

Tesina di Architettura dei Sistemi Digitali

Gruppo 35

Giovanni Celentano - Mat. M63/001164 Anna Lamboglia - Mat. M63/001219
Agostino Vitaglione - Mat. M63/001214
Mario Vitaglione - Mat. M63/001213

25 gennaio 2021

Indice

1 Decoder 4:16	2
1.1 Traccia	2
1.2 Soluzione	2
1.2.1 Schematici	2
1.2.2 Codice	4
1.3 Simulazione	8
2 Riconoscitore di Sequenza	9
2.1 Traccia 1	9
2.2 Soluzione 1	9
2.2.1 Schematici	9
2.2.2 Codice	11
2.3 Simulazione	19
2.4 Traccia 2	21
2.5 Soluzione 2	21
2.5.1 Schematici	21
2.5.2 Codice	22
2.6 Simulazione	31
3 Orologio	33
3.1 Traccia	33
3.2 Problemi riscontrati	33
3.3 Soluzione	34
3.3.1 Schematici	35
3.3.2 Alternativa di Intertempi	37
3.3.3 Codice	37
3.4 Simulazione	56
3.5 Sintesi	59
4 Registro a Scorrimento	60
4.1 Traccia	60
4.2 Soluzione	60
4.2.1 Schematici	60
4.2.2 Codice	61
4.3 Simulazione	68

5 Operazione di Modulo	70
5.1 Traccia	70
5.2 Soluzione	70
5.2.1 Handshaking	72
5.2.2 Control Store	73
5.2.3 Codice	73
5.3 Simulazione	85
6 Comunicazione tra sistemi mediante buffer	88
6.1 Traccia	88
6.2 Soluzione	88
6.2.1 System A	90
6.2.2 System B	91
6.2.3 Convertitore 4/8	92
6.2.4 Codice	94
6.3 Simulazione	109
6.4 Sintesi	110
6.4.1 Mapping su scheda	111
7 Prodotto Scalare	113
7.1 Traccia	113
7.2 Soluzione	113
7.2.1 Unità Operativa	115
7.2.1.1 Vector	116
7.2.1.2 ALU	117
7.2.2 Unità di Controllo	118
7.2.2.1 Automi dell'unità di controllo	119
7.2.2.2 Microistruzioni	122
7.2.2.3 Struttura interna	123
7.3 Codice	124
7.4 Simulazione	141
8 Processore Mic-1	142
8.1 Traccia	142
8.2 Programma	142
8.2.1 RAM	143
8.3 Simulazione del programma sul processore Mic-1	144
8.3.1 Prima simulazione	144
8.3.2 Identificazione e risoluzione del problema	145
8.3.3 Seconda simulazione	147
8.4 Flusso di esecuzione dell'istruzione ISUB	148
8.5 Flusso di esecuzione dell'istruzione ISTORE	150
8.6 Aggiunta di un codice operativo	153
8.7 Istruzioni Input/Output	155

9 Comunicazione Seriale	157
9.1 Traccia	157
9.2 Interfaccia UART	157
9.3 Problematiche e soluzioni UART Tappo	158
9.3.1 Codice UART Tappo	160
9.3.2 Simulazione UART Tappo	163
9.3.3 Mapping sulla scheda	164
9.4 Problematiche e soluzioni 2 UART	166
9.4.1 Codice 2 UART	167
10 Reti di Interconnessione	170
10.1 Traccia	170
10.2 Introduzione	170
10.3 Rete di Interconnessione a Priorità	172
10.3.1 Switch 2:2	172
10.3.2 Architettura completa: Switch Multistadio	173
10.3.3 Codice della rete di interconnessione a priorità	174
10.3.4 Simulazione	181
10.4 Rete di Interconnessione con controllo distribuito	182
10.4.1 Switch 2:2	183
10.4.2 Codice della rete di Interconnessione con controllo distribuito	184
10.4.3 Simulazione	189
10.5 Switch con Protocollo	191
10.5.1 Architettura dello Switch 2:2	191
10.5.2 Unità Operativa	192
10.5.3 Unità di Controllo	192
11 Moltiplicatore di Booth	193
11.1 Traccia	193
11.2 Soluzione moltiplicatore di Booth	193
11.2.1 Adder	193
11.2.2 Shift Register	196
11.2.3 Unità di Controllo	198
11.2.3.1 Control Store	203
11.3 Codice Moltiplicatore	204
11.4 Simulazione	208

Prefazione

Il seguente elaborato è composto dall'analisi degli undici esercizi assegnati durante il corso *Architettura dei Sistemi Digitali*, anno 2020/2021, presieduto dal Professore Nicola Mazzocca e dalla Dottoressa Alessandra de Benedictis.

Tutti gli schemi, i grafi, le tabelle e i codici implementativi sono stati realizzati dai componenti del gruppo, tranne dove espressamente indicato, poiché sono stati sfruttati i file del materiale didattico messo a disposizione.

Per ogni progetto sono state esposte delle architetture che fossero in grado di rispettare le specifiche di ogni singola traccia; ove necessario, sono state proposte delle soluzioni alternative confrontando i pro e i contro.

In particolare, tutte le soluzioni proposte rispettano le specifiche in fase di simulazione, mentre alcune non sono sintetizzabili in quanto adottano costrutti che rendono più complessa l'implementazione su scheda.

I progetti sintetizzati sulla scheda *Nexys Artix-7* sono:

- Esercizio 3: Orologio/Cronometro,
- Esercizio 6: Comunicazione tra sistemi mediante buffer
- Esercizio 9: Comunicazione Seriale (Punto a)

Nei capitoli dedicati a questi progetti è stato dunque aggiunto un paragrafo che illustra come è stato effettuato il mapping sulla board con annesse immagini che mostrano il funzionamento della scheda.

In allegato a questo elaborato, vengono forniti i codici sorgente di ogni implementazione illustrata, i lucidi relativi all'esercizio 3 presentati durante il corso e il codice relativo all'ALU del processore MIC-1 (Esercizio 8).

Durante l'analisi dell'esercizio 8, è stato individuato un'imprecisione all'interno del codice implementativo dell'ALU del MIC-1; pertanto, non ci si è limitati ad approfondire ciò che era richiesto dalla traccia, bensì sono stati dedicati dei paragrafi sul riscontro e sulla metodologia adottata per la sua risoluzione.

Lo scopo dell'elaborato è di mostrare il lavoro svolto, dimostrando di aver riflettuto sugli argomenti trattati durante il corso, cercando di applicare al meglio la metodologia esposta durante le lezioni.

Capitolo 1

Decoder 4:16

1.1 Traccia

Si progetti un **decoder 4:16** utilizzando componenti decoder 2:4 opportunamente interconnessi
a) in una struttura ad albero e b) in una struttura a semiselezione.

1.2 Soluzione

Utilizzando l'approccio modulare, ovvero decomponendo la macchina da implementare in componenti più piccoli, si è proceduto con l'implementazione di un decoder 4:16 in questo modo:

- (a) 5 decoder 2:4 interconnessi in una struttura ad albero
- (b) 2 decoder 2:4 interconnessi in una struttura a semiselezione

Il componente fondamentale del progetto è il *decoder 2:4*, implementato con un approccio dataflow; tramite composizione di questo componente, come precedentemente indicato, è stato realizzato il *decoder 4:16*, implementato con un approccio strutturale.

1.2.1 Schematici

Il *decoder 2:4* è una macchina combinatoria notevole con 2 ingressi, 4 uscite e un segnale di abilitazione, rappresentata in figura 1.1.

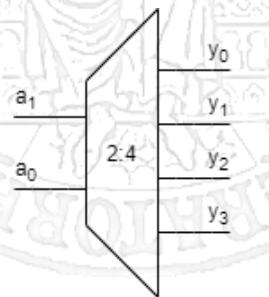


Figura 1.1: Decoder 2:4

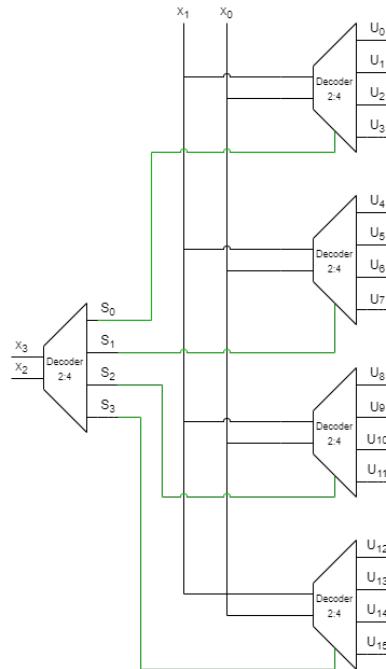


Figura 1.2: Struttura ad albero

È stata utilizzata questa componente per implementare il *decoder 4:16* tramite un approccio strutturale. In particolare, sono stati utilizzati 5 decoder per la struttura ad albero (figura 1.3) e 2 decoder per la struttura a semiselezione (figura 1.2). Quest'ultima richiede l'impiego di 16 porte logiche AND.

Per quanto riguarda la prima struttura, come è possibile notare, il decoder del primo livello ha lo scopo di abilitare quelli del secondo. Il decoder del primo livello prende in ingresso 2 dei 4 bit in ingresso al sistema complessivo; tali bit rappresentano le cifre più significative, mentre i restanti 2 bit in ingresso al sistema *decoder 4:16* sono gli input dei decoder del secondo livello della struttura. Il fulcro del progetto è far sì che le uscite del primo decoder facciano da abilitazione al secondo livello così da avere alta l'uscita corrispondente alla selezione.

Per quanto riguarda invece la struttura a semiselezione, sono utilizzati soltanto 2 *decoder 2:4* dove i bit più significativi sono gli input del decoder di sinistra e i bit meno significativi sono gli input del decoder superiore. Al variare delle cifre meno significative ci si muove lungo le righe

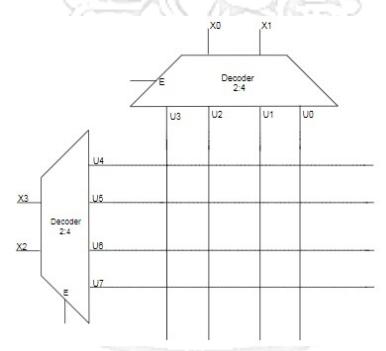


Figura 1.3: Struttura a semiselezione

della matrice formata dalle intersezioni degli output dei decoder; invece, al variare delle cifre più significative ci si muove lungo le colonne. In ogni intersezione vi è una AND così da avere in uscita gli output attesi da un *decoder 4:16*.

1.2.2 Codice

Decoder 2:4

Innanzitutto, è stata definita l'entità *decoder 2:4* che ha come input i 2 bit per la selezione e un segnale per l'abilitazione e in output un vettore di 4 bit; come scelta di progetto, il bit che corrisponde alla selezione 00 è $y(0)$ e 11 è $y(3)$. L'approccio adottato per l'implementazione è quello dataflow. Le espressioni booleane per l'assegnazione concorrente dell'uscita sono la realizzazione della tabella di verità di un decoder a 2 ingressi.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.all;
4
5 -- Definizione dell'interfaccia del modulo decoder 2 4.
6 entity decoder_2_4 is
7
8   port(  a0  : in STD_LOGIC;
9         a1  : in STD_LOGIC;
10        en  : in STD_LOGIC;
11        y    : out STD_LOGIC_VECTOR(0 to 3)
12      );
13
14 end decoder_2_4;
15
16 -- Definizione architettura del modulo decoder 2 4 attraverso il livello di
17 -- astrazione dataflow
18
19 architecture dataflow of decoder_2_4 is
20
21 begin
22   y(0) <= (((NOT a0) AND (NOT a1)) AND en);
23   y(1) <= ((a0 AND (NOT a1)) AND en);
24   y(2) <= (((NOT a0) AND a1) AND en);
25   y(3) <= ((a0 AND a1) AND en);
26 end dataflow;
```

Decoder 4:16

Per la realizzazione del *decoder 4:16* sono stati adottati due diversi approcci strutturali per sviluppare la struttura a semiselezione e la struttura ad albero, come in figura 1.3 e in figura 1.2. L'interfaccia implementata è la stessa ed è la seguente: 4 bit di selezione e un segnale di abilitazione come input e 16 bit di uscita. Per l'implementazione della struttura ad albero sono stati istanziati

5 decoder e sono stati collegati come precedentemente descritto. Similmente è stato adottato lo stesso approccio per la struttura a semiselezione. Per l'istanziazione delle 16 AND è stato utilizzato il costrutto *generate* per collegare opportunamente i segnali interni.

```

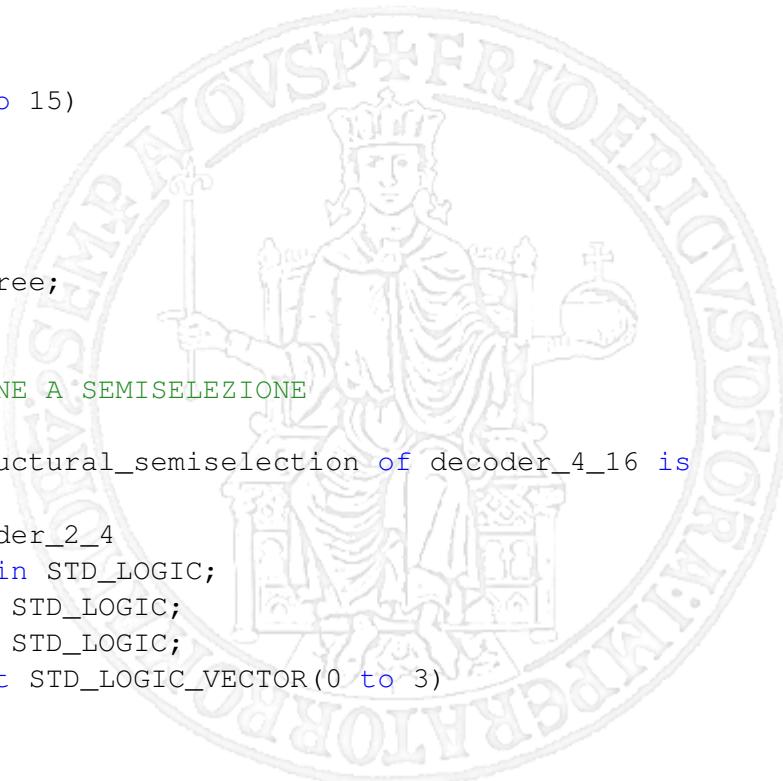
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.all;
4
5
6 -- INTERFACCIA DECODER_4_16
7
8 entity decoder_4_16 is
9
10 port(
11     b      : in STD_LOGIC_VECTOR(3 downto 0);
12     b_en   : in STD_LOGIC;
13     b_y    : out STD_LOGIC_VECTOR(0 to 15)
14 );
15
16 end decoder_4_16;
17
18 -- IMPLEMENTAZIONE AD ALBERO
19
20 architecture structural_tree of decoder_4_16 is
21
22 component decoder_2_4
23     port( a0  : in STD_LOGIC;
24            a1  : in STD_LOGIC;
25            en   : in STD_LOGIC;
26            y    : out STD_LOGIC_VECTOR(0 to 3)
27        );
28 end component;
29
30 signal s : STD_LOGIC_VECTOR (0 to 3) := (others => 'U');
31
32 begin
33     -- PRIMO DECODER DI INTERFACCIA
34     dec0: decoder_2_4
35         port map(
36             b(2),
37             b(3),
38             b_en,
39             s
40         );
41
42     -- DECODER DI SECONDO LIVELLO
43     dec1: decoder_2_4
44         port map(

```

```

46      b(0),
47      b(1),
48      s(0),
49      b_y(0 to 3)
50  );
51
52  dec2: decoder_2_4
53  port map(
54      b(0),
55      b(1),
56      s(1),
57      b_y(4 to 7)
58  );
59
60  dec3: decoder_2_4
61  port map(
62      b(0),
63      b(1),
64      s(2),
65      b_y(8 to 11)
66  );
67
68  dec4: decoder_2_4
69  port map(
70      b(1),
71      b(0),
72      s(3),
73      b_y(12 to 15)
74  );
75
76
77
78 end structural_tree;
79
80
81 -- IMPLEMENTAZIONE A SEMISELEZIONE
82
83 architecture structural_semiselection of decoder_4_16 is
84
85 component decoder_2_4
86  port( a0 : in STD_LOGIC;
87        a1 : in STD_LOGIC;
88        en : in STD_LOGIC;
89        y : out STD_LOGIC_VECTOR(0 to 3)
90  );
91 end component;
92
93 -- AND CUSTOM
94 component and_gate

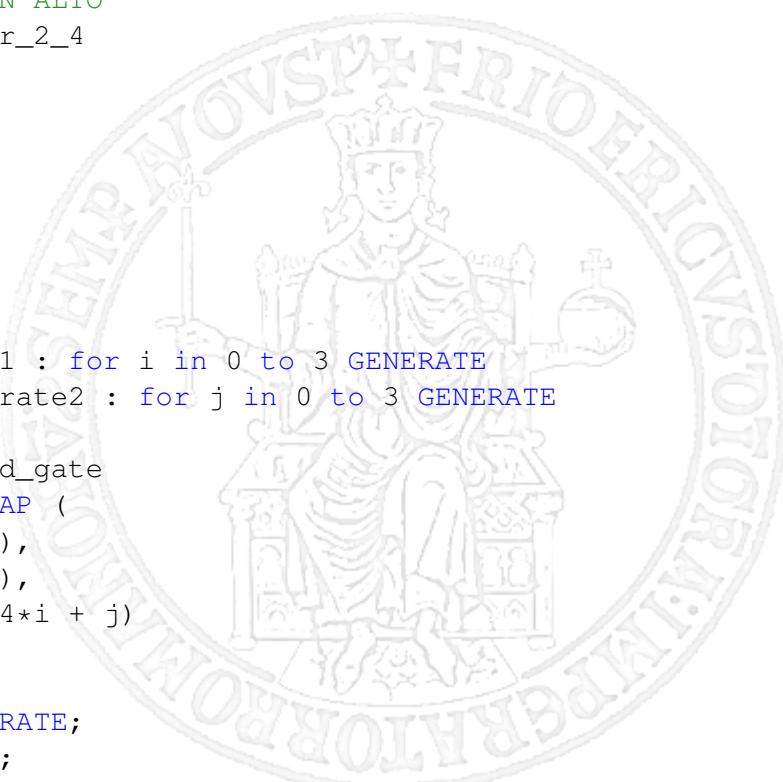
```



```

95  port (
96      in1: in std_logic;
97      in2: in std_logic;
98      o : out std_logic
99  );
100 end component;
101
102 signal s1 : STD_LOGIC_VECTOR (0 to 3) := (others => 'U'); -- SEGNALE DI
103     COLONNA
104 signal s2 : STD_LOGIC_VECTOR (0 to 3) := (others => 'U'); -- SEGNALE DI
105     RIGA
106
107 begin
108
109     -- DECODER A SINISTRA
110     dec1: decoder_2_4
111         port map(
112             b(2),
113             b(3),
114             b_en,
115             s1
116
117         );
118
119     -- DECODER IN ALTO
120     dec2: decoder_2_4
121         port map(
122             b(0),
123             b(1),
124             b_en,
125             s2
126
127         );
128
129
130     and_generate1 : for i in 0 to 3 GENERATE
131         and_generate2 : for j in 0 to 3 GENERATE
132
133             c : and_gate
134                 PORT MAP (
135                     s1(i),
136                     s2(j),
137                     b_y(4*i + j)
138                 );
139
140             END GENERATE;
141         END GENERATE;
142
143     end structural_semiselection;

```



1.3 Simulazione



Figura 1.4: Simulazione del Decoder

Per effettuare la simulazione è stato utilizzato il testbench il cui codice è possibile trovare qui di seguito. In questo caso, dato che era semplice vedere il comportamento della macchina non sono stati utilizzati degli assert. Sono state provate tutte le possibili 16 combinazioni utilizzando un loop e incrementando il segnale di input di 1 ad ogni ciclo. Dalla simulazione è possibile valutare la variazione dell'uscita del decoder.

