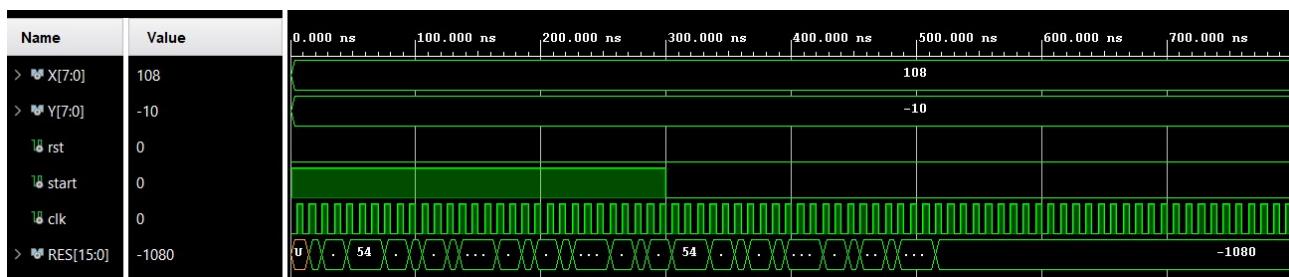


Figura 7.4: Test del sistema complessivo

Capitolo 8

Esercizio libero

8.1 Traccia

Progettare, implementare in VHDL e simulare la seguente architettura
Un nodo A è alimentato da 2 ROM di N byte. Il nodo A trasmette
al nodo B il valore ottenuto sommando gli elementi delle due ROM
in posizioni omologhe. Il nodo B a sua volta è dotato di due moduli
di memoria MEM1 e MEM2: i byte ricevuti da A sono memorizzati
in MEM1 se positivi, mentre vengono memorizzati in MEM2 se nul-
li o negativi. Progettare il sistema utilizzando in ciascun nodo un
componente contatore e un componente demultiplexer e inserendo un
sommatore strutturale in A.

8.2 Esercizio 12

Per la realizzazione del progetto abbiamo individuato, per entrambe le entità, una parte operativa e una parte di controllo. La macchina da realizzare è un sistema in grado di sommare due valori caricati da due ROM differenti e stabilire se il risultato è un numero negativo o positivo. In funzione del segno, questo valore verrà memorizzato in una determinata memoria piuttosto che in un'altra. Il risultato dell'addizione, verrà inviato all'entità B tramite un protocollo di handshaking.

8.2.1 Unità operativa – entità A

L'unità operativa è stata realizzata mediante un approccio strutturale.

Contatore

Il contatore è stato realizzato mediante un approccio comportamentale. Il componente ha lo scopo di indirizzare le due memorie ROM presenti nell'entità A che contengono i valori da sommare, che andranno dunque in ingresso al sommatore. La scelta di avere *un unico contatore per indirizzare entrambe le memorie* è stata dettata dal fatto che la somma avviene tra *valori memorizzati in posizioni omologhe*.

```
1 entity contatore is
2   Port(
3     clock : in std_logic;
4     reset : in std_logic;
```

```
5      enable : in std_logic;
6      counter : out std_logic_vector(0 to 2);
7      uscita : out std_logic
8  );
9 end contatore;
10
11 architecture Behavioral of contatore is
12 begin
13
14 process(clock)
15 variable count : integer := 0;
16 begin
17   if reset = '1' then
18     count := 0;
19     uscita <= '0';
20   end if;
21   if falling_edge(clock) then
22     if enable = '1' then
23       if count = 7 then
24         count := 0;
25         uscita <= '1';
26       else
27         count := count + 1;
28         uscita <= '0';
29       end if;
30     end if;
31   end if;
32   counter <= std_logic_vector(to_unsigned(count, 3));

```

```
33 end process;  
34  
35 end Behavioral;
```

ROM

Le memorie ROM, realizzate in modo comportamentale, *contengono i valori da sommare*.

```
1 entity ROM_in is  
2   port(  
3     clock      : in  std_logic;  
4     reset       : in  std_logic;  
5     read        : in  std_logic;  
6     addr        : in  std_logic_vector(0 to 2);  
7     value_out   : out std_logic_vector(0 to 3)  
8   );  
9 end ROM_in;  
10  
11 architecture Behavioral of ROM_in is  
12  
13   type rom_type is array (0 to 7) of std_logic_vector(0 to 3);  
14   type init_rom is array (0 to 7) of std_logic_vector(0 to 3);  
15  
16   signal ROM : rom_type;  
17   signal value : init_rom := (  
18     "0000", "1001", "1100", "1101", "0011", "0111", "1111", "1010"  
19   );
```

```

20
21 begin
22
23 process(clock)
24 begin
25   if rising_edge(clock) then
26     init: for i in 0 to 7 loop
27       ROM(i) <= value(i);
28     end loop init;
29     if(reset = '1') then
30       value_out <= ROM(0);
31     elsif(READ = '1') then
32       value_out <= ROM(to_integer(unsigned(addr)));
33     end if;
34   end if;
35 end process;
36 end Behavioral;

```

Sommatore

Il sommatore è stato realizzato in modo strutturale. Abbiamo costruito un ripple carry adder a partire dai full adder. Il full adder è un'estensione dell'half adder, in quanto permette di *gestire anche eventuali riporti in ingresso*, oltre che in uscita (per la realizzazione del nostro progetto lo abbiamo supposto a zero). Il full adder presenta 3 ingressi, due bit e un riporto entrante, e ha due bit in uscita, sempre di somma e riporto.

X	Y	C	S	R
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figura 8.1: Tabella di verità del full adder

Dalla tabella, si possono evincere le espressioni booleane della somma e del riporto:

$$S = X \text{ xor } Y \text{ xor } C$$

$$R = X \text{ and } Y + C \text{ and } (X \text{ xor } Y)$$

L'espressione del riporto è piuttosto semplice da ricordare in quanto un riporto viene generato se *entrambi gli addendi sono alti oppure se c'è un riporto in ingresso e uno dei due addendi è alto*.

Nel nostro esercizio, abbiamo realizzato il full adder con un approccio dataflow, quindi a livello di porte logiche, sfruttando le espressione booleane della somma e del riporto in uscita, ottenuti dalla tabella di verità.

```
1 entity FullAdder is
2   Port (
3     a          :  in  std_logic;
4     b          :  in  std_logic;
5     carry_in  :  in  std_logic;
```

```

6      carry_out :  out std_logic;
7      ris        :  out std_logic
8  );
9 end FullAdder;
10
11 architecture Dataflow of FullAdder is
12 begin
13     ris      <= ( (a xor b) xor carry_in);
14     carry_out <= ( (a and b) or (carry_in and (a xor b)));
15 end Dataflow;

```

Il ripple carry adder è un *sommatore a propagazione di riporto*. Viene realizzato *concatenando vari full adder*, ognuno dei quali realizza la somma di una coppia di bit omologhi. In generale, il primo sommatore, associato al bit meno significativo delle due stringhe, potrebbe anche essere un half adder, in quanto non abbiamo un riporto entrante. Tuttavia, per ottenere un'architettura omogenea, abbiamo preferito utilizzare in ogni caso un full adder, mettendo il riporto entrante a 0.

```

1 entity RippleCarryAdder is
2 Port(
3
4     a: in std_logic_vector (0 to 3);
5     b: in std_logic_vector (0 to 3);
6     c_in: in std_logic;
7     c_out: out std_logic;
8     s: out std_logic_vector (0 to 3)
9 );

```

```

10 end RippleCarryAdder;

11

12 architecture Structural of RippleCarryAdder is

13

14 component FullAdder is

15   Port (

16     a: in std_logic;

17     b: in std_logic;

18     carry_in: in std_logic;

19     carry_out: out std_logic;

20     ris: out std_logic

21   );

22 end component;

23

24 signal carry_temp: std_logic_vector(0 to 2);

25

26 begin

27   F0: FullAdder port map(a(0), b(0), c_in, carry_temp(0), s(0));

28   F1: FullAdder port map(a(1), b(1), carry_temp(0), carry_temp(1),

29     s(1));

30   F2: FullAdder port map(a(2), b(2), carry_temp(1), carry_temp(2),

31     s(2));

32   F3: FullAdder port map(a(3), b(3), carry_temp(2), c_out, s(3));

33 end Structural;

```

Notiamo dal codice VHDL che *F0*, il full adder associato al bit meno significativo prende in ingresso il riporto entrante, mentre gli altri il riporto in uscita del full adder precedente.