```

1 -- Stimulus process
2 stim_proc: process
3
4 VARIABLE output : bit_vector(0 to 15) := (0 => '1', others => '0');
5 begin
6
7
8     -- hold reset state for 100 ns.
9     wait for 10 ns;
10    b_en <= '1';
11
12    FOR i IN 0 TO 15 LOOP
13        wait for 10 ns;
14        b <= std_logic_vector(input);
15        input <= input + 1;
16        output := output srl 1;
17
18    END LOOP;
19
20
21    wait;
22
23    -- insert stimulus here
24
25    wait;
26
27 end process;
28
29 END;
```

Capitolo 2

Riconoscitore di Sequenza

2.1 Traccia 1

Si vuole progettare un riconoscitore di sequenza come macchina sincrona a sincronizzazione esterna. La macchina riceve attraverso un ingresso seriale stringhe di 3 bit e, alla ricezione del terzo bit di ciascuna stringa, fornisce uscita alta se la sequenza ricevuta è **1-1**. Si disegni l'automa e si proceda alla sintesi utilizzando flip-flop D.

Implementare la macchina in VHDL utilizzando **a)** una descrizione di tipo comportamentale che faccia uso di un unico processo e **b)** una descrizione strutturale in cui vengano evidenziati tutti i componenti risultanti dalla sintesi (porte logiche e flip-flop) e le loro interconnessioni.

NOTA: per risolvere il punto b) è richiesto l'utilizzo di componenti realizzati ad hoc che implementano le porte AND e OR. L'implementazione del flip-flop D può essere fatta utilizzando una descrizione comportamentale.

2.2 Soluzione 1

Per la progettazione del riconoscitore di sequenza si è deciso di adottare un approccio di tipo comportamentale per soddisfare il punto **a** realizzando un apposito automa. Mentre per soddisfare il punto **b** è stato adottato un approccio di tipo strutturale implementando degli appositi moduli cioè **memoria**, **rete combinatoria** e un **flip flop D** utilizzato come “Buffer” in uscita. Tutti i moduli sono stati realizzati utilizzando un approccio strutturale, partendo dalle singole porte logiche fino ad arrivare alla realizzazione completa della macchina.

2.2.1 Schematici

In figura 2.1 è rappresentato l'automa realizzato per soddisfare il punto a, da come si può notare è stato realizzato un automa di Mealy, scelta più naturale per la risoluzione della problematica. Come detto in precedenza per il punto **b** invece è stato utilizzato un approccio di tipo strutturale collegando opportunamente i componenti fondamentali. Per la creazione della macchina combinatoria, sono state estratte le tabelle di verità e le espressioni booleane sia dei flip flop sia dell'uscita, in modo da costruire la rete combinatoria in figura 2.1 tramite porte **AND**, **OR** e **NOT**, progettate appositamente. Anche la memoria è stata implementata con un approccio strutturale, infatti sono stati istanziati 3 flip flop D realizzati tramite approccio comportamentale. Inoltre è stato

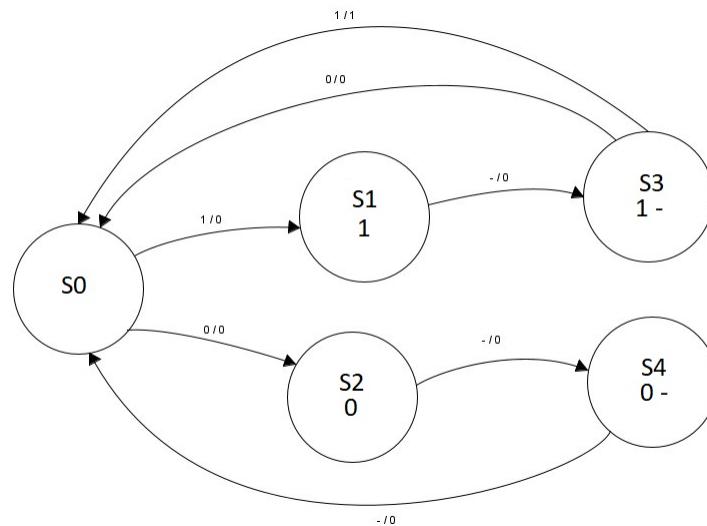


Figura 2.1: Automa a stati finiti del riconoscitore 1-1

collegato un ulteriore flip flop D in uscita alla rete combinatoria per rendere l'automa di Mealy sincrono al segnale di clock. Per quanto riguarda il sistema completo in figura 2.2 sono stati collegati i moduli **memoria** e **macchina combinatoria** fra loro e il flip flop D sull'uscita y, il tutto sincronizzato da un clock.

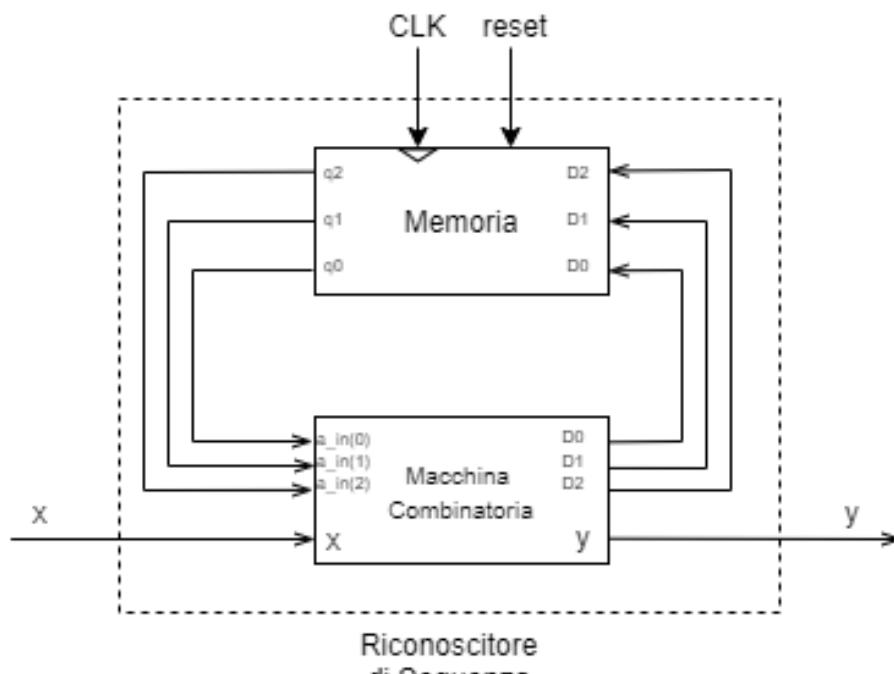


Figura 2.2: Architettura del riconoscitore di sequenza

2.2.2 Codice

Porte Logiche

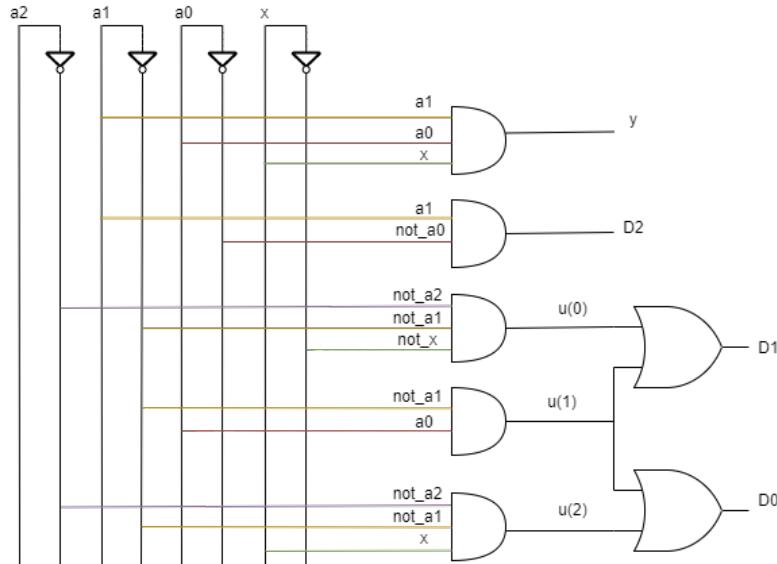


Figura 2.3: Rete combinatoria del riconoscitore di sequenza

In figura 2.3 è possibile notare che vi è la necessità di due tipologie di porte logiche AND, in particolare una con 3 ingressi e 1 uscita ed un'altra con 2 ingressi e 1 uscita. Per evitare di istanziare due componenti differenti è stato deciso in fase di implementazione di utilizzare il costrutto generic, in modo da definire soltanto un'entità.

```

1 entity and_gate is
2   generic(
3     N : positive := 2
4   );
5   port (
6     x : in std_logic_vector(0 to N-1);
7     y : out std_logic
8   );
9 end and_gate;
10
11 architecture Behavioral of and_gate is
12
13 signal y_temp : std_logic;
14
15 begin
16
17   y <= y_temp;
18
19   func : process(x)
20   variable temp : std_logic := '1';
21   begin

```

```

22      temp := '1';
23      for i in 0 to n-1 loop
24          temp := temp and x(i);
25      end loop;
26      y_temp <= temp;
27  end process;
28
29
30
31
32 end Behavioral;

```

```

1 entity port_or_2 is
2   port (
3     in1 : in std_logic;
4     in2 : in std_logic;
5     y : out std_logic
6   );
7 end port_or_2;
8
9 architecture Dataflow of port_or_2 is
10
11 begin
12   y <= in1 OR in2;
13
14 end Dataflow;

```

```

1 entity port_not is
2   port (
3     in1 : in std_logic;
4     y : out std_logic
5   );
6 end port_not;
7
8 architecture Dataflow of port_not is
9
10 begin
11   y <= NOT in1;
12
13 end Dataflow;

```

Macchina Combinatoria

Da come si può notare in figura 2.3, la rete combinatoria è stata realizzata strutturalmente tramite le varie porte logiche collegate opportunamente col costrutto *port map*.

```

1 entity rete_comb is
2   port (

```

```

3      x : in std_logic;
4      a_in : in std_logic_vector(2 downto 0);
5      d : out std_logic_vector(2 downto 0);
6      y : out std_logic
7  );
8 end rete_comb;
9
10 architecture Structural of rete_comb is
11
12     component and_gate
13         generic(
14             N : positive := 2
15         );
16
17         port (
18             x : in std_logic_vector(0 to N-1);
19             y : out std_logic
20         );
21     end component;
22
23     component port_or_2
24         port (
25             in1 : in std_logic;
26             in2 : in std_logic;
27             y : out std_logic
28         );
29     end component;
30
31     component port_not
32         port (
33             in1 : in std_logic;
34             y : out std_logic
35         );
36     end component;
37
38
39     signal not_a_in : std_logic_vector(2 downto 0) := (others => '0');
40     signal not_x : std_logic := '0';
41     signal u : std_logic_vector(0 to 2) := (others => '0'); -- SEGNALI
42             USCITA AND
43
44 begin
45
46     -- VEDERE DISEGNO PER CAPIRE
47     not_0 : port_not port map (a_in(0), not_a_in(0));
48     not_1 : port_not port map (a_in(1), not_a_in(1));
49     not_2 : port_not port map (a_in(2), not_a_in(2));
50     not_3 : port_not port map (x, not_x);

```

```

51
52
53 and_0 : and_gate generic map(3)
54     port map(
55         x(0) => a_in(1),
56         x(1) => a_in(0),
57         x(2) => x,
58         y => y
59     ); --y
60
61 and_1 : and_gate port map(
62     x(0) => a_in(1),
63     x(1) => not_a_in(0),
64     y => d(2)
65 ); --D2
66
67 and_2 : and_gate generic map(3)
68     port map(
69         x(0) => not_a_in(2),
70         x(1) => not_a_in(1),
71         x(2) => not_x,
72         y => u(0)
73     );
74
75 and_3 : and_gate port map(
76     x(0) => not_a_in(1),
77     x(1) => a_in(0),
78     y => u(1)
79 );
80
81 or_0 : port_or_2 port map (u(0), u(1), d(1)); --D1
82
83 and_4 : and_gate generic map(3)
84     port map (
85         x(0) => not_a_in(2),
86         x(1) => not_a_in(1),
87         x(2) => x,
88         y => u(2)
89     );
90
91 or_1 : port_or_2 port map (u(1), u(2), d(0)); --D0
92
93
94
95 end Structural;

```

Flip Flop D

```

1 entity flip_flop_d is
2   port(
3     d : in std_logic;
4     clk: in std_logic;
5     rst : in std_logic;
6     q : out std_logic
7   );
8 end flip_flop_d;
9
10 architecture Behavioral of flip_flop_d is
11
12 begin
13
14   process(clk, rst)
15   begin
16
17     if(rst = '1') then
18       q <= '0';
19
20     elsif(clk'event AND clk='1') then
21       q <= d;
22     end if;
23
24   end process;
25
26 end Behavioral;

```

Memoria

Da come si evince dal codice, la memoria è stata realizzata strutturalmente attraverso i flip flop creati in precedenza e collegati tramite un opportuno port map.

```

1 entity memoria is
2   port(
3     d : in std_logic_vector(2 downto 0);
4     clk : in std_logic;
5     rst : in std_logic;
6     q : out std_logic_vector(2 downto 0)
7   );
8 end memoria;
9
10 architecture Structural of memoria is
11
12   component flip_flop_d
13   port(
14     d : in std_logic;
15     clk: in std_logic;
16     rst : in std_logic;

```

```

17      q : out std_logic
18  );
19 end component;
20
21 begin
22
23   c0 : flip_flop_d port map (d(0), clk, rst, q(0));
24   c1 : flip_flop_d port map (d(1), clk, rst, q(1));
25   c2 : flip_flop_d port map (d(2), clk, rst, q(2));
26
27 end Structural;

```

Riconoscitore di Sequenza

Il riconoscitore è stato realizzato collegando tutti i componenti precedentemente descritti.

```

1 entity riconoscitore is
2   port (
3     x : in std_logic;
4     clk : in std_logic;
5     rst : in std_logic;
6     y : out std_logic
7   );
8
9 end riconoscitore;
10
11 architecture Behavioral of riconoscitore is
12
13   TYPE STATE IS (S0,S1,S2,S3,S4);
14   signal stato_corrente : STATE := S0;
15
16 begin
17
18   process(clk, rst)
19   begin
20
21     if(rst = '1') then
22       stato_corrente <= S0;
23       y <= '0';
24
25     elsif (clk'event AND clk = '1') then
26
27       case stato_corrente IS
28
29         when S0 =>
30           if(x = '0') then
31             stato_corrente <= S2;
32             y <= '0';
33           elsif (x = '1') then

```

```

34         stato_corrente <= S1;
35         y <= '0';
36     end if;
37
38     when S1 =>
39         if(x = '0' OR x = '1') then
40             stato_corrente <= S3;
41             y <= '0';
42         end if;
43
44     when S2 =>
45         if(x = '0' OR x = '1') then
46             stato_corrente <= S4;
47             y <= '0';
48         end if;
49
50     when S3 =>
51         if(x = '0') then
52             stato_corrente <= S0;
53             y <= '0';
54         elsif (x = '1') then
55             stato_corrente <= S0;
56             y <= '1';
57         end if;
58
59     when S4 =>
60         if(x = '0' OR x = '1') then
61             stato_corrente <= S0;
62             y <= '0';
63         end if;
64
65     when others =>
66         stato_corrente <= S0;
67         y <= '0';
68
69     end case;
70 end if;
71
72 end process;
73
74 end Behavioral;
75
76
77 architecture Structural of riconoscitore is
78
79 component rete_comb
80     port (
81         x : in std_logic;
82         a_in : in std_logic_vector(2 downto 0);

```

```

83      d : out std_logic_vector(2 downto 0);
84      y : out std_logic
85    );
86  end component;
87
88 component memoria
89   port(
90     d : in std_logic_vector(2 downto 0);
91     clk : in std_logic;
92     rst : in std_logic;
93     q : out std_logic_vector(2 downto 0)
94   );
95 end component;
96
97 component flip_flop_d
98   port(
99     d : in std_logic;
100    clk: in std_logic;
101    rst : in std_logic;
102    q : out std_logic
103  );
104 end component;
105
106
107 signal s_q : std_logic_vector(2 downto 0) := (others => '0');
108 signal s_d : std_logic_vector(2 downto 0) := (others => '0'); -- SEGNALE
109 signal s_b : std_logic := '0';
110           INTERNO PER IL BUFFER
111
112 begin
113   rc : rete_comb port map(x, s_q, s_d, s_b);
114   m : memoria port map (s_d, clk, rst, s_q); -- BUFFER
115   out_buff : flip_flop_d port map(s_b, clk, rst, y);
116           PER L'USCITA
117
118 end Structural;

```

2.3 Simulazione

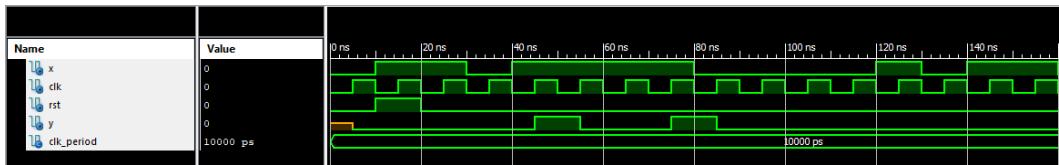


Figura 2.4: Simulazione del riconoscitore di sequenza

Per effettuare la simulazione è stato realizzato il testbench qui di seguito illustrato. In questo caso, dato che era semplice vedere il comportamento della macchina non sono stati utilizzati degli assert. Sono stati effettuati più casi test per provare il corretto funzionamento della macchina. Prima dell'inserimento delle sequenze, la macchina è stata resettata alzando e abbassando il segnale di reset. Sono state date in input le due sequenze corrette ‘101’ e ‘111’ per verificare se l'uscita fosse alta e, successivamente, le sequenze non corrette ‘000’, ‘010’ e ‘110’ che hanno generato una risposta bassa del sistema, come è possibile osservare in figura 2.4