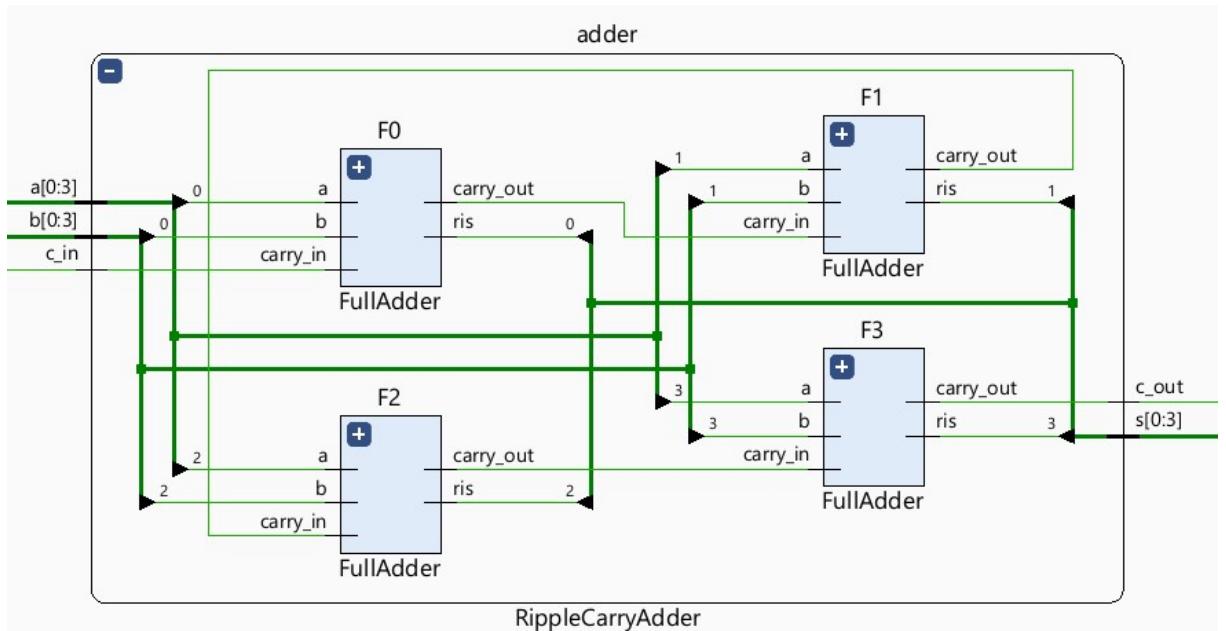


Figura 8.2: Ripple carry adder strutturale

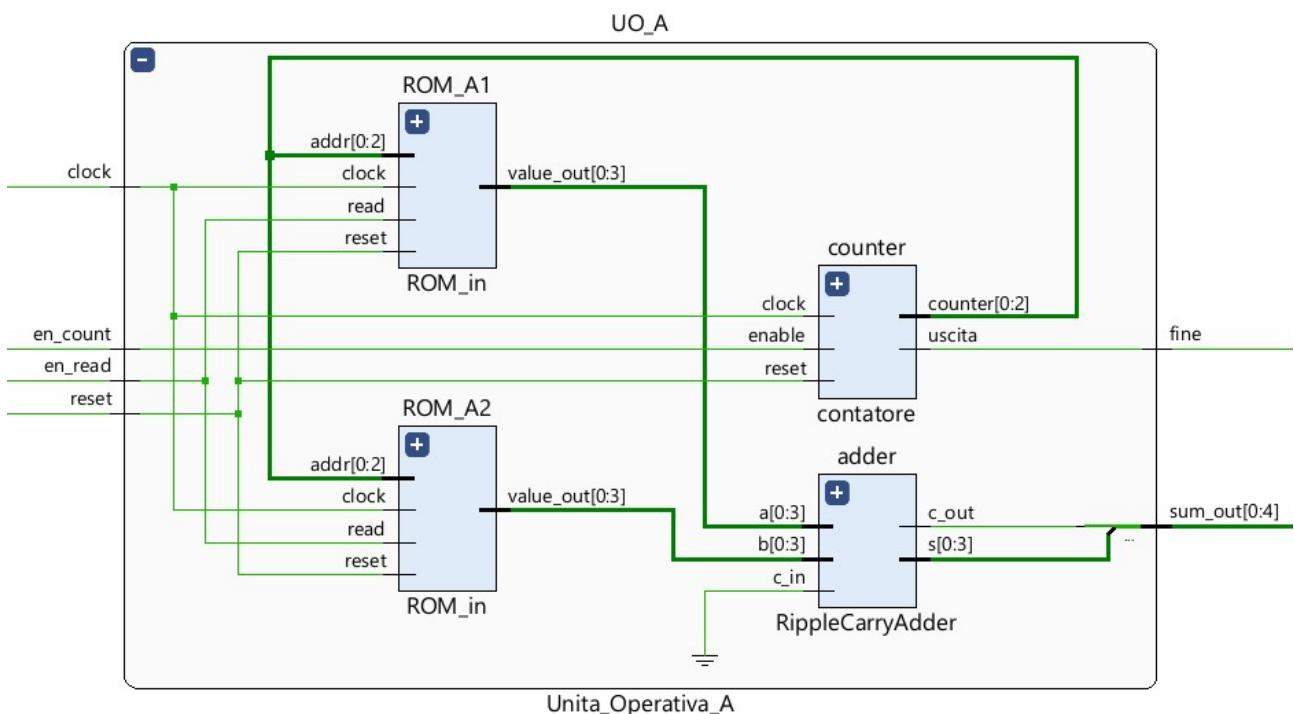


Figura 8.3: Schematic unità operativa entità A

8.2.2 Unità di controllo – entità A

L’unità di controllo è stata sviluppata con un approccio comportamentale, con un singolo process, in logica cablata.

L’unità di controllo coordina le attività dell’unità operativa attraverso un segnale *req*, che viene utilizzato per l’handshaking, segnalando all’unità di controllo di B che è pronto il risultato dell’adder, un segnale *en_counter*, che permette di leggere i valori alle locazioni successive nelle due ROM e un segnale *en_read*, che abilita alla lettura dei valori dalle due ROM.

In ingresso, oltre ai classici segnali *start*, *clock* e *reset*, riceve un segnale *fine* dall’unità operativa, che notifica quando sono finiti i valori da leggere e un segnale *ack*, utilizzato per l’handshaking.

```
1 entity Unita_Controllo_A is
2     Port (
3         clock      :  in  std_logic;
4         reset      :  in  std_logic;
5         start      :  in  std_logic;
6         ack        :  in  std_logic;
7         fine       :  in  std_logic;
8         req        :  out std_logic;
9         en_count   :  out std_logic;
10        en_read    :  out std_logic
11    );
12 end Unita_Controllo_A;
```