```

1  -- Stimulus process
2  stim_proc: process
3  begin
4      -- hold reset state for 100 ns.
5      wait for clk_period;
6      rst <= '1';
7      x <= '1';
8
9      wait for clk_period;
10     rst <= '0';
11
12
13     wait for clk_period;
14     x <= '0';
15
16     wait for clk_period;
17     x <= '1';
18
19     -- CASO POSITIVO 111
20     wait for clk_period;
21     x <= '1';
22
23     wait for clk_period;
24     x <= '1';
25
26     wait for clk_period;
27     x <= '1';
28
29     -- CASO POSITIVO 000
30     wait for clk_period;
31     x <= '0';

```

```
32
33     wait for clk_period;
34     x <= '0';
35
36     wait for clk_period;
37     x <= '0';
38
39     -- CASO POSITIVO 010
40     wait for clk_period;
41     x <= '0';
42
43     wait for clk_period;
44     x <= '1';
45
46     wait for clk_period;
47     x <= '0';
48
49     -- CASO POSITIVO 110
50     wait for clk_period;
51     x <= '1';
52
53     wait for clk_period;
54     x <= '1';
55
56     wait for clk_period;
57     x <= '0';
58
59
60     -- insert stimulus here
61
62     wait;
63 end process;
64
65 END;
```



2.4 Traccia 2

Si vuole progettare un riconoscitore di sequenza come macchina sincrona a sincronizzazione esterna: la macchina fornisce uscita alta quando viene riconosciuta la sequenza **1-10**, e le sequenze possono sovrapporsi (esempio: la sequenza 11101010 produrrebbe un'uscita alta in corrispondenza del quarto, sesto e ottavo bit). Si disegni l'automa e si progetti la macchina utilizzando flip-flop D.

Implementare la macchina in VHDL utilizzando **a)** una descrizione di tipo comportamentale che faccia uso di due processi, uno che realizza la funzione di uscita e di transizione e l'altro che rappresenta la memoria di stato, e **b)** una descrizione ibrida in cui le funzioni di uscita/transizione vengano realizzate mediante un modello di astrazione di tipo dataflow e la memoria di stato (i flip-flop) sia realizzata mediante una descrizione comportamentale.

2.5 Soluzione 2

Per la creazione del riconoscitore di sequenza si è deciso di adottare un approccio di tipo comportamentale per soddisfare il punto **a**, implementando due processi, di cui uno descrive un automa appositamente progettato e l'altro implementa la memoria di stato. Per ciò che concerne il punto **b**, è stato adottato un approccio di tipo strutturale implementando i moduli **memoria**, **rete combinatoria** e **flip flop D** utilizzato come *buffer* in uscita. I componenti sono stati progettati in maniera diversa fra loro, infatti **memoria** e **flip flop D** utilizzando un'architettura behavioral invece la **rete combinatoria** con un'architettura dataflow.

2.5.1 Schematici

In figura 2.5 è rappresentato l'automa realizzato per soddisfare il punto **a**, progettato tramite un automa di Mealy.

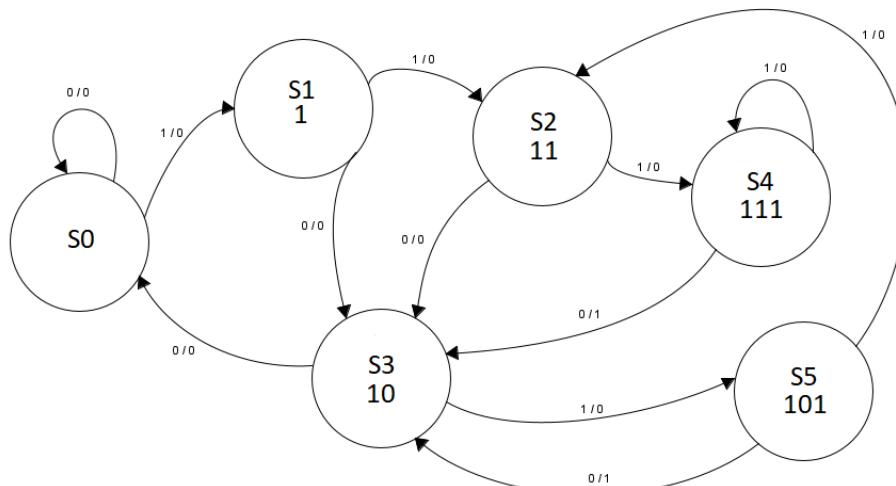


Figura 2.5: Automa a stati finiti del riconoscitore 1-10

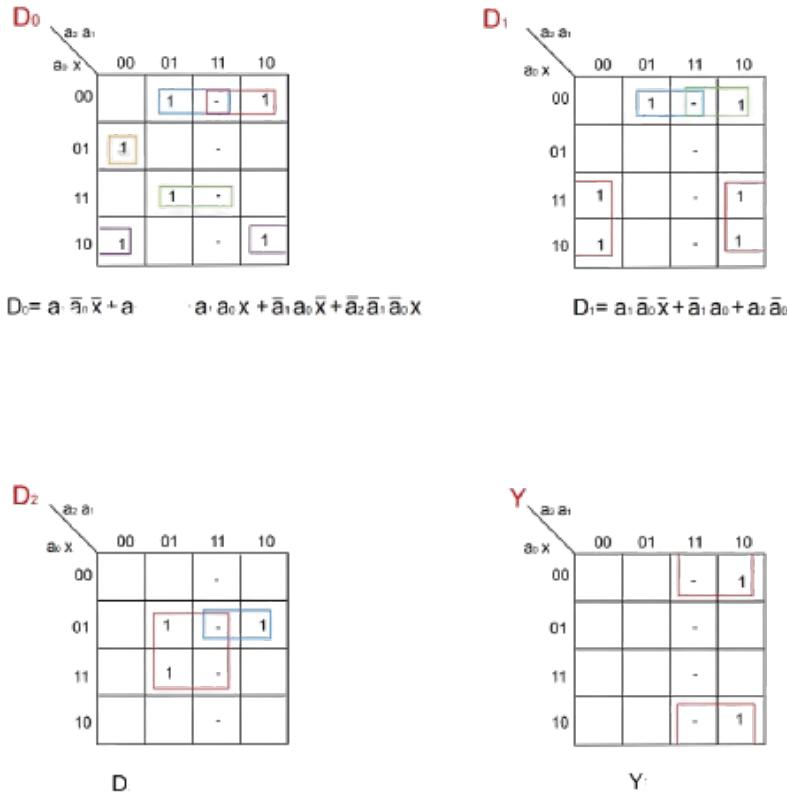


Figura 2.6: Automa a stati finiti del riconoscitore 1-10

Come detto in precedenza, per il punto b è stato utilizzato un approccio di tipo strutturale collegando i tre componenti precedentemente citati. In particolare, per progettare la rete combinatoria è stata calcolata la **tabella di verità** dall'automa e sono state generate le **Mappe di Karnaugh** in figura 2.6 per ottenere le espressioni booleane dei flip flop D e dell'uscita che sono utili per l'approccio di tipo **dataflow** adottato. Per la memoria è stato adottato un approccio **comportamentale**.

2.5.2 Codice

Per quanto riguarda la soluzione di tipo Behavioural (punto a), sono stati realizzati due processi in cui uno rappresenta la memoria di stato, mentre l'altro descrive l'automa.

```

1 entity riconoscitore is
2   port (
3     x : in std_logic;
4     clk : in std_logic;
5     rst : in std_logic;
6     y : out std_logic
7   );
8
9 end riconoscitore;
10
11 architecture Behavioral of riconoscitore is

```

```

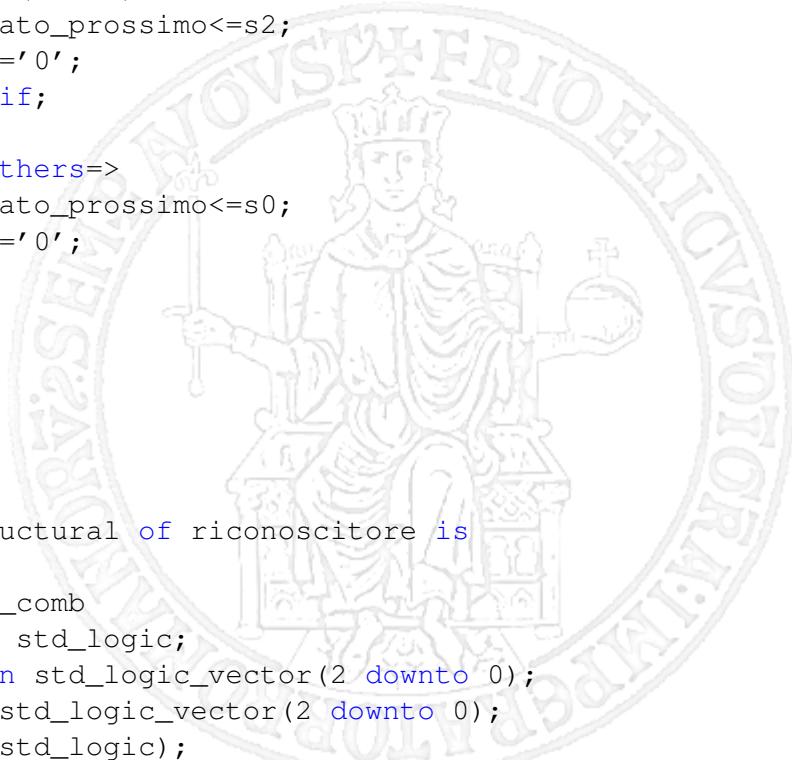
12
13     TYPE stato IS (s0,s1,s2,s3,s4,s5);
14
15     signal stato_corrente : stato := s0;
16     signal stato_prossimo : stato := s0;
17
18 BEGIN
19     mem: process(clk,rst)
20
21         begin
22             if(rst='1') then
23                 stato_corrente<=s0;
24
25             elsif(clk'event AND clk='1') then
26                 stato_corrente<=stato_prossimo;
27
28         end if;
29
30
31     end process;
32
33     rete_combinatoria: process(x,stato_corrente)
34
35         begin
36             case stato_corrente is
37                 when s0=>
38                     if(x='0') then
39                         stato_prossimo<=s0;
40                         y<='0';
41                     elsif(x='1') then
42                         stato_prossimo<=s1;
43                         y<='0';
44                     end if;
45                 when s1=>
46                     if(x='0') then
47                         stato_prossimo<=s3;
48                         y<='0';
49                     elsif(x='1') then
50                         stato_prossimo<=s2;
51                         y<='0';
52                     end if;
53
54                 when s2=>
55                     if(x='0') then
56                         stato_prossimo<=s3;
57                         y<='0';
58                     elsif(x='1') then
59                         stato_prossimo<=s4;
60                         y<='0';

```

```

61         end if;
62
63     when s3=>
64         if(x='0') then
65             stato_prossimo<=s0;
66             y<='0';
67         elsif(x='1') then
68             stato_prossimo<=s5;
69             y<='0';
70         end if;
71
72     when s4=>
73         if(x='0') then
74             stato_prossimo<=s3;
75             y<='1';
76         elsif(x='1') then
77             stato_prossimo<=s4;
78             y<='0';
79         end if;
80
81     when s5=>
82         if(x='0') then
83             stato_prossimo<=s3;
84             y<='1';
85         elsif(x='1') then
86             stato_prossimo<=s2;
87             y<='0';
88         end if;
89
90     when others=>
91         stato_prossimo<=s0;
92         y<='0';
93     end case;
94
95 end process;
96
97 end Behavioral;
98
99
100 architecture Structural of riconoscitore is
101
102 component rete_comb
103     port( x : in std_logic;
104           a_in : in std_logic_vector(2 downto 0);
105           d : out std_logic_vector(2 downto 0);
106           y : out std_logic);
107     end component;
108
109 component flip_flop_d

```



```

110     port( d : in std_logic;
111         clk: in std_logic;
112         rst : in std_logic;
113         q : out std_logic );
114 end component;
115
116 component memoria
117     port( d : in std_logic_vector(2 downto 0);
118         clk : in std_logic;
119         rst : in std_logic;
120         q : out std_logic_vector(2 downto 0));
121 end component;
122
123 signal s_d: std_logic_vector(2 downto 0);
124 signal s_q: std_logic_vector(2 downto 0);
125 signal s_buff: std_logic;
126
127 begin
128     mem : memoria port map(s_d,clk,rst,s_q);
129     rc : rete_comb port map(x, s_q, s_d, s_buff);
130     buff : flip_flop_d port map(s_buff,clk,rst,y);
131
132 end Structural;

```

Flip Flop D

```

1 entity flip_flop_d is
2 port(
3     d : in std_logic;
4     clk: in std_logic;
5     rst : in std_logic;
6     q : out std_logic
7 );
8 end flip_flop_d;
9
10 architecture Behavioral of flip_flop_d is
11
12 begin
13
14     process(clk, rst)
15     begin
16
17         if(rst = '1') then
18             q <= '0';
19
20         elsif(clk'event AND clk='1') then
21             q <= d;
22         end if;

```

```

23
24     end process;
25
26 end Behavioral;
```

Memoria

```

1  entity memoria is
2      port(
3          d : in std_logic_vector(2 downto 0);
4          clk : in std_logic;
5          rst : in std_logic;
6          q : out std_logic_vector(2 downto 0)
7      );
8  end memoria;
9
10 architecture Behavioral of memoria is
11
12 begin
13
14     m : process(clk,rst)
15
16         begin
17
18             if(rst='1') then
19                 q<="000";
20             elsif(clk'event and clk='1') then
21                 q<=d;
22             end if;
23
24         end process;
25
26 end Behavioral;
```

Rete Combinatoria

```

1  entity rete_comb is
2      port (
3          x : in std_logic;
4          a_in : in std_logic_vector(2 downto 0);
5          d : out std_logic_vector(2 downto 0);
6          y : out std_logic
7      );
8  end rete_comb;
9
10 architecture Dataflow of rete_comb is
```

```

11
12 begin
13
14   d(2)<=(a_in(1) AND x) OR (a_in(2) AND (NOT a_in(0)) AND x);
15   d(1)<=((NOT a_in(1)) AND a_in(0)) OR (a_in(1) AND (NOT a_in(0)) AND (NOT x
16     )) OR (a_in(2) AND (NOT a_in(0)) AND (NOT x));
17   d(0)<=((NOT a_in(2)) AND (NOT a_in(1)) AND (NOT a_in(0)) AND x) OR (a_in
18     (1) AND (NOT a_in(0)) AND (NOT x)) OR (a_in(2) AND (NOT a_in(0)) AND (
19       NOT x))
20     OR ((NOT a_in(1)) AND a_in(0) AND (NOT x)) OR (a_in(1) AND a_in(0)
21       AND x);
22   y<= a_in(2) AND (NOT x);
23
24 end Dataflow;

```

Riconoscitore

Il riconoscitore è stato quindi realizzato collegando tutti i componenti precedentemente descritti.

```

1 entity riconoscitore is
2   port (
3     x : in std_logic;
4     clk : in std_logic;
5     rst : in std_logic;
6     y : out std_logic
7   );
8
9 end riconoscitore;
10
11 architecture Behavioral of riconoscitore is
12
13   TYPE stato is(s0,s1,s2,s3,s4,s5);
14
15   signal stato_corrente : stato := s0;
16   signal stato_prossimo : stato := s0;
17
18 BEGIN
19   mem: process(clk,rst)
20
21     begin
22       if(rst='1') then
23         stato_corrente<=s0;
24
25       elsif(clk'event AND clk='1') then
26         stato_corrente<=stato_prossimo;
27
28     end if;
29
30

```

```
31 end process;  
32  
33 rete_combinatoria: process(x, stato_corrente)  
34 begin  
35 case stato_corrente is  
36 when s0=>  
37     if(x='0') then  
38         stato_prossimo<=s0;  
39         y<='0';  
40     elsif(x='1') then  
41         stato_prossimo<=s1;  
42         y<='0';  
43     end if;  
44 when s1=>  
45     if(x='0') then  
46         stato_prossimo<=s3;  
47         y<='0';  
48     elsif(x='1') then  
49         stato_prossimo<=s2;  
50         y<='0';  
51     end if;  
52  
53 when s2=>  
54     if(x='0') then  
55         stato_prossimo<=s3;  
56         y<='0';  
57     elsif(x='1') then  
58         stato_prossimo<=s4;  
59         y<='0';  
60     end if;  
61  
62 when s3=>  
63     if(x='0') then  
64         stato_prossimo<=s0;  
65         y<='0';  
66     elsif(x='1') then  
67         stato_prossimo<=s5;  
68         y<='0';  
69     end if;  
70  
71 when s4=>  
72     if(x='0') then  
73         stato_prossimo<=s3;  
74         y<='1';  
75     elsif(x='1') then  
76         stato_prossimo<=s4;  
77         y<='0';  
78     end if;  
79
```

```

80
81      when s5=>
82          if(x='0') then
83              stato_prossimo<=s3;
84              y<='1';
85          elsif(x='1') then
86              stato_prossimo<=s2;
87              y<='0';
88          end if;
89
90      when others=>
91          stato_prossimo<=s0;
92          y<='0';
93      end case;
94
95  end process;
96
97 end Behavioral;
98
99
100 architecture Structural of riconoscitore is
101
102     component rete_comb
103         port( x : in std_logic;
104               a_in : in std_logic_vector(2 downto 0);
105               d : out std_logic_vector(2 downto 0);
106               y : out std_logic );
107     end component;
108
109     component flip_flop_d
110         port( d : in std_logic;
111               clk: in std_logic;
112               rst : in std_logic;
113               q : out std_logic );
114     end component;
115
116     component memoria
117         port( d : in std_logic_vector(2 downto 0);
118               clk : in std_logic;
119               rst : in std_logic;
120               q : out std_logic_vector(2 downto 0));
121     end component;
122
123     signal s_d: std_logic_vector(2 downto 0);
124     signal s_q: std_logic_vector(2 downto 0);
125     signal s_buff: std_logic;
126
127 begin
128     mem : memoria port map(s_d,clk,rst,s_q);

```

```
129      rc : rete_comb port map(x, s_q, s_d, s_buff);  
130      buff : flip_flop_d port map(s_buff,clk,rst,y);  
131  
132 end Structural;
```



2.6 Simulazione



Figura 2.7: Simulazione del riconoscitore di sequenza

Per effettuare la simulazione si è realizzato il testbench qui di seguito riportato. In questo caso, poiché era semplice valutare il corretto comportamento della macchina, non sono stati utilizzati gli assert. Prima dell'inserimento delle sequenze, la macchina è stata portata in uno stato noto alzando e successivamente abbassando il segnale di reset.

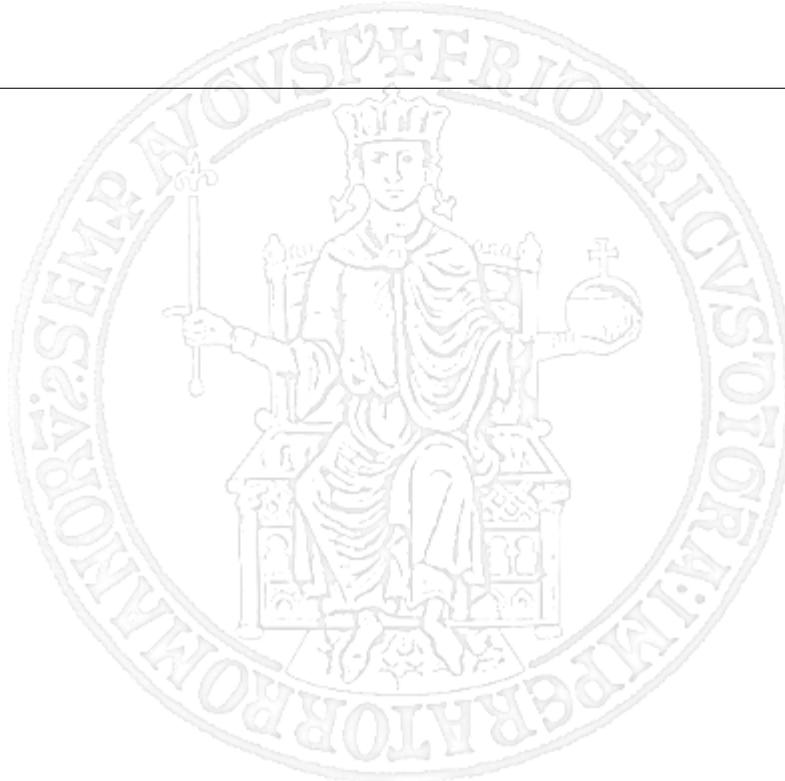
Sono state fornite in input le due sequenze corrette ‘1110’ e ‘1010’ come da traccia ed è possibile notare dalla figura 2.7 che l’uscita è alta in entrambi i casi. Inoltre, per valutare il funzionamento nel caso di sequenze non corrette, la macchina è stata resettata ed è stata inserita la sequenza “0011” così da verificare che l’uscita rimanesse bassa.

```

1  stim_proc: process
2  begin
3      -- hold reset state for 100 ns.
4
5      -- Test 111 01010
6
7      -- CASO DI TEST 1110
8      wait for clk_period;
9      rst <= '1';
10     x <= '1';
11
12    wait for clk_period;
13    rst <= '0';
14
15    wait for clk_period;
16    x <= '1';
17
18    wait for clk_period;
19    x <= '1';
20
21    wait for clk_period;
22    x <= '0';
23
24    wait for clk_period;
25    x <= '1';
26
27    wait for clk_period;
28    x <= '0';
29

```

```
30      wait for clk_period;
31      x <= '1';
32
33      wait for clk_period;
34      x <= '0';
35      --CASO DI TEST 0011
36      wait for clk_period;
37      rst <= '1';
38
39      wait for clk_period*2;
40      rst <= '0';
41
42      wait for clk_period;
43      x <= '0';
44
45      wait for clk_period;
46      x <= '0';
47
48      wait for clk_period;
49      x <= '1';
50
51      wait for clk_period;
52      x <= '1';
53      wait;
54  end process;
55
56 END;
```



Capitolo 3

Orologio

3.1 Traccia

Progettare e implementare in VHDL un orologio che, a partire da un clock di riferimento di 50MHz che opera da base dei tempi, genera, mediante uso di contatori, il secondo, il minuto e l'ora. Utilizzare un approccio strutturale collegando opportunamente i contatori secondo uno schema a scelta.

Il progetto deve prevedere la possibilità di inizializzare l'orologio con un valore iniziale, sempre espresso in termini di ore, minuti e secondi, mediante un opportuno ingresso di set (l'ingresso di set può corrispondere ad un unico segnale oppure a tre segnali differenti, a scelta dello studente) e deve prevedere un ingresso di reset per azzerare il tempo.

Opzionale: il sistema deve acquisire un insieme di al massimo N intertempi in corrispondenza di un ingresso di stop. Ogni intertempo, nella forma ora|minuto|secondo, deve essere memorizzato in una memoria interna (registri).

3.2 Problemi riscontrati

Prima di esporre la soluzione è necessario evidenziare alcune problematiche che sono state riscontrati sia in fase di progettazione che in fase sintesi. Una prima problematica riscontrata riguarda l'acquisizione dei segnali di ingresso, in quanto si potrebbero riscontrare problemi di acquisizione multipla a causa dell'elevata frequenza del segnale di clock rispetto alle dinamiche dell'input che dipendono sia da fattori meccanici sia biologici. Un secondo problema è relativo ai registri, infatti è stato ben definito il protocollo da utilizzare per la scrittura e la lettura di essi per evitare eventuali sfasamenti temporali. In vista dell'implementazione del progetto sulla board, è stato necessario adottare opportuni accorgimenti per evitare la crescita esponenziale del numero dei collegamenti tra le varie componenti e la conseguente non possibilità di sintesi. Inoltre per far sì che il progetto sia sintetizzabile è stata utilizzata una frequenza di clock pari a 100 MHz, ossia quella della scheda Nexus 4DDR Artix 7. Infine, il numero limitato di switch ha costretto ad adottare dei meccanismi che permettessero di aggirare tale problematica.

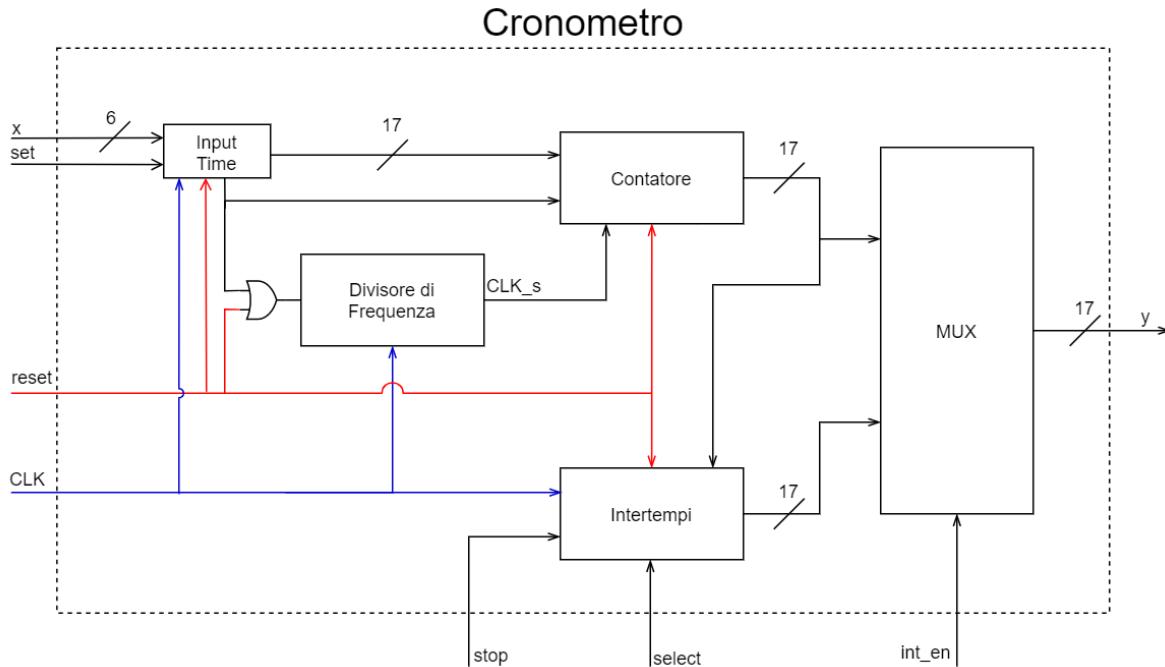


Figura 3.1: Architettura dell'orologio

3.3 Soluzione

Per la realizzazione dell'orologio/cronometro è stato utilizzato un approccio strutturale così da suddividere il problema in moduli più facilmente implementabili, come è possibile osservare in figura 3.1. I segnali che compongono l'interfaccia del orologio/cronometro sono:

- **x**: vettore contenente ore, minuti o secondi da voler settare;
- **set**: “abilitazione” per impostare l'ingresso **x**;
- **reset**;
- **clock**;
- **stop**: cattura gli intertempi;
- **select**: scorre gli intertempi catturati;
- **int_en**: decide la modalità di utilizzo;
- **y**: uscita

Il dispositivo progettato ha due funzionalità: è un *orologio*, in quanto permette di inserire un orario e tenere il tempo, e può essere utilizzato anche come *cronometro* per catturare N intertempi. L'uscita del dispositivo dipende dal segnale **int_en** che permette di impostare una delle due modalità descritte. Affinché l'orologio possa essere implementato sulla board, sono stati utilizzati 6 switch per impostare in 3 fasi distinte l'orario: prima le ore, poi i minuti ed infine i secondi; questo per evitare sfasamenti tra l'orario settato e quello effettivo. I segnali di **set**, **reset**, **stop** ed **int_en** sono mappati sui 4 dei 5 bottoni della board.

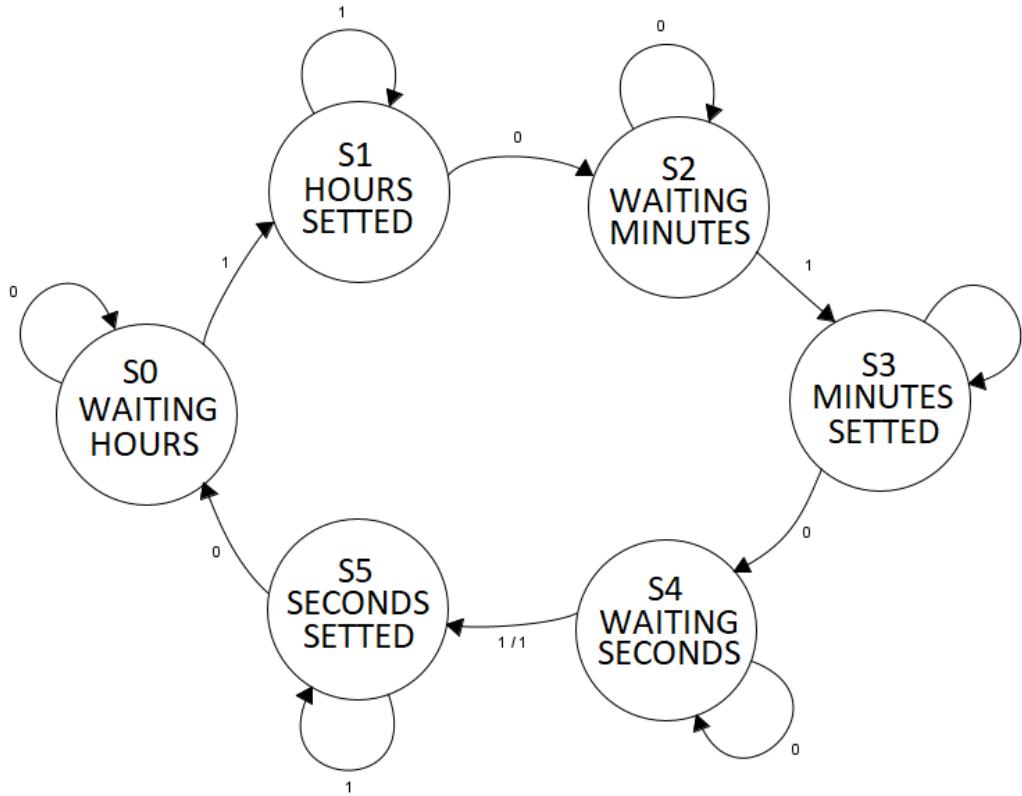


Figura 3.2: Grafo di Input Time

3.3.1 Schematici

In figura 3.1 è possibile osservare lo schema generale per l'implementazione dell'orologio. Essendo stato adottato un approccio strutturale, sono evidenti le componenti fondamentali che lo compongono. Il modulo **contatore** è il componente principale per la realizzazione dell'orologio, in quanto si occupa di incrementare il tempo, ed è stato implementato con un approccio comportamentale. Il segnale di conteggio in ingresso è un segnale di frequenza 1Hz, ossia di periodo 1s, ottenuto tramite un **divisore di frequenza**. Oltre al segnale di conteggio, in ingresso è previsto un vettore di 17 bit che, nel caso in cui il segnale di *load* sia alto, permette di impostare un determinato orario.

Il vettore di 17 bit è stato così suddiviso: partendo dai bit più significativi, i primi 5 codificano l'ora, i seguenti 6 i minuti e i restanti 6 i secondi. Tale vettore, insieme al segnale di load, è pilotato da **Input Time**; quest'ultimo è un automa a stati finiti (figura 3.2) il cui compito è ottenere in 3 fasi diverse i 17 bit che compongono l'orario e consentire di risolvere il problema dell'acquisizione multipla degli input.

Dallo schema generale è possibile osservare che il segnale di load è in ingresso anche al divisore di frequenza come segnale di reset. Infatti, resettare il segnale in uscita dal divisore di frequenza nel momento in cui si setta un nuovo orario consente di non avere un conteggio prematuro, e quindi errato, del tempo.

Il modulo **intertempi** si occupa di salvare gli istanti di tempo catturati con il segnale di stop. A differenza dei moduli appena illustrati, intertempi è stato realizzato mediante un approccio strutturale, come è possibile apprezzare dalla figura 3.3. Esso è composto da un registro a scorrimento circolare e da un'unità di controllo dedita alla generazione dei segnali di abilitazione in scrittura

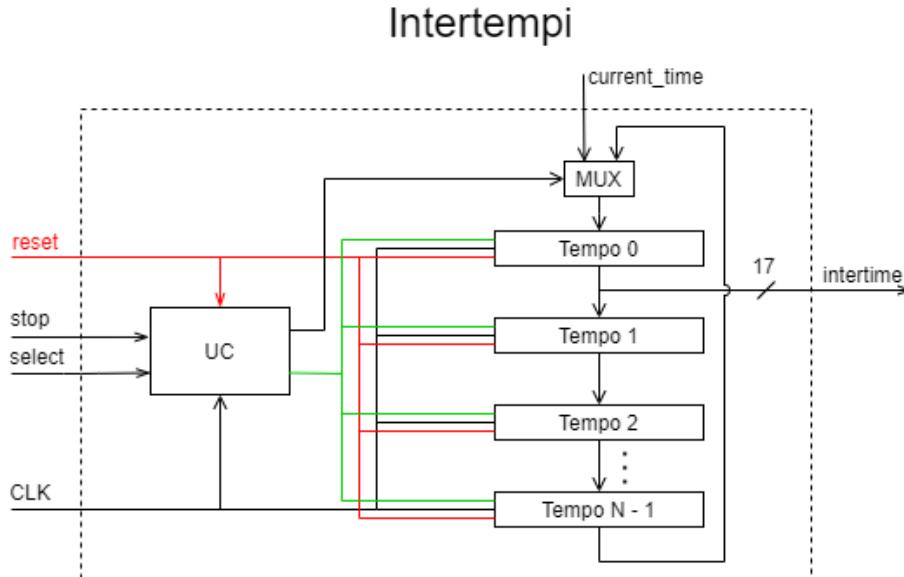


Figura 3.3: Architettura di Intertempi

Segnali ingresso/uscita:
stop sel / en mux_sel

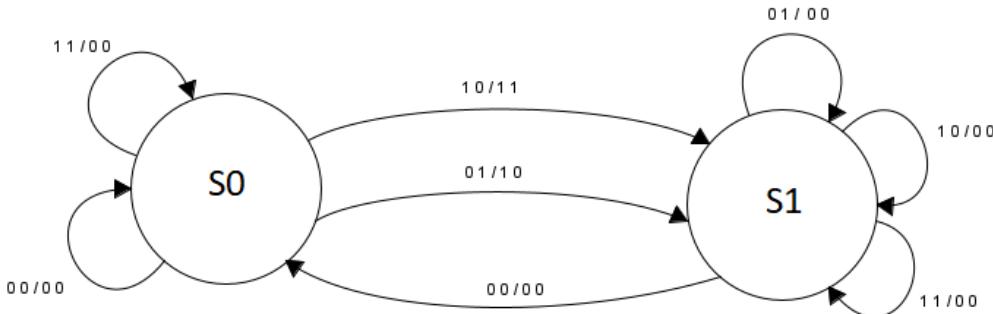


Figura 3.4: Grafo dell'Unità di Controllo di Intertempi

dei registri e del pilotaggio del multiplexer che consente di inserire un nuovo valore nel registro a scorrimento. In particolare, se **mux_sel = 1**, **intertempi** acquisisce un nuovo tempo, se **mux_sel = 0**, in **Tempo 0** viene caricato il valore di **Tempo N-1**.

Per consentire il corretto funzionamento del componente, è stata fatta particolare attenzione sull'abilitazione dei registri di memoria e la loro scrittura; infatti, sul fronte di salita del clock vengono attivati i segnali di abilitazione tramite l'unità di controllo (UC), mentre sul successivo fronte di discesa i registri di memoria acquisiscono il dato in ingresso. Infine, sul successivo fronte di salita i segnali di abilitazione vengono abbassati.

L'unità di controllo, implementata con l'automa in figura 3.4, consente di evitare acquisizioni multiple dei segnali di **stop** e di **sel**.

Il multiplexer in figura 3.1 permette di impostare l'uscita del sistema o col valore di conteggio, ossia l'orario, o con uno degli intertempi salvati.

3.3.2 Alternativa di Intertempi

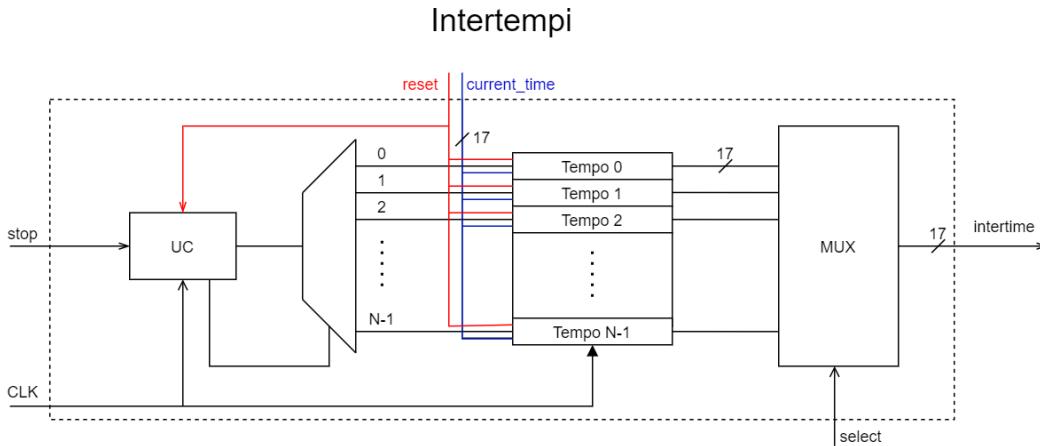


Figura 3.5: Architettura alternativa di intertempi

Prima di progettare la struttura di intertempi come visto in figura 3.3, è stata proposta un'architettura alternativa; tuttavia, rispetto alla soluzione adottata, questa presenta una complessità maggiore in termine di collegamento dei componenti che avrebbe potuto causare problemi in fase di sintesi ma non in fase di simulazione.

Come è possibile osservare in figura 3.5, la logica interna è diversa sotto vari aspetti. Prima di tutto, al posto di uno *shift register* vi è un banco di registri le cui uscite sono collegate ad un *multiplexer* che si occupa di selezionare quale intertempo visualizzare. Inoltre, *current_time* non è salvato solamente in *Tempo 0* ma può essere caricato in uno dei registri abilitato con un decoder. A tal fine, l'unità di controllo funziona da contatore: ad ogni stop, sul fronte di salita del clock abilita il decoder che seleziona il registro in cui salvare *current_time* in base al valore di conteggio; sul successivo fronte di salita, disabilita il decoder ed incrementa il contatore. In tal modo, il contatore “punta” alla locazione in cui verrà salvato il prossimo *current_time*.

Il vantaggio di tale architettura è la possibilità di poter selezionare quale intertempo visualizzare, senza dover necessariamente shiftare tutti i registri fino all'intertempo desiderato. Lo svantaggio è quello di aver un multiplexer decisamente complesso, con un numero di ingressi pari a $17 \cdot \text{NumeroRegistri}$. Per questi motivi, è stata adottata l'altra soluzione, nonostante questa possa essere implementata utilizzando pochi registri *Tempo*.

3.3.3 Codice

Contatore

Il contatore è stato implementato con un approccio behavioral con un singolo processo. I segnali di reset e di set sono asincroni rispetto al segnale di conteggio. Il vettore di 17 bit è stato suddiviso logicamente con degli alias, così da poter manipolare facilmente i bit delle ore, dei minuti e dei secondi che lo compongono. Tramite l'utilizzo di if innestati è stata implementata la logica dell'orologio.