L’unità di controllo evolve attraverso i seguenti stati: *idle*, *read*, *send*, *wait*, *increment*, *end_check*.

Lo stato iniziale è lo stato *idle*, dove l’unità di controllo vi permane fin quando non riceve dall’esterno un segnale di *start* che avvia l’evoluzione tra gli stati della macchina. Successivamente, va in uno stato di *read*, dove abilita la lettura dei valori da sommare dalle ROM. Notiamo inoltre che l’adder, essendo una macchina combinatoria, ed essendo quindi sprovvista di un’abilitazione, una volta avvenuta la lettura dalle memorie, inizierà subito a sommare. Pertanto, bisogna poi, in fase di simulazione, scegliere opportunamente il periodo del clock, affinché in un ciclo di clock l’adder riesca a fare la somma correttamente. Dunque, per andare allo stato successivo dell’automa, devo *dimensionare il clock con un periodo almeno pari al tempo di commutazione dell’adder*.

Poi, l’unità di controllo va nello stato *invio*, nel quale, dopo esserci assicurati che la somma sia avvenuta correttamente, viene alzato il segnale *req*. La macchina poi va nello stato di *wait*, mettendosi in attesa del segnale di *ack* da parte dell’unità di controllo dell’entità B. Infine, l’unità di controllo di A prosegue la sua evoluzione attraverso gli stati *increment*, dove si abilita il contatore per incrementare il suo valore e *end_check*, dove viene verificato se ci sono altri valori da leggere.

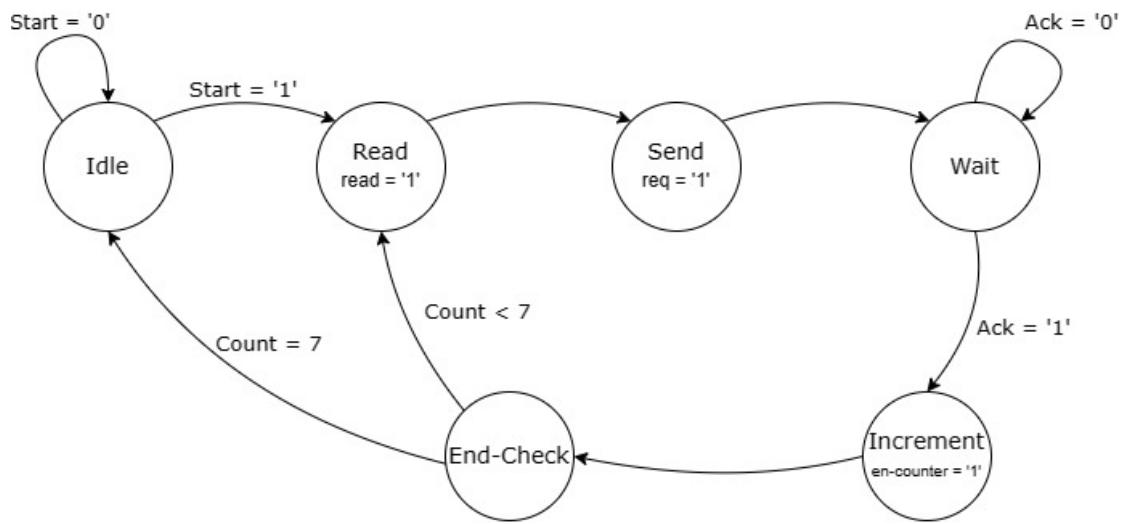


Figura 8.4: Automa unità di controllo entità A

Di seguito è riportata l'implementazione in VHDL.

```

1 architecture Behavioral of Unita_Controllo_A is
2
3 type stato is (idle, read, send, wait_ack, increment, end_check);
4 signal curr : stato := idle;
5
6 begin
7
8 fsmA : process(clock, reset)
9 begin
10 if reset = '1' then
11     req <= '0';
12     en_count <= '0';
13     en_read <= '0';
14     curr <= idle;
15 end if;
    
```

```
16
17    if rising_edge(clock) then
18        case curr is
19            when idle =>
20                req <= '0';
21                en_count <= '0';
22                en_read <= '0';
23                if start = '1' then
24                    curr <= read;
25                else
26                    curr <= idle;
27                end if;
28            when read =>
29                req <= '0';
30                en_count <= '0';
31                en_read <= '1';
32                curr <= send;
33            when send =>
34                req <= '1';
35                en_count <= '0';
36                en_read <= '0';
37                curr <= wait_ack;
38            when wait_ack =>
39                req <= '1';
40                en_count <= '0';
41                en_read <= '0';
42                if ack = '1' then
43                    curr <= increment;
```

```
44      else
45          curr <= wait_ack;
46      end if;
47
48      when increment =>
49          req <= '0';
50          en_count <= '1';
51          en_read <= '0';
52          curr <= end_check;
53
54      when end_check =>
55          req <= '0';
56          en_count <= '0';
57          en_read <= '0';
58          if fine = '1' then
59              curr <= idle;
60          else
61              curr <= read;
62          end if;
63
64      end case;
65
66  end if;
67
68  end process;
69
70  end Behavioral;
```

8.2.3 Unità operativa – entità B

L'unità operativa, come quella già sviluppata per l'entità B, è stata progettata mediante un approccio strutturale.

Comparatore

Il comparatore è stato realizzato tramite un approccio comportamentale. Il comparatore riceve in ingresso il dato inviato dall'entità A, il cui scopo è quello di verificare se tale valore sia positivo o negativo, in modo da poter decidere a posteriori in quale memoria scriverlo, comparandolo dunque col valore 0.

```
1 entity CompB is
2     Port(
3         enable : in std_logic;
4         sum_in : in std_logic_vector(0 to 4);
5         res : out std_logic
6     );
7 end CompB;
8
9 architecture Behavioral of CompB is
10
11 begin
12 process(sum_in)
13 variable var : integer := 0;
14
15 begin
```

```
16  var := to_integer(signed(sum_in));
17
18  if var < 0 then
19      res <= '1';
20  else
21      res <= '0';
22  end if;
23
24 end process;
25 end Behavioral;
```

Il risultato in uscita sarà 0 se il valore è positivo, 1 se viceversa.

Demux

Siccome bisogna memorizzare il dato ricevuta da A, in funzione del suo segno, è necessario dunque reindirizzare tale dato su una delle due memorie. A tale scopo è stato implementato un demultiplexer 2:1 con un approccio comportamentale.

```
1 entity Demux1_2 is
2     Port(
3         X    : in std_logic;
4         SEL : in std_logic;
5         Y    : out std_logic_vector(1 downto 0)
6     );
7 end Demux1_2;
8
```

```

9  architecture bhv of Demux1_2 is
10 begin
11 process(X, SEL)
12 begin
13 if(SEL = '0') then
14     Y(0) <= X;
15     Y(1) <= '0';
16 elsif(SEL = '1') then
17     Y(0) <= '0';
18     Y(1) <= X;
19 end if;
20 end process;
21
22 end bhv;

```

Abbiamo in ingresso un segnale che andrà successivamente mappato in uscita, X, e la selezione SEL. La selezione viene mappata nell'unità operativa con l'uscita del comparatore, in modo tale che se il risultato della somma dovesse essere positiva, allora in ingresso alla selezione avremo 0 in ingresso alla selezione, e dunque il demultiplexer mappa X in uscita su Y (0), viceversa invece la selezione risulterebbe pari ad 1 e dunque avremo X in uscita su Y (1).