```

1 entity contatore is
2 port (

```

```

3      x : in std_logic_vector(16 downto 0);
4      set : in std_logic;
5      clk_s : in std_logic;
6      reset : in std_logic;
7      current_time : out std_logic_vector(16 downto 0)
8  );
9
10 end contatore;
11
12 architecture Behavioral of contatore is
13
14     signal count_s : unsigned(5 downto 0) := (others => '0');
15     signal count_m : unsigned(5 downto 0) := (others => '0');
16     signal count_h : unsigned(4 downto 0) := (others => '0');
17
18     alias x_h : std_logic_vector(4 downto 0) is x(16 downto 12);
19     alias x_m : std_logic_vector(5 downto 0) is x(11 downto 6);
20     alias x_s : std_logic_vector(5 downto 0) is x(5 downto 0);
21
22
23
24 begin
25
26
27
28     current_time <= std_logic_vector(count_h & count_m & count_s);
29
30     counter : process(clk_s, set, reset)
31
32         begin
33
34             if(reset = '1') then
35
36                 count_s <= (others => '0');
37                 count_m <= (others => '0');
38                 count_h <= (others => '0');
39
40             elsif (set = '1') then
41
42                 count_h <= unsigned(x_h);
43                 count_m <= unsigned(x_m);
44                 count_s <= unsigned(x_s);
45
46
47             elsif(clk_s'event and clk_s = '1') then
48
49                 if(count_s = 59) then
50                     count_s <= (others => '0');
51

```

```

52      if(count_m = 59) then
53          count_m <= (others => '0');
54
55      if(count_h = 23) then
56          count_h <= (others => '0');
57      else
58          count_h <= count_h + 1;
59      end if; -- hour
60
61      else
62          count_m <= count_m + 1;
63      end if; -- minute
64
65      else
66          count_s <= count_s + 1;
67
68      end if; -- second
69
70  end if; -- clk'event
71
72
73 end process;
74
75
76 end Behavioral;

```

Input Time

Input Time implementa l'automa a stati finiti in figura 3.2 attraverso il costrutto switch case. Il segnale di uscita è la concatenazione del vettore delle ore, dei minuti e dei secondi acquisiti. Anche in questo caso, il segnale di reset è asincrono rispetto al clock.

```

1 entity input_time is
2     port(
3         x : in std_logic_vector(5 downto 0);
4         set : in std_logic;
5         clk : in std_logic;
6         reset : in std_logic;
7         reg_time : out std_logic_vector(16 downto 0);
8         load : out std_logic
9     );
10 end input_time;
11
12 architecture Behavioral of input_time is
13
14 type state is (S0, S1, S2, S3, S4, S5);
15
16 --alias reg_s : std_logic_vector(5 downto 0) is reg_time(5 downto 0);

```

```

17 --alias reg_m : std_logic_vector(5 downto 0) is reg_time(11 downto 6);
18 --alias reg_h : std_logic_vector(4 downto 0) is reg_time(16 downto 12);
19
20 signal current_state : state := S0;
21 signal load_temp : std_logic := '0';
22
23 signal reg_s_temp : std_logic_vector(5 downto 0) := (others => '0');
24 signal reg_m_temp : std_logic_vector(5 downto 0) := (others => '0');
25 signal reg_h_temp : std_logic_vector(4 downto 0) := (others => '0');
26
27
28 begin
29
30   load <= load_temp;
31   reg_time <= reg_h_temp & reg_m_temp & reg_s_temp;
32
33   p : process(clk, reset)
34   begin
35     if(reset = '1') then
36       current_state <= S0;
37       load_temp <= '0';
38
39     elsif (clk'event AND clk = '1') then
40
41       case current_state is
42
43         -- CARICO LE ORE
44         when S0 =>
45           if(set = '0') then
46             current_state <= S0;
47             load_temp <= '0';
48
49           else
50             reg_h_temp <= x(4 downto 0);
51             current_state <= S1;
52             load_temp <= '0';
53             end if;
54
55         -- ATTENDERE PULSANTE
56         when S1 =>
57           if(set = '0') then
58             current_state <= S2;
59             load_temp <= '0';
60
61           else
62             current_state <= S1;
63             end if;
64
65         -- SETTAGGIO MINUTI

```

```

66      when S2 =>
67          if(set = '0') then
68              current_state <= S2;
69              load_temp <= '0';
70
71      else
72          reg_m_temp <= x;
73          current_state <= S3;
74          load_temp <= '0';
75      end if;
76
77      -- ATTENDERE PULSANTE
78      when S3 =>
79          if(set = '0') then
80              current_state <= S4;
81              load_temp <= '0';
82
83      else
84          current_state <= S3;
85      end if;
86
87      -- SETTAGGIO SECONDI E INVIO DATI
88      when S4 =>
89          if(set = '0') then
90              current_state <= S4;
91              load_temp <= '0';
92
93      else
94          reg_s_temp <= x;
95          current_state <= S5;
96          load_temp <= '1';
97      end if;
98
99      -- ATTENDERE PULSANTE
100     when S5 =>
101         if(set = '0') then
102             current_state <= S0;
103             load_temp <= '0';
104
105         else
106             current_state <= S5;
107             load_temp <= '0';
108         end if;
109
110     -- ALTRI CASI
111     when others =>
112         current_state <= S0;
113         load_temp <= '0';
114

```

```

115     end case;
116
117   end if;
118
119   end process;
120
121
122 end Behavioral;
```

Divisore di Frequenza

Il divisore di frequenza è quello fornito dal materiale didattico. È stata tuttavia apportata una modifica sul segnale di reset per resettare la macchina al valore alto del segnale.

```

1  entity clock_filter is
2    generic(
3      clock_frequency_in : integer := 50000000;
4      clock_frequency_out : integer := 5000000
5    );
6    Port ( clock_in : in STD_LOGIC;
7           reset : in STD_LOGIC;
8           clock_out : out STD_LOGIC);
9  end clock_filter;
10
11 architecture Behavioral of clock_filter is
12
13 signal clockfx : std_logic := '0';
14
15 constant count_max_value : integer := clock_frequency_in/
16   clock_frequency_out)-1;
17 signal counter : integer range 0 to count_max_value := 0;
18
19 begin
20
21   clock_out <= clockfx;
22   -- reset <= not reset_n;
23
24   count_for_division: process(clock_in, reset)
25     variable counter : integer range 0 to count_max_value := 0;
26
27   begin
28
29     if reset = '1' then
30       counter <= 0;
31       clockfx <= '0';
32     elsif clock_in'event and clock_in = '1' then
33       if counter = count_max_value then
34         clockfx <= '1';
```

```

34     counter <= 0;
35   else
36     clockfx <= '0';
37     counter <= counter + 1;
38   end if;
39 end if;
40
41 end process;
42
43
44 end Behavioral;

```

Multiplexer

Il multiplexer è stato realizzato con un approccio generico. In successiva analisi è stato valutato superfluo, in quanto non necessario. Il multiplexer consente di selezionare i 17 bit in uscita o da *contatore* o da *intertempi* in base alla modalità di utilizzo selezionata.

```

1 entity mux is
2
3   generic (
4     n : positive := 2      -- Bit di ingresso al MUX per la selezione. 2^n
5       ingressi
6   );
7
8   port (
9     x  : in std_logic_vector(17*(2**n) downto 1);
10    sel : in std_logic_vector(n-1 downto 0);
11    y  : out std_logic_vector(16 downto 0)
12  );
13
14 end mux;
15
16 architecture Behavioral of mux is
17
18
19   signal y_temp : std_logic_vector(16 downto 0) := (others => '0');
20
21 begin
22
23   y <= y_temp;
24
25   p : process(sel, x)
26
27     variable temp : integer;
28
29   begin

```

```

30      temp := to_integer(unsigned(sel));
31      y_temp <= x(17 * (2**n - temp) downto (17 * (2**n - temp - 1) + 1));
32
33
34  end process;
35
36
37
38
39 end Behavioral;
```

Unità di Controllo di Intertempi

Per l'unità di controllo di intertempi è stato adottato un approccio Behavioral che implementa l'automa in figura 3.4.

```

1  entity unita_controllo is
2
3    port (
4      stop : in std_logic;
5      reset : in std_logic;
6      clk : in std_logic;
7      sel : in std_logic;
8      reg_en : out std_logic;
9      op : out std_logic      -- 0 circolare, 1 nuovo dato
10 );
11
12 end unita_controllo;
13
14 architecture Behavioral of unita_controllo is
15
16   type state is (S0, S1, S2);
17   signal current_state : state := S0;
18   signal reg_en_temp : std_logic := '0';
19   signal op_temp : std_logic := '0';
20
21 begin
22   reg_en <= reg_en_temp;
23   op <= op_temp;
24
25
26   p : process(clk, reset)
27   begin
28     if(reset = '1') then
29       current_state <= S0;
30       reg_en_temp <= '0';
31       op_temp <= '0';
32
```

```

33      elsif(clk'event AND clk = '1') then
34
35          case current_state is
36              when S0 =>
37
38                  if (stop = '1' AND sel = '0') then
39                      current_state <= S1;
40                      reg_en_temp <= '1';
41                      op_temp <= '1';
42
43                  elsif (stop = '0' AND sel = '1') then
44                      current_state <= S1;
45                      reg_en_temp <= '1';
46                      op_temp <= '0';
47
48              else
49                  current_state <= S0;
50                  reg_en_temp <= '0';
51                  op_temp <= '0';
52
53          end if;
54
55          when S1 =>
56
57              if(stop = '0' and sel = '0') then
58                  current_state <= S0;
59
60              else
61                  current_state <= S1;
62
63          end if;
64
65          reg_en_temp <= '0';
66          op_temp <= '0';
67
68          when others =>
69              current_state <= S0;
70              reg_en_temp <= '0';
71              op_temp <= '0';
72
73      end case;
74
75  end if;
76
77 end process;
78
79 end Behavioral;

```

Registro Tempo

I registro tempo consentono di memorizzare 17 bit sul fronte di discesa del clock. La scrittura avviene solamente se il segnale di abilitazione è alto, mentre la lettura è sempre consentita.

```

1  entity registro_tempo is
2
3    port (
4      x    : in std_logic_vector(16 downto 0);
5      en   : in std_logic;
6      clk  : in std_logic;
7      reset : in std_logic;
8      y    : out std_logic_vector(16 downto 0)
9    );
10
11 end registro_tempo;
12
13 architecture Behavioral of registro_tempo is
14
15   signal y_temp : std_logic_vector(16 downto 0) := (others => '0');
16
17 begin
18
19   y <= y_temp;
20
21   p : process(clk, reset)
22   begin
23
24     if(reset = '1') then
25       y_temp <= (others => '0');
26
27     elsif(clk'event and clk = '0' and en = '1') then
28       y_temp <= x;
29
30
31     end if;
32
33
34   end process;
35
36
37 end Behavioral;

```

Intertempi

Intertempi è stato progettato con un'architettura di tipo strutturale, così da poter avere un livello di dettaglio maggiore rispetto ad un approccio comportamentale. Le componenti da cui è costituito sono: l'unità di controllo, N registri tempo ed il multiplexer. In figura 3.3 è rappresentato lo schema dell'intero modulo.

```

1 entity intertempi is
2   generic (
3     n_register : positive := 2           -- NUMERO DI REGISTRI
4   );
5
6   port (
7     stop : in std_logic;
8     reset : in std_logic;
9     clk : in std_logic;
10    sel : in std_logic;                  -- SHIFT
11    current_time : in std_logic_vector(16 downto 0);
12    intertime : out std_logic_vector(16 downto 0)
13  );
14
15 end intertempi;
16
17 architecture Structural of intertempi is
18
19   component mux is
20     generic (
21       n : positive := 1      -- Bit di ingresso al MUX per la selezione. 2^n
22       ingressi
23     );
24
25     port (
26       x : in std_logic_vector(17*(2**n) downto 1);
27       sel : in std_logic_vector(n-1 downto 0);
28       y : out std_logic_vector(16 downto 0)
29     );
30   end component;
31
32   component unita_controllo is
33     port (
34       stop : in std_logic;
35       reset : in std_logic;
36       clk : in std_logic;
37       sel : in std_logic;
38       reg_en : out std_logic;
39       op : out std_logic      -- 0 circolare, 1 nuovo dato
40     );
41   end component;
42
43   component registro_tempo
44     port (
45       x : in std_logic_vector(16 downto 0);
46       en : in std_logic;
47       clk : in std_logic;
48       reset : in std_logic;
49     );
50
51   begin
52     process(clk, reset)
53     variable current_time : std_logic_vector(16 downto 0);
54     variable intertime : std_logic_vector(16 downto 0);
55     variable shift : integer;
56     variable sel : std_logic;
57     variable reg_en : std_logic;
58     variable op : std_logic;
59
60     begin
61       if reset = '1' then
62         current_time <= (others => '0');
63       elsif rising_edge(clk) then
64         if stop = '1' then
65           current_time <= (others => '0');
66         else
67           if reg_en = '1' then
68             current_time <= x;
69           else
70             if op = '0' then
71               current_time <= current_time;
72             else
73               current_time <= current_time shifted_left shift;
74             end if;
75           end if;
76         end if;
77       end if;
78       intertime <- current_time;
79     end process;
80   end;
81
82 end architecture;

```

```

48      y    : out std_logic_vector(16 downto 0)
49  );
50 end component;
51
52 signal s_sel_mux : std_logic := '0';
53 signal s_reg_en : std_logic := '0';
54 signal s_temp : std_logic_vector(1 to 17*(n_register+1)); -- s_temp(1 to
55   17) mux out, s_temp(17*n to 17*(n+1)) mux in
56 signal var : std_logic_vector(1 to 34);
57 begin
58
59   intertime <= s_temp(18 to 34);
60   var <= s_temp(17*n_register + 1 to 17*(n_register+1)) & current_time;
61
62   mx : mux
63   port map (
64     x => var,
65     sel(0) => s_sel_mux,
66     y => s_temp(1 to 17)
67   );
68
69   uc : unita_controllo
70   port map(
71     stop => stop,
72     reset => reset,
73     clk => clk,
74     sel => sel,
75     reg_en => s_reg_en,
76     op => s_sel_mux
77   );
78
79   reg_temps : for i in 0 to n_register - 1 generate
80     reg_temp : registro_tempo
81     port map(
82       x => s_temp(17*i + 1 to 17*(i+1)),
83       en => s_reg_en,
84       clk => clk,
85       reset => reset,
86       y => s_temp(17*(i+1) + 1 to 17*(i+2))
87     );
88
89   end generate;
90
91 end Structural;

```

Display Controller

Il codice del componente *Display Controller* è riportato qui di seguito. La sua utilità è illustrata in *Sintesi*.

```

1  entity display_controller is
2    port (
3      x : in std_logic_vector(16 downto 0);
4      clk : in std_logic;
5      anodo : out std_logic_vector(7 downto 0);
6      catodo : out std_logic_vector(6 downto 0)
7    );
8  end display_controller;
9
10 architecture Behavioral of display_controller is
11
12
13 constant zero   : std_logic_vector(6 downto 0) := "0000001";
14 constant one    : std_logic_vector(6 downto 0) := "1001111";
15 constant two    : std_logic_vector(6 downto 0) := "0010010";
16 constant three  : std_logic_vector(6 downto 0) := "0000110";
17 constant four   : std_logic_vector(6 downto 0) := "1001100";
18 constant five   : std_logic_vector(6 downto 0) := "0100100";
19 constant six    : std_logic_vector(6 downto 0) := "0100000";
20 constant seven  : std_logic_vector(6 downto 0) := "0001111";
21 constant eight  : std_logic_vector(6 downto 0) := "0000000";
22 constant nine   : std_logic_vector(6 downto 0) := "0000100";
23 constant F      : std_logic_vector(6 downto 0) := "0111000";
24
25 signal count : unsigned(2 downto 0) := (others => '0');
26 signal anodo_temp : std_logic_vector(7 downto 0) := (others => '1');
27 signal catodo_temp : std_logic_vector(6 downto 0) := (others => '1');
28
29 alias x_s : std_logic_vector(5 downto 0) is x(5 downto 0);
30 alias x_m : std_logic_vector(5 downto 0) is x(11 downto 6);
31 alias x_h : std_logic_vector(4 downto 0) is x(16 downto 12);
32
33 begin
34   anodo <= anodo_temp;
35   catodo <= catodo_temp;
36
37   p : process(clk)
38   begin
39
40     if(clk'event AND clk = '1') then
41       -- PRIMA CIFRA SECONDI
42       if(count = 0) then
43         count <= count + 1;
44         anodo_temp <= "11111110";
45

```

```

46      case to_integer(unsigned(x_s)) is
47
48          when 0|10|20|30|40|50 => catodo_temp <= zero;
49          when 1|11|21|31|41|51 => catodo_temp <= one;
50          when 2|12|22|32|42|52 => catodo_temp <= two;
51          when 3|13|23|33|43|53 => catodo_temp <= three;
52          when 4|14|24|34|44|54 => catodo_temp <= four;
53          when 5|15|25|35|45|55 => catodo_temp <= five;
54          when 6|16|26|36|46|56 => catodo_temp <= six;
55          when 7|17|27|37|47|57 => catodo_temp <= seven;
56          when 8|18|28|38|48|58 => catodo_temp <= eight;
57          when 9|19|29|39|49|59 => catodo_temp <= nine;
58          when others => catodo_temp <= F;
59
60      end case;
61
62      -- SECONDA CIFRA SECONDI
63      elsif (count = 1) then
64          count <= count + 1;
65          anodo_temp <= "11111101";
66
67      case to_integer(unsigned(x_s)) is
68
69          when 0 to 9 => catodo_temp <= zero;
70          when 10 to 19 => catodo_temp <= one;
71          when 20 to 29 => catodo_temp <= two;
72          when 30 to 39 => catodo_temp <= three;
73          when 40 to 49 => catodo_temp <= four;
74          when 50 to 59 => catodo_temp <= five;
75          when others => catodo_temp <= F;
76
77      end case;
78
79      -- PRIMA CIFRA MINUTI
80      elsif(count = 2) then
81          count <= count + 1;
82          anodo_temp <= "11111011";
83
84      case to_integer(unsigned(x_m)) is
85
86          when 0|10|20|30|40|50 => catodo_temp <= zero;
87          when 1|11|21|31|41|51 => catodo_temp <= one;
88          when 2|12|22|32|42|52 => catodo_temp <= two;
89          when 3|13|23|33|43|53 => catodo_temp <= three;
90          when 4|14|24|34|44|54 => catodo_temp <= four;
91          when 5|15|25|35|45|55 => catodo_temp <= five;
92          when 6|16|26|36|46|56 => catodo_temp <= six;
93          when 7|17|27|37|47|57 => catodo_temp <= seven;
94          when 8|18|28|38|48|58 => catodo_temp <= eight;

```

```

95      when 9|19|29|39|49|59 => catodo_temp <= nine;
96      when others => catodo_temp <= F;
97
98  end case;
99
100 -- SECONDA CIFRA MINUTI
101 elsif (count = 3) then
102   count <= count + 1;
103   anodo_temp <= "11110111";
104
105  case to_integer(unsigned(x_m)) is
106
107    when 0 to 9 => catodo_temp <= zero;
108    when 10 to 19 => catodo_temp <= one;
109    when 20 to 29 => catodo_temp <= two;
110    when 30 to 39 => catodo_temp <= three;
111    when 40 to 49 => catodo_temp <= four;
112    when 50 to 59 => catodo_temp <= five;
113    when others => catodo_temp <= F;
114
115  end case;
116
117
118 -- PRIMA CIFRA ORE
119 elsif(count = 4) then
120   count <= count + 1;
121   anodo_temp <= "11101111";
122
123  case to_integer(unsigned(x_h)) is
124
125    when 0|10|20 => catodo_temp <= zero;
126    when 1|11|21 => catodo_temp <= one;
127    when 2|12|22 => catodo_temp <= two;
128    when 3|13|23 => catodo_temp <= three;
129    when 4|14 => catodo_temp <= four;
130    when 5|15 => catodo_temp <= five;
131    when 6|16 => catodo_temp <= six;
132    when 7|17 => catodo_temp <= seven;
133    when 8|18 => catodo_temp <= eight;
134    when 9|19 => catodo_temp <= nine;
135    when others => catodo_temp <= F;
136
137  end case;
138
139 -- SECONDA CIFRA ORE
140 elsif (count = 5) then
141   count <= (others => '0');
142   anodo_temp <= "11011111";
143

```

```

144      case to_integer(unsigned(x_h)) is
145
146          when 0 to 9 => catodo_temp <= zero;
147          when 10 to 19 => catodo_temp <= one;
148          when 20 to 23 => catodo_temp <= two;
149          when others => catodo_temp <= F;
150
151      end case;
152
153      end if;
154  end if;
155
156 end process;
157
158 end Behavioral;

```

Cronometro

Il cronometro/orologio è stato sviluppato con un approccio strutturale. È stato realizzato con il costrutto generic per poter definire gli n registri per soddisfare la richiesta della specifica di catturare n intertempi. Tutte le componenti appena descritte sono state collegate secondo lo schema generale (figura 3.1).

```

1 entity cronometro is
2
3     generic(
4         n : positive := 2      -- Bit per codificare i registri di intertempi
5     );
6
7     port (
8         x      : in std_logic_vector(5 downto 0);
9         set    : in std_logic;
10        clk   : in std_logic;
11        reset : in std_logic;
12        stop  : in std_logic;
13        sel   : in std_logic;
14        int_en : in std_logic;
15        y      : out std_logic_vector(16 downto 0)
16
17    );
18
19 end cronometro;
20
21 architecture Structural of cronometro is
22
23     component contatore is
24
25         port (

```

```

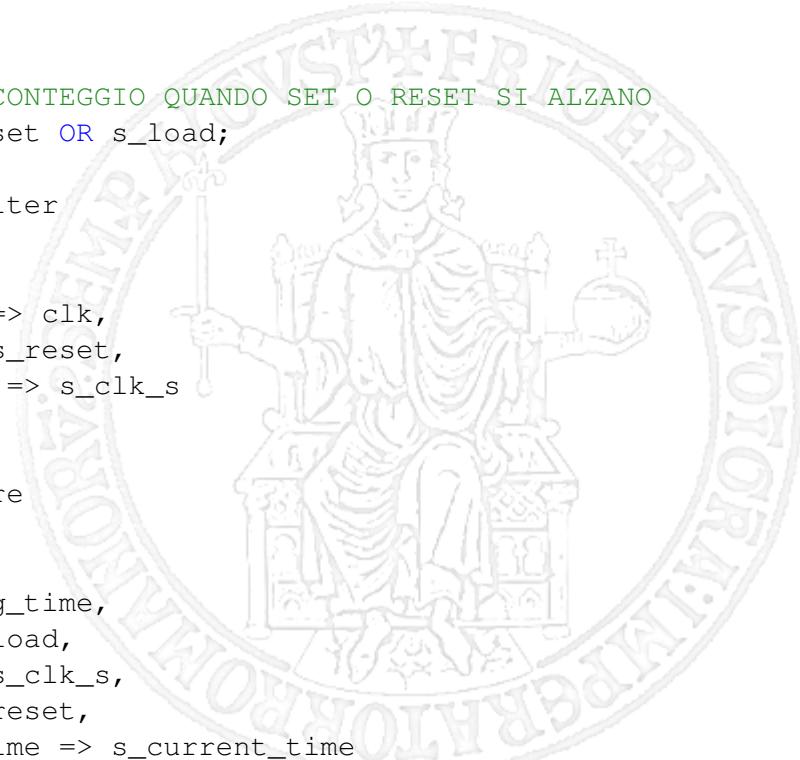
26      x : in std_logic_vector(16 downto 0);
27      set : in std_logic;
28      clk_s : in std_logic;
29      reset : in std_logic;
30      current_time : out std_logic_vector(16 downto 0)
31  );
32
33 end component;
34
35 component clock_filter is
36     generic(
37         clock_frequency_in : integer := 100000000;
38         clock_frequency_out : integer := 1      --1
39     );
40     port (
41         clock_in : in STD_LOGIC;
42         reset : in STD_LOGIC;
43         clock_out : out STD_LOGIC
44     );
45 end component;
46
47
48 component intertempi is
49     generic (
50         n_register : positive := 2           -- NUMERO DI REGISTRI
51     );
52
53     port (
54         stop : in std_logic;
55         reset : in std_logic;
56         clk : in std_logic;
57         sel : in std_logic;                -- SHIFT
58         current_time : in std_logic_vector(16 downto 0);
59         intertime : out std_logic_vector(16 downto 0)
60     );
61 end component;
62
63 component mux is
64
65     generic (
66         n : positive := 1      -- Bit di ingresso al MUX per la selezione. 2^n
67         ingressi
68     );
69
70     port (
71         x : in std_logic_vector(17*(2**n) downto 1);
72         sel : in std_logic_vector(n-1 downto 0);
73         y : out std_logic_vector(16 downto 0)
74     );

```

```

74
75    end component;
76
77    component input_time is
78        port(
79            x : in std_logic_vector(5 downto 0);
80            set : in std_logic;
81            clk : in std_logic;
82            reset : in std_logic;
83            reg_time : out std_logic_vector(16 downto 0);
84            load : out std_logic
85        );
86
87    end component;
88
89
90
91    signal s_clk_s : std_logic := '0';
92    signal s_current_time : std_logic_vector(16 downto 0) := (others => '0');
93    signal s_intertime : std_logic_vector(16 downto 0) := (others => '0');
94    signal s_reset : std_logic := '0';
95    signal s_load : std_logic := '0';
96    signal s_reg_time : std_logic_vector(16 downto 0) := (others => '0');
97
98
99 begin
100
101    -- AZZERA IL CONTEGGIO QUANDO SET O RESET SI ALZANO
102    s_reset <= reset OR s_load;
103
104    df : clock_filter
105
106        port map(
107            clock_in => clk,
108            reset => s_reset,
109            clock_out => s_clk_s
110        );
111
112    con : contatore
113
114        port map(
115            x => s_reg_time,
116            set => s_load,
117            clk_s => s_clk_s,
118            reset => reset,
119            current_time => s_current_time
120        );
121
122    it : intertempi

```



```
123  port map(
124      stop => stop,
125      reset => reset,
126      clk => clk,
127      sel => sel,
128      current_time => s_current_time,
129      intertime => s_intertime
130  );
131
132  mx : mux
133  port map(
134      x => s_current_time & s_intertime,
135      sel => (others => int_en),
136      y => Y
137
138  );
139
140  intime : input_time
141  port map(
142      x => x,
143      set => set,
144      clk => clk,
145      reset => reset,
146      reg_time => s_reg_time,
147      load => s_load
148  );
149
150
151 end Structural;
```



3.4 Simulazione



Figura 3.6: Simulazione dell’orologio

Qui di seguito è riportato il testbench per la simulazione dell’orologio. Per assicurarsi il corretto funzionamento del modulo, sono stati inseriti gli input necessari per testare tutte le funzionalità implementate. A causa di ciò, la simulazione ottenuta è risultata temporalmente lunga e non facilmente rappresentabile graficamente. Nonostante ciò, non è stato utilizzato il meccanismo degli assert per facilitare l’attività di testing, dato che è stato comunque necessario valutare graficamente i segnali interni. In virtù di ciò, è stato riportato in figura 3.6 soltanto un piccolo intervallo temporale della simulazione.

Dal grafico di simulazione è possibile notare che il conteggio varia sul fronte di salita del *clock2*, ossia il segnale di clock in uscita dal divisore di frequenza. Dopo aver catturato il valore sul fronte di salita del segnale di *stop*, è possibile selezionare il campione desiderato con il segnale *sel* (select); in quest’ultimo caso, la variazione effettiva dell’output avviene sul fronte di discesa del *clock*, poiché è su tale fronte che i *registri Tempo* memorizzano un nuovo valore.

```

1 stim_proc: process
2 begin
3     -- hold reset state for 100 ns.
4
5     wait for 10 ns;
6     reset <= '1';
7
8     wait for 10 ns;
9     reset<= '0';
10
11    -- 10111_111011_111011
12    wait for 10 ns;
13    x <= "001111";
14
15    wait for 10 ns;
16    set <= '1';
17
18    wait for 300 ns;
19    set <= '0';
20
21    wait for 10 ns;
22    x <= "101011";
23
24    wait for 10 ns;
```

```
25      set <= '1';
26
27      wait for 300 ns;
28      set <= '0';
29
30      wait for 10 ns;
31      x <= "100001";
32
33      wait for 10 ns;
34      set <= '1';
35
36      wait for 300 ns;
37      set <= '0';
38
39      wait for 400 ns;
40      stop <='1';
41
42      wait for 10 ns;
43      stop <='0';
44
45      wait for 200 ns;
46      stop<= '1';
47
48      wait for 10 ns;
49      stop <='0';
50
51      wait for 10 ns;
52      sel <= '1';
53      int_en <='1';
54
55      wait for 10 ns;
56      sel <= '0';
57
58      wait for 10 ns;
59      sel <= '1';
60
61      wait for 10 ns;
62      sel <= '0';
63
64      wait for 10 ns;
65      sel <= '1';
66
67      wait for 10 ns;
68      sel <= '0';
69
70      wait for 10 ns;
71      sel <= '1';
72
73      wait for 10 ns;
```