In questo modo dunque possiamo indirizzare il valore della somma in una sola delle due memorie. Tuttavia, bisogna tenere in considerazione anche il contatore per poter incrementare l'indirizzo di una delle due memorie. Dunque per il progetto si è utilizzato un demultiplexer

sia per la memoria e sia per i contatori. Quindi il segnale in ingresso X verrà mappato con l'apposito segnale di abilitazione (della memoria o del contatore), e in base al valore della selezione, esso verrà dato a sua volta in ingresso a solo uno dei due dispositivi; mostriamo il port mapping fatta nell'unità operativa per maggior chiarezza.

```
1 demux_count : Demux1_2
2
3     Port map(
4         X      => en_count,
5         SEL    => res_comp,
6         Y(0)   => en_count1, -- enable contatore 1
7         Y(1)   => en_count2  -- enable contatore 2
8     );
9
10
11 demux_mem : Demux1_2
12
13     Port map(
14         X      => en_write,
15         SEL    => res_comp,
16         Y(0)   => en_mem1,  -- enable memoria 1
17         Y(1)   => en_mem2  -- enable memoria 2
18     );
```

Come si può notare dal codice, abbiamo due demultiplexer, sia per i contatori `demux_count`, sia per le memorie, `demux_mem`. Si osserva come il segnale X sia mappato dunque con un apposita abilitazione, `en_count` per i contatori e `en_mem` per le memorie, e la selezione viene mappata con il risultato del comparatore `res_comp`. In uscita

dunque come accennato prima avremo che il segnale di abilitazione andrà ad uno solo dei due dispositivi.

Memorie

L'unità operativa possiede due memorie dove poter scrivere in una il risultato positivo della somma, nell'altra se viceversa, dove tale controllo abbiamo visto viene fatto dal comparatore.

```
1  entity memory is
2
3      Port (
4          clock      : in std_logic;
5          reset      : in std_logic;
6          write       : in std_logic;
7          addr        : in std_logic_vector(0 to 2);
8          value_in    : in std_logic_vector(0 to 4) -- somma
9      );
10
11 end memory;
```

In ingresso abbiamo l'indirizzo `addr`, il quale sarà incrementato da un contatore, di cui ciascuna delle memorie possiede, in modo tale da incrementare l'indirizzo di uno o l'altra a seconda del risultato.

Il valore del risultato andrà comunque in ingresso a *entrambe le memorie*, tuttavia grazie all'implementazione dei demultiplexer, abbiamo che solo una delle due memorie riceverà il segnale di abilitazione, in modo tale che la scrittura avverrà in mutua esclusione. Analogamente

dunque per quanto concerne i due contatori, anch'essi lavoreranno in mutua esclusione ricevendo solo uno di loro l'abilitazione.

Registro

Il registro è stato realizzato secondo un approccio comportamentale.

Il suo scopo è di contenere il valore ricevuto dall'entità A.

```
1 entity Reg is
2     Port(
3         clock : in std_logic;
4         reset : in std_logic;
5         enable : in std_logic;
6         value_in : in std_logic_vector(0 to 4);
7         value_out : out std_logic_vector(0 to 4)
8     );
9 end Reg;
10
11 architecture Behavioral of Reg is
12
13 begin
14 process (clock, reset)
15 begin
16     if reset = '1' then
17         value_out <= (others => '0');
18     end if;
19     if rising_edge(clock) then
20         if enable = '1' then
```

```

21      value_out <= value_in;
22
23  end if;
24 end process;
25 end Behavioral;
```

8.2.4 Unità di controllo – entità B

L’unità di controllo è stata realizzata in logica cablata, mediante un’automma a stati finiti, con un approccio comportamentale.

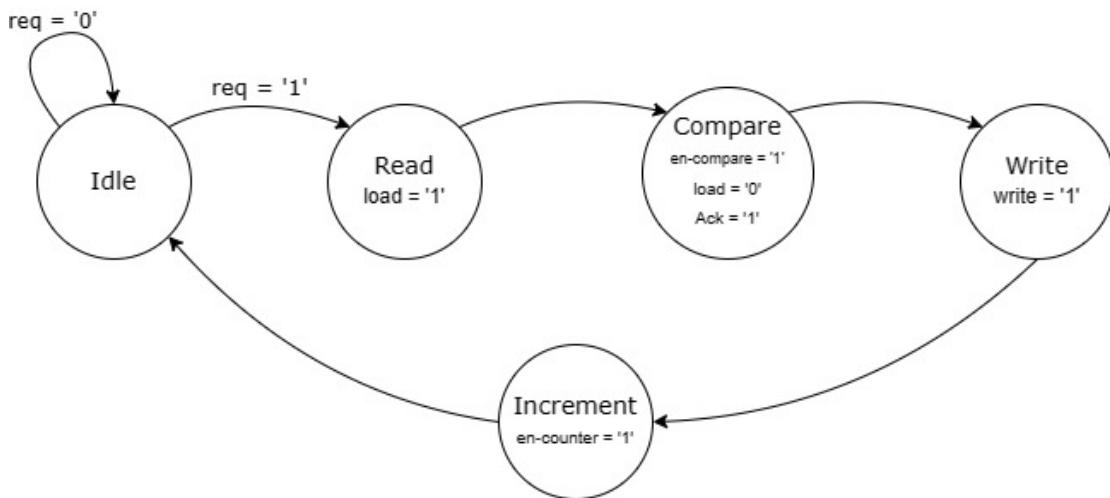


Figura 8.5: Automa unità di controllo entità B

L’automa rappresentate la rete di controllo di B evolve attraverso i seguenti stati: *Idle*, *Read*, *Compare*, *Write*, *Increment*. L’automa parte dallo stato *idle* e vi permane fin quando non arriva una richiesta da parte dell’entità A, ovvero quando *req*, dopodiché evolve e verso lo stato di *Read*. A questo punto viene letto il valore e viene abilitato

un segnale per la lettura dal registro, *load*. Successivamente arriva nello stato di *Compare* dove viene alzato il segnale di *en-compare*, per poter abilitare il comparatore, e viene alzato il valore dell'*ack*. Dopo la comparazione la macchina scrive in memoria il valore, nello lo stato *Write*, dove viene resto alto il segnale apposito per la scrittura per le memorie, *write*. Infine evolve nello stato *Increment*, dove viene alzata l'abilitazione dei contatori per l'indirizzo delle memorie, *en-counter*, dopodichè si porterà sempre nello stato di *Idle* per attendere un'ulteriore richiesta da parte di A.

Di seguito è riportata l'implementazione in VHDL.

```

1 entity Unita_Controllo_B is
2     Port (
3         clock      : in  std_logic;
4         reset       : in  std_logic;
5         req        : in  std_logic;
6         ack        : out std_logic;
7         load       : out std_logic;
8         en_count   : out std_logic;
9         en_write   : out std_logic;
10        en_comp    : out std_logic
11    );
12 end Unita_Controllo_B;
13
14 architecture Behavioral of Unita_Controllo_B is
15
```

```
16 type stato is (idle, read, compare, write, increment);  
17 signal curr : stato := idle;  
18  
19 begin  
20  
21 fsmB : process(clock, reset)  
22 begin  
23 if reset = '1' then  
24     ack <= '0';  
25     en_write <= '0';  
26     en_comp <= '0';  
27     curr <= idle;  
28 end if;  
29  
30 if rising_edge(clock) then  
31     case curr is  
32         when idle =>  
33             ack <= '0';  
34             load <= '0';  
35             en_count <= '0';  
36             en_write <= '0';  
37             en_comp <= '0';  
38             if req = '1' then  
39                 curr <= read;  
40             else  
41                 curr <= idle;  
42             end if;  
43         when read =>
```

```
44      ack <= '0';
45      load <= '1';
46      en_count <= '0';
47      en_write <= '0';
48      en_comp <= '0';
49      curr <= compare;
50
51      when compare =>
52          ack <= '1';
53          load <= '0';
54          en_count <= '0';
55          en_write <= '0';
56          en_comp <= '1';
57          curr <= write;
58
59      when write =>
60          ack <= '0';
61          load <= '0';
62          en_count <= '0';
63          en_write <= '1';
64          en_comp <= '0';
65          curr <= increment;
66
67      when increment =>
68          ack <= '0';
69          load <= '0';
70          en_count <= '1';
71          en_write <= '0';
72          en_comp <= '0';
73          curr <= idle;
74
75      end case;
```

```

72   end if;
73 end process;
74 end Behavioral;
```

8.2.5 Sistema complessivo

Infine, mostriamo lo schematic del sistema complessivo, insieme al risultato della simulazione. Il sistema inizia a funzionare quando, dall'esterno, alziamo il segnale di *start*.