```
74     int_en<= '0';  
75  
76  
77     wait;  
78 end process;  
79  
80 END;
```



3.5 Sintesi

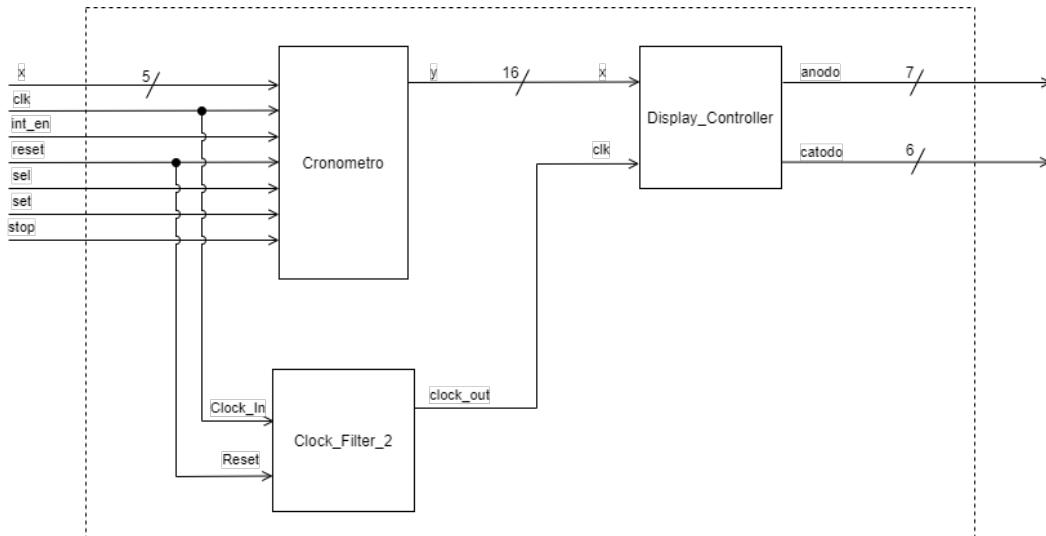


Figura 3.7: Architettura del sistema per il pilotaggio del display

Nello schema mostrato in figura 3.7 è possibile notare i componenti aggiuntivi per l'implementazione sulla scheda. In particolare, il *display_controller* permette di gestire il display a 7 segmenti della scheda, mentre il *clock_filter* è un ulteriore divisore di frequenza che riduce la frequenza del clock a 1000 Hz per fornire ai segmenti il tempo necessario di scaricarsi e di non avere artefatti in uscita visibili dal display.

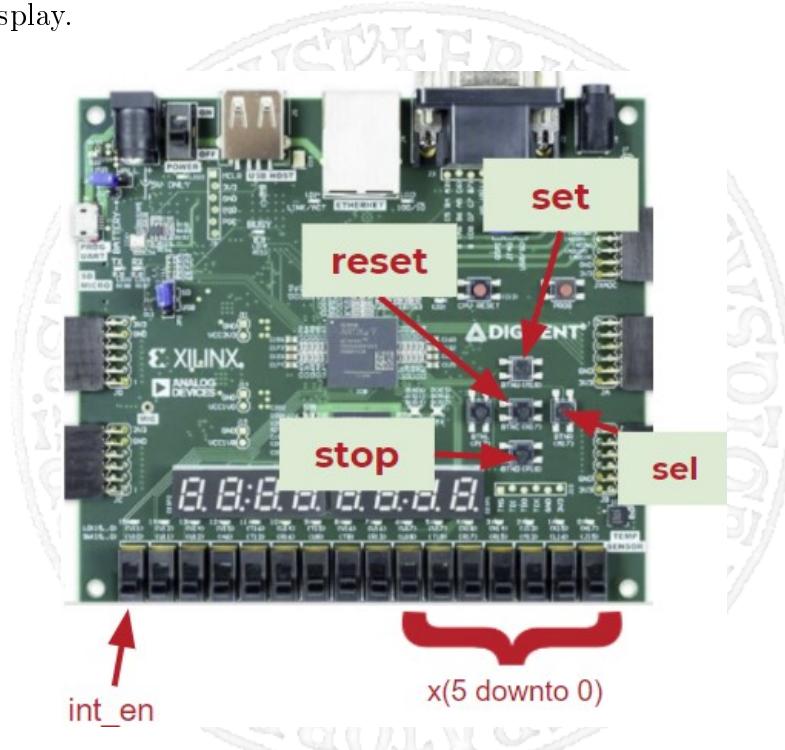


Figura 3.8: Simulazione dell'orologio

Capitolo 4

Registro a Scorrimento

4.1 Traccia

Progettare un registro a scorrimento di 4 bit in grado di operare, in base ad una selezione, in 4 diverse modalità: (1) scorrimento a sinistra con caricamento seriale di un bit pari a 0, (2) scorrimento a destra con caricamento seriale di un bit pari a 0, (3) scorrimento circolare verso sinistra, (4) scorrimento a sinistra con caricamento seriale di un bit x . Il valore iniziale del registro può essere configurato mediante un segnale di reset oppure tramite il caricamento parallelo di un valore A3A2A1A0 fornito dall'esterno, inserito grazie ad un segnale di load. Un segnale di shift regola lo scorrimento del registro.

Si progetti e implementi il registro utilizzando un approccio (a) strutturale e (b) comportamentale.

4.2 Soluzione

Per risolvere il problema posto dalla traccia è stato realizzato un registro a scorrimento sia con un approccio di tipo comportamentale (a) sia strutturale rappresentato in figura 4.1. Per il modello strutturale sono state utilizzate due macchine notevoli, ossia flip-flop di tipo D e multiplexer per il punto **b**, che sono state implementate con un approccio comportamentale.

La principale problematica affrontata consiste nell'implementare una struttura che permettesse sia un caricamento parallelo dei dati sia uno shift dei valori nei registri che varia in base alla modalità impostata. Inoltre, è stato aggiunto un segnale di *enable* per abilitare lo shift dei registri ed un segnale di load per il caricamento parallelo. Infine, è stato necessario “conservare” il valore in uscita dal registro di scorrimento in un ulteriore flip-flop.

Come è possibile notare nello schema in figura 4.1, alcuni multiplexer in input presentano più volte lo stesso segnale. Questa è stata una scelta progettuale che permette di evitare una rete combinatoria per la generazione dei segnali *ad hoc* per ogni multiplexer.

4.2.1 Schematici

Nell'architettura illustrata precedentemente in figura 4.1 viene descritta la macchina nel suo complesso. Facendo particolare attenzione, si possono notare delle particolarità: partendo dalla parte alta a scendere si nota l'utilizzo di due multiplexer necessari al corretto caricamento del valore in

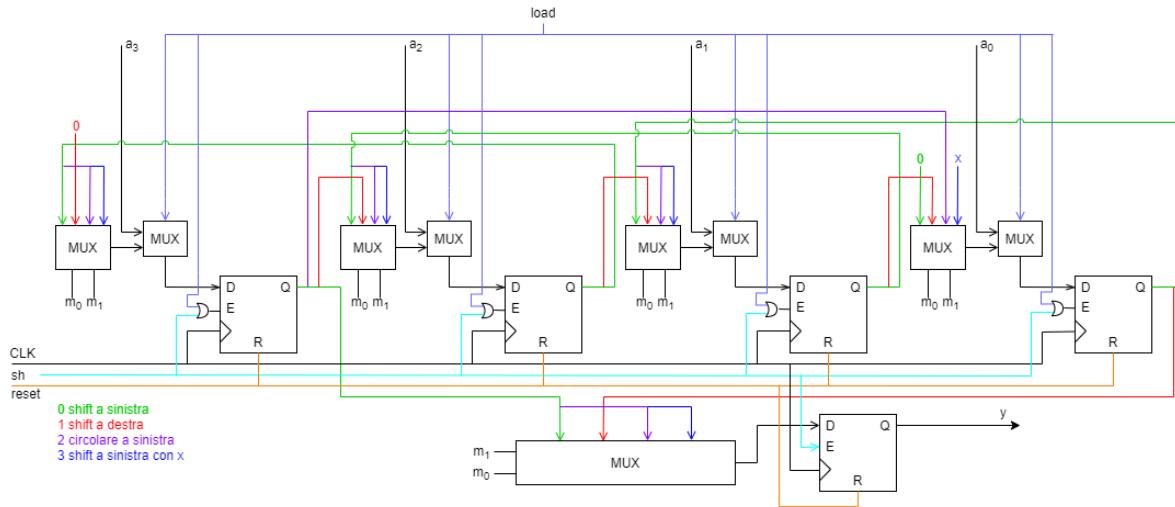


Figura 4.1: Struttura generale del Registro a Scorrimento

parallelo oppure in serie. In ingresso ad ogni flip-flop vi è una porta **or** che serve ad abilitare la scrittura nel flip-flop sia per il caricamento parallelo ($load = 1$) sia per lo shift ($sh = 1$).

Nella parte bassa dello schema è presente un multiplexer utilizzato per la selezione del corretto valore in uscita dallo shift register, poiché sono consentiti sia shift verso destra sia verso sinistra a seconda della modalità di utilizzo. Per essere più precisi, partendo da sinistra, viene selezionato il valore del primo registro per shift a sinistra e l'ultimo per shift verso destra. Infine, il flip-flop che prende il valore selezionato da questo multiplexer è necessario per avere in output l'ultimo bit uscente dallo shift register.

4.2.2 Codice

Multiplexer

I multiplexer sono stati realizzati generici con un approccio comportamentale, in modo tale che, con una sola implementazione, si avessero sia i multiplexer 4:1 che quelli 2:1. Inoltre, è possibile notare che il multiplexer posto in fondo alla figura 4.1 presenta 3 ingressi uguali, poiché sono state richieste 3 modalità che effettuano uno shift a sinistra, mentre il quarto ingresso è adibito a valutare il valore del registro durante lo shift verso destra.

```

1 entity mux is
2
3 generic (
4     n : positive := 2 -- Bit di ingresso al MUX per la selezione.  $2^n$ 
5         ingressi
6 );
7
8 port (
9     x : in std_logic_vector(2**n downto 1);
10    sel : in std_logic_vector(n-1 downto 0);
11    y : out std_logic
12 );

```

```

12
13 end mux;
14
15 architecture Behavioral of mux is
16
17 signal y_temp : std_logic := '0';
18
19 begin
20
21     y <= y_temp;
22
23     p : process(sel, x)
24
25         variable temp : integer;
26
27     begin
28         temp := to_integer(unsigned(sel));
29         y_temp <= x(2**n - temp);
30
31     end process;
32
33 end Behavioral;

```

Flip Flop

I Flip Flop di tipo D sono stati realizzati con un approccio comportamentale. Sono stati dotati di un ingresso *enable* così da salvare il dato sul fronte di discesa del clock solo quando enable è alto.

```

1 entity flip_flop is
2     port(
3         d      : in  std_logic;
4         clk    : in  std_logic;
5         reset : in  std_logic;
6         en     : in  std_logic;
7         q      : out std_logic
8     );
9 end flip_flop;
10
11 architecture Behavioral of flip_flop is
12
13 signal q_temp : std_logic := '0';
14
15 begin
16     q <= q_temp;
17
18     f: process(clk, reset)
19
20         begin

```

```

21      if(reset='1') then
22          q_temp <= '0';
23
24      elsif(clk'event and clk='0' and en='1') then
25          q_temp <= d;
26
27      end if;
28  end process;
29
30 end Behavioral;

```

Registro

All'interno del process è stato definito, tramite il costrutto switch case, il comportamento del registro a seconda delle quattro modalità, dove a seconda del valore di ingresso, salva il valore in y_temp sul fronte di discesa del clock. Inoltre è stato definito il segnale interno y_temp in modo da rendere l'assegnazione dell'uscita concorrente al processo.

```

1  entity registro is
2      port(
3          sh: in std_logic;
4          a: in std_logic_vector(3 downto 0);
5          x: in std_logic;
6          clk: in std_logic;
7          reset: in std_logic;
8          load: in std_logic;
9          m: in std_logic_vector(1 downto 0); --Segnale di modalita: 00 shf
10             sinistra, 01 shift destra, 10 circolare sinistra, 11 shift sinistra
11             con x
12          y: out std_logic
13      );
14  end registro;
15
16 architecture Structural of registro is
17
18     component mux is
19         generic(
20             n : positive
21         );
22
23         port(
24             x : in std_logic_vector(2**n downto 1);
25             sel : in std_logic_vector(n-1 downto 0);
26             y : out std_logic
27         );
28     end component;

```

```

29
30 component flip_flop is
31   port (
32     d: in std_logic;
33     clk: in std_logic;
34     reset: in std_logic;
35     en: in std_logic;
36     q: out std_logic
37   );
38 end component;
39
40 signal s_ff_out : std_logic_vector(3 downto 0);
41 signal s_ff_in : std_logic_vector(3 downto 0);
42 signal s_mux4_out : std_logic_vector(3 downto 0);
43 signal s_y: std_logic;
44
45 begin
46
47   mux_4_3 : mux
48
49     generic map(
50       n => 2
51     )
52
53
54     port map(
55       x(4) => s_ff_out(2),
56       x(3) => '0',
57       x(2) => s_ff_out(2),
58       x(1) => s_ff_out(2),
59       sel =>m,
60       y => s_mux4_out(3)
61     );
62
63   mux_4_2 : mux
64
65     generic map(
66       n => 2
67     )
68
69     port map(
70       x(4) => s_ff_out(1),
71       x(3) => s_ff_out(3),
72       x(2) => s_ff_out(1),
73       x(1) => s_ff_out(1),
74       sel =>m,
75       y => s_mux4_out(2)
76     );
77

```

```

78      mux_4_1 : mux
79
80      generic map(
81          n => 2
82      )
83
84      port map(
85          x(4) => s_ff_out(0),
86          x(3) => s_ff_out(2),
87          x(2) => s_ff_out(0),
88          x(1) => s_ff_out(0),
89          sel =>m,
90          y => s_mux4_out(1)
91      );
92
93
94      mux_4_0 : mux
95
96      generic map(
97          n => 2
98      )
99
100     port map(
101         x(4) => '0',
102         x(3) => s_ff_out(1),
103         x(2) => s_ff_out(3),
104         x(1) => x,
105         sel =>m,
106         y => s_mux4_out(0)
107     );
108
109
110    mux2_gen : for i in 0 to 3 generate
111
112        mux_2 : mux
113
114            generic map(
115                n => 1
116            )
117
118            port map(
119                x(2) => s_mux4_out(i),
120                x(1) => a(i),
121                sel(0) =>load,
122                y => s_ff_in(i)
123            );
124
125
126    end generate;

```

```

127
128      --MULTIPLEXER FINALE
129      mux_4_fin : mux
130
131      generic map(
132          n => 2
133      )
134
135      port map(
136          x(4) => s_ff_out(3),
137          x(3) => s_ff_out(0),
138          x(2) => s_ff_out(3),
139          x(1) => s_ff_out(3),
140          sel =>m,
141          y => s_y
142      );
143
144
145      ff_gen : for i in 0 to 3 generate
146
147          ff: flip_flop
148
149          port map(
150              d=> s_ff_in(i),
151              clk=> clk,
152              reset=> reset,
153              en=> sh OR load,
154              q=> s_ff_out(i)
155          );
156
157
158      end generate;
159
160      ff_buffer: flip_flop
161          port map(
162              d=> s_y,
163              clk=> clk,
164              reset=> reset,
165              en=> sh,
166              q=> y
167          );
168
169
170  end Structural;
171
172
173
174  architecture Behavioral of registro is
175

```

```

176 signal reg : std_logic_vector(3 downto 0) := (others => '0');
177 signal y_temp : std_logic := '0';
178
179 begin
180   y <=y_temp;
181   reg_scorrimento : process(clk,reset)
182
183     begin
184
185       if(reset='1') then
186         y_temp <= '0';
187         reg<= "0000";
188
189       elsif (clk'event and clk='0') then
190         if(load='1') then
191           reg<= a;
192         elsif(sh='1') then
193           case m is
194
195             --Scorrimento da destra verso sinistra
196             when "00" => y_temp<=reg(3);
197               reg(3 downto 1) <= reg(2 downto 0) ;
198               reg(0)<='0';
199
200             --Scorrimento da sinistra verso destra
201             when "01" => y_temp<=reg(0);
202               reg(2 downto 0) <= reg(3 downto 1) ;
203               reg(3)<='0';
204
205             --Scorrimento da circolare verso sinistra
206             when "10" => y_temp<=reg(3);
207               reg(3 downto 1) <= reg(2 downto 0) ;
208               reg(0)<=reg(3);
209
210             --Scorrimento da destra verso sinistra con x
211             when "11" => y_temp<=reg(3);
212               reg(3 downto 1) <= reg(2 downto 0) ;
213               reg(0)<=x;
214
215             when others => y_temp <= '0';
216               reg<= "0000";
217           end case;
218         end if;
219       end if;
220     end process;
221
222 end Behavioral;

```

4.3 Simulazione

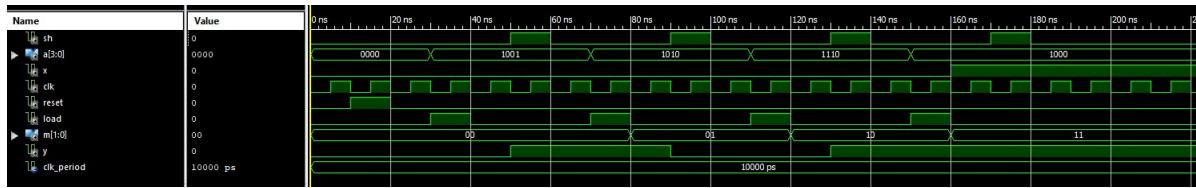


Figura 4.2: Simulazione del Registro a Scorrimento

Per effettuare la simulazione si è realizzato il testbench in figura 4.2. In questo caso, poiché era semplice valutare il corretto comportamento della macchina, non sono stati utilizzati gli assert. Sono state testate tutte le modalità richieste precaricando un valore di 4 bit diverso per ognuna di loro. Nella quarta modalità è stato fornito in ingresso anche il valore x che verrà poi salvato nell'ultimo registro a partire da sinistra.

```

1  wait for 10 ns;
2  reset <='1';
3  wait for 10 ns;
4  reset <='0';

5
6
7  --Prova Modalita 1
8  wait for 10 ns;
9  a<="1001";
10 load <='1';

11
12 wait for 10 ns;
13 load <='0';
14 --Scorrimento a sinistra
15 m<="00";

16
17 wait for 10 ns;
18 sh<='1';

19
20 wait for 10 ns;
21 sh<='0';

22
23 --Prova Modalita 2
24 wait for 10 ns;
25 a<="1010";
26 load <='1';

27
28 wait for 10 ns;
29 load <='0';
30 --Scorrimento a destra
31 m<="01";