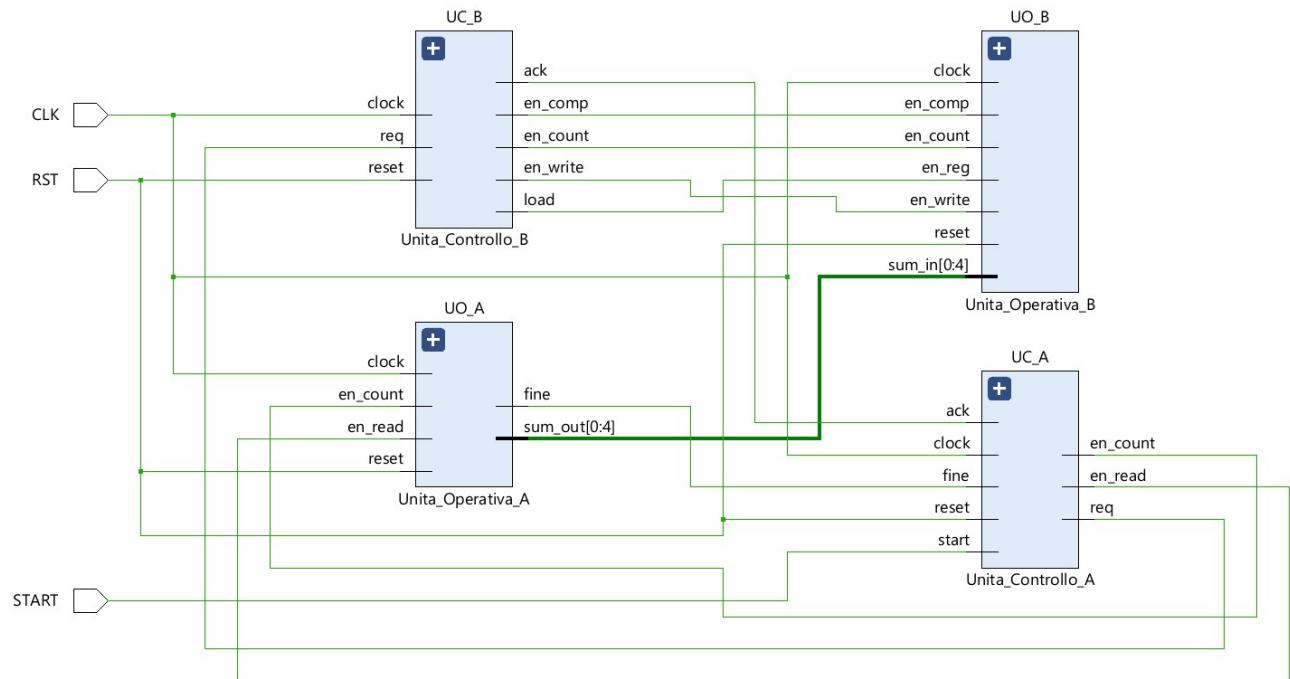


Figura 8.6: Schematic del sistema complessivo

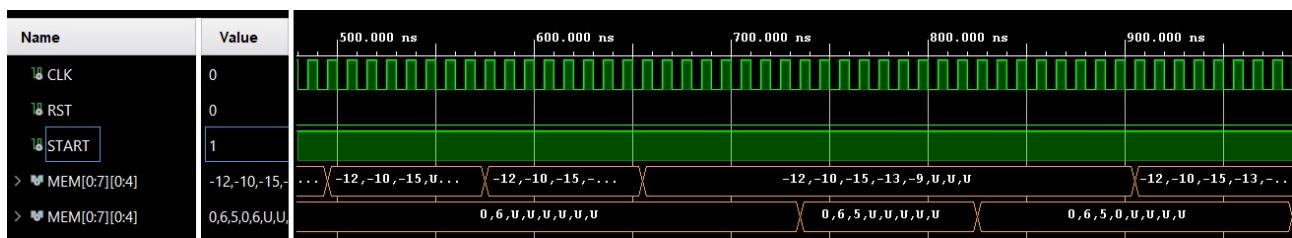


Figura 8.7: Testbench del sistema complessivo

Come possiamo osservare, nella prima memoria abbiamo i valori -12, -10, -15, -13, -9, quindi tutti i valori negativi mentre nella seconda memoria abbiamo i valori 0, 6, 5, quindi tutti valori positivi (e nulli).