32
33 wait for 10 ns;
```

```
34      sh<='1';
35
36      wait for 10 ns;
37      sh<='0';
38
39
40      --Prova Modalita 3
41      wait for 10 ns;
42      a<="1110";
43      load <='1';
44
45      wait for 10 ns;
46      load <='0';
47      --Scorrimento circolare a sinistra
48      m<="10";
49
50      wait for 10 ns;
51      sh<='1';
52
53      wait for 10 ns;
54      sh<='0';
55
56
57      --Prova Modalita 4
58      wait for 10 ns;
59      a<="1000";
60      load <='1';
61
62
63      wait for 10 ns;
64      load <='0';
65      x<='1';
66      --Scorrimento a sinistra con x
67      m<="11";
68
69      wait for 10 ns;
70      sh<='1';
71
72      wait for 10 ns;
73      sh<='0';
74
75      wait for 10 ns;
76
77      wait;
78  end process;
79
80 END;
```

Capitolo 5

Operazione di Modulo

5.1 Traccia

Progettare e implementare in VHDL un sistema che, date due stringhe binarie A e B di 8 bit ciascuna acquisite mediante handshaking, calcoli il valore A mod B. Il sistema deve essere progettato utilizzando un approccio modulare basato sull'individuazione della parte operativa e della parte di controllo, e la parte di controllo deve essere realizzata mediante (a) logica cablata e (b) logica microprogrammata. Con riferimento alle modalità di acquisizione delle stringhe in input mediante handshaking, si discutano due diverse soluzioni possibili.

5.2 Soluzione

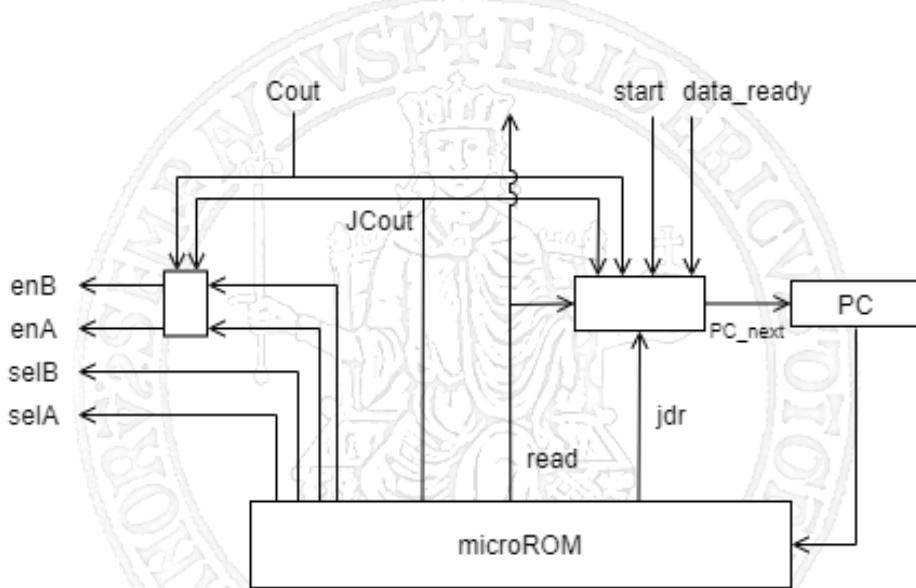


Figura 5.1: Struttura generale dell'Unità di Controllo

Come richiesto dalla traccia, è stato utilizzato un approccio modulare per realizzare il sistema, in particolare sono state progettate unità operativa, unità di controllo e una memoria. L'unità operativa, la cui struttura è esposta in seguito, è la parte adibita per effettuare l'operazione A mod B. L'unità di controllo, invece, gestisce il sistema per l'acquisizione delle stringhe mediante

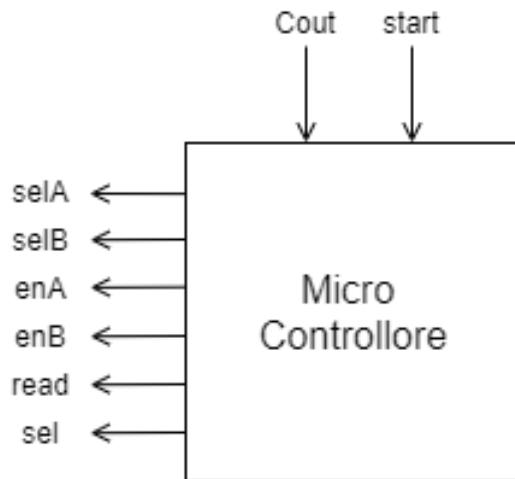


Figura 5.2: Interfaccia dell'Unità di Controllo

handshaking asincrono da una memoria esterna, costituita da 4 locazione di 7 bit ciascuna, ed è stata implementata sia in logica cablata sia in logica microprogrammata. In figura 5.2 è raffigurata l'interfaccia dell'unità di controllo. *Read* e *data_ready* rappresentano i segnali necessari per un corretto handshaking asincrono.

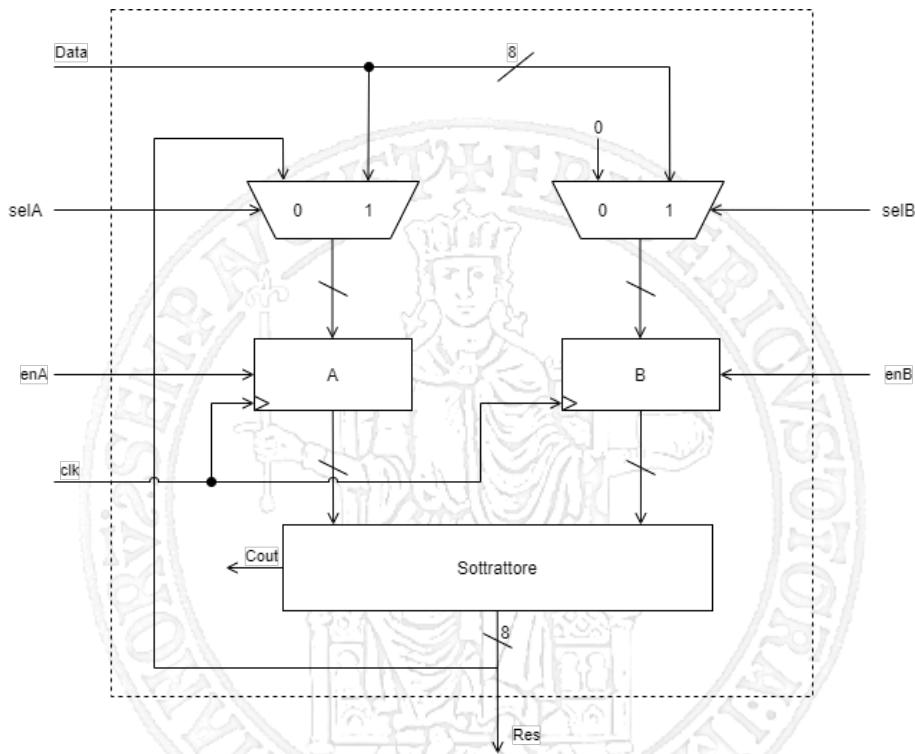


Figura 5.3: Struttura dell'Unità Operativa

Sia se esso è implementato in logica cablata sia in logica microprogrammata, l'interfaccia rimane invariata come il suo comportamento esterno. In figura 5.4 è rappresentato l'automa progettato, implementato in entrambe le logiche.

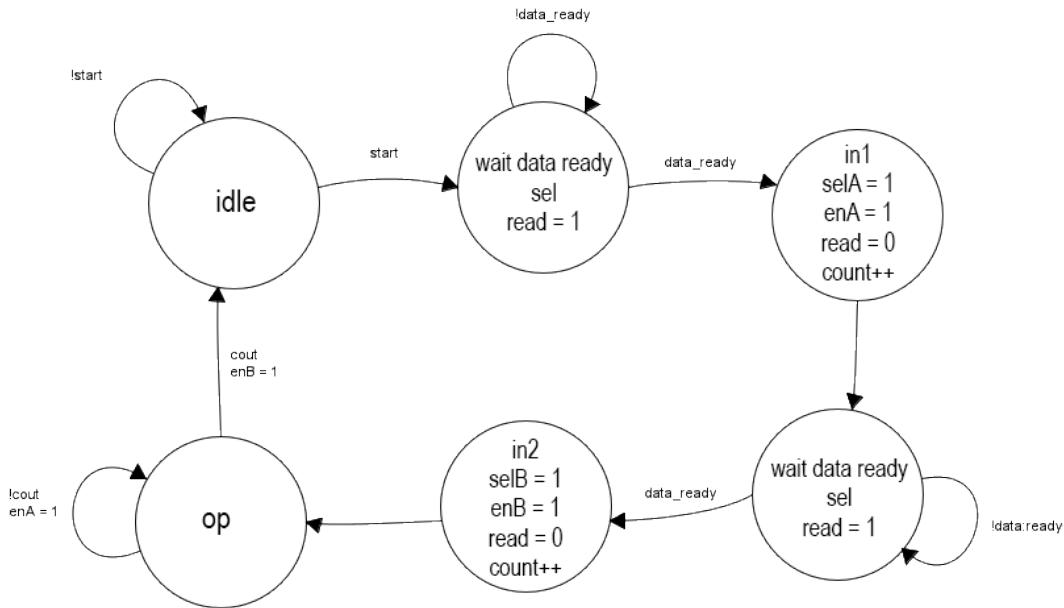


Figura 5.4: Grafo dell'automa dell'Unità di Controllo

5.2.1 Handshaking

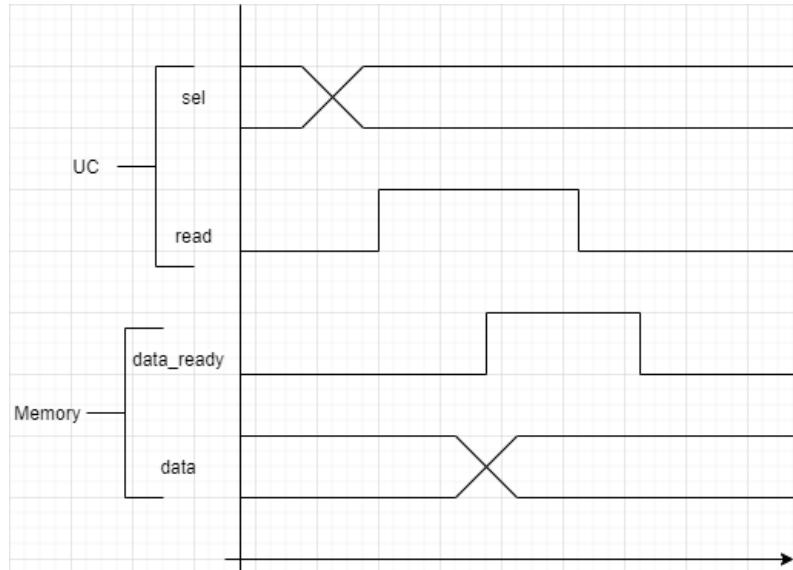


Figura 5.5: Protocollo di comunicazione tra unità di controllo e memoria

Nella realizzazione del progetto è stato deciso di implementare un protocollo di tipo *asincrono* per la comunicazione tra l'unità di controllo e la memoria, in modo tale da non fare alcuna assunzione sui tempi di lettura dalla memoria. In caso si voglia sostituire il protocollo asincrono con uno sincrono, bisognerebbe modificare l'unità di controllo lasciando inalterata l'unità operativa. In figura 5.5 sono rappresentati i segnali che implementano l'handshaking su un diagramma temporale.

5.2.2 Control Store

	PC_NEXT		JCout	JDR	enA	enB	selA	selB	read
idle	0	0	0	0	0	0	0	0	0
in1	0	1	0	0	0	1	0	1	0
mem_ack1	0	1	1	0	1	0	0	0	0
in2	1	0	0	0	0	0	1	0	1
mem_ack2	1	0	1	0	1	0	0	0	0
op	1	0	1	1	0	1	1	0	0

Figura 5.6: Control Store per la gestione dell'Unità Operativa

La tabella in figura 5.6 rappresenta il microprogramma all'interno della *MicroRom*. I primi 3 bit delle 6 control words rappresentano il segnale *pc_next*, che definisce il prossimo stato del controllo. I successivi 2 bit, *Jcout* e *JDR*, sono segnali di controllo di flusso che consentono di determinare in maniera condizionata lo stato prossimo in funzione rispettivamente dei segnali *Cout* e *Data_Ready* in ingresso al microcontrollore. I bit di enable e di selection sono collegati direttamente all'unità operativa come segnali di controllo per eseguire l'operazione di modulo. Infine, il bit *read* è sia un segnale di flusso che segnale di handshaking utilizzato per avviare la lettura in memoria. Rispetto agli stati indicati nel materiale didattico per l'implementazione dell'operazione modulo, sono stati introdotti due ulteriori stati, *mem_ack1* e *mem_ack2*, che hanno la funzione di attendere che il segnale di *data_ready* si abbassi.

5.2.3 Codice

Memoria

Gli operandi su cui lavora il programma sono salvati in una memoria implementata con un approccio comportamentale. A questo scopo, è stato definito un vettore di interi di dimensione 4.

```

1 entity Memoria is
2   port(
3     sel : in std_logic_vector(1 downto 0);
4     read1 : in std_logic;
5     data : out std_logic_vector(7 downto 0);
6     data_ready : out std_logic
7   );
8
9
10
11 end Memoria;
```

```

12
13 architecture Behavioral of Memoria is
14 SUBTYPE data_type is integer range 0 to 255;
15 TYPE memory_type is array(0 to 3) of data_type;
16 constant memory : memory_type := (
17     0 => 13,
18     1 => 3,
19     2 => 9,
20     3 => 5
21 );
22
23 begin
24     mem : process
25         variable index : integer;
26         begin
27             WAIT UNTIL read1 = '1';
28             index := to_integer(unsigned(sel));
29             data <= std_logic_vector(to_unsigned(memory(index), data'length));
30             data_ready <= '1';
31             WAIT UNTIL read1 = '0';
32             data_ready <= '0';
33         end process;
34     end Behavioral;

```

MicroRom

Il modulo MicroRom implementa la *control store* in figura 5.6 ed è costituito da 6 *control words*:

- *Idle* rappresenta lo stato di attesa del segnale *start* che avvia l'inizio dell'operazione di modulo.
- Negli stati *in* l'unità di controllo genera sia i segnali per la richiesta dei dati dalla memoria esterna sia per il caricamento degli stessi nei registri dell'unità operativa.
- Gli stati *mem_ack* consentono di generare il segnale di controllo *JDR* per l'attesa di *data_ready* = 0, così da terminare correttamente il protocollo di comunicazione tra l'unità di controllo e la memoria.
- Durante lo stato di *op* sono generati i segnali per effettuare l'operazione di modulo, fin quando *cout* è basso.

```

1 entity MicroRom is
2     port(
3         PC : in unsigned(2 downto 0);
4         pc_next : out unsigned (2 downto 0);
5         jcout : out std_logic;
6         enA : out std_logic;
7         enB : out std_logic;

```

```

8      selA : out std_logic;
9      selB : out std_logic;
10     read1 : out std_logic
11   );
12 end MicroRom;
13
14 architecture Synth of MicroRom is
15
16 TYPE control_word_type is record
17     pc_next : unsigned (2 downto 0);
18     jcout : std_logic;
19     enA : std_logic;
20     enB : std_logic;
21     selA : std_logic;
22     selB : std_logic;
23     read1 : std_logic;
24 end record;
25
26 constant idle : control_word_type := (
27     pc_next => "000",
28     jcout => '0',
29     enA => '0',
30     enB => '0',
31     selA => '0',
32     selB => '0',
33     read1 => '0'
34 );
35
36 constant in1 : control_word_type := (
37     pc_next => "010",
38     jcout => '0',
39     enA => '1',
40     enB => '0',
41     selA => '1',
42     selB => '0',
43     read1 => '1'
44 );
45
46 constant memory_ack : control_word_type := (
47     pc_next => "011",
48     jcout => '0',
49     enA => '0',
50     enB => '0',
51     selA => '0',
52     selB => '0',
53     read1 => '0'
54 );
55
56 constant in2 : control_word_type := (

```

```

57      pc_next => "100",
58      jcout => '0',
59      enA => '0',
60      enB => '1',
61      selA => '0',
62      selB => '1',
63      read1 => '1'
64    );
65
66 constant op : control_word_type := (
67     pc_next => "100",
68     jcout => '1',
69     enA => '1',
70     enB => '1',
71     selA => '0',
72     selB => '0',
73     read1 => '0'
74   );
75
76
77 type rom_type is array(0 to 4) of control_word_type;
78
79 constant rom : rom_type := (
80     0 => idle,
81     1 => in1,
82     2 => memory_ack,
83     3 => in2,
84     4 => op
85 );
86
87 signal s_control : control_word_type;
88
89 begin
90   s_control <= rom(to_integer(pc));
91
92   pc_next <= s_control.pc_next;
93   jcout <= s_control.jcout;
94   enA <= s_control.enA;
95   enB <= s_control.enB;
96   selA <= s_control.selA;
97   selB <= s_control.selB;
98   read1 <= s_control.read1;
99
100 end Synth;

```

Unità Operativa

L'unità operativa è la componente che esegue l'operazione di modulo ed è pilotata attraverso i segnali generati dall'unità di controllo. È stata realizzata con un approccio comportamentale basandosi sul grafico in figura 5.3.

```

1  entity Unita_Operativa is
2      port (
3          clk : in std_logic;
4          data : in std_logic_vector(7 downto 0);
5          selA : in std_logic;
6          selB : in std_logic;
7          enA : in std_logic;
8          enB : in std_logic;
9          cout : out std_logic;
10         res : out std_logic_vector(7 downto 0)
11     );
12
13 end Unita_Operativa;
14
15 architecture Behavioral of Unita_Operativa is
16 signal reg_A, reg_B : std_logic_vector(7 downto 0) := (others => '0');
17 signal res_temp : std_logic_vector(7 downto 0) := (others => '0');
18
19 begin
20     res <= res_temp;
21
22     mood : process(clk)
23     begin
24         if (clk'event and clk = '1') then
25             if(enA = '1') then
26                 case selA is
27                     when '0' => reg_A <= res_temp;
28                     when '1' => reg_A <= data;
29                     when others => reg_A <= (others => '0');
30                 end case;
31
32             elsif(enB = '1') then
33                 case selB is
34                     when '0' => reg_B <= (others => '0');
35                     when '1' => reg_B <= data;
36                     when others => reg_B <= (others => '0');
37                 end case;
38
39             end if;
40         end if;
41     end process;
42
43     sub : process(reg_A, reg_B)

```

```

45 begin
46     if(reg_A < reg_B) then
47         cout <= '1';
48     else
49         cout <= '0';
50     end if;
51
52     res_temp <= std_logic_vector(unsigned(reg_A) - unsigned(reg_B));
53
54 end process;
55
56 end Behavioral;

```

Unità di Controllo

L'unità di controllo è stata realizzata sia in logica cablata, tramite un approccio behavioral con due processi, sia microprogrammata con un approccio ibrido.

```

1 entity Unità_Controllo is
2     port (
3         clk : in std_logic;
4         data_ready : in std_logic;
5         start : in std_logic;
6         cout : in std_logic;
7         read1 : out std_logic;
8         sel : out std_logic_vector(1 downto 0);      -- Selezione dalla
9             memoria
10        selA : out std_logic;
11        selB : out std_logic;
12        enA : out std_logic;
13        enB : out std_logic
14    );
15 end Unità_Controllo;
16
17 architecture Behavioral of Unità_Controllo is
18     TYPE State is (idle, in1, in2, op);
19     signal current_state, next_state : state;
20
21 begin
22     f_state:process(clk)
23     begin
24         if (clk'event and clk = '1') then
25             if(start = '1') then
26                 current_state <= in1;
27             else
28                 current_state <= next_state;
29             end if;

```

```

30      end if;
31  end process;
32
33 f_comb:process
34 variable count : unsigned(1 downto 0) := (others => '0');
35 begin
36     WAIT ON current_state, cout;
37     selA <= '0';
38     selB <= '0';
39     enA <= '0';
40     enB <= '0';
41     read1 <= '0';
42     next_state <= idle;
43
44 case current_state is
45
46     when idle =>
47         next_state<=idle;
48
49     when in1 =>
50         sel <= std_logic_vector(count);
51         read1 <= '1';
52         WAIT UNTIL data_ready = '1';
53         selA <= '1';
54         enA <= '1';
55         read1 <= '0';
56         count := count + 1;
57         next_state <= in2;
58
59     when in2 =>
60         sel <= std_logic_vector(count);
61         read1 <= '1';
62         WAIT UNTIL data_ready = '1';
63         selB <= '1';
64         enB <= '1';
65         read1 <= '0';
66         count := count + 1;
67         next_state <= op;
68
69     when op =>
70         if(cout = '0') then
71             enA <= '1';
72             next_state <= op;
73         else
74             enB <= '1';
75             next_state <= idle;
76         end if;
77     end case;
78 end process;

```

```

79
80
81
82 end Behavioral;
83
84
85 architecture MicroProg of Unita_Controllo is
86
87 component MicroRom is
88     port (
89         PC : in unsigned(2 downto 0);
90         pc_next : out unsigned (2 downto 0);
91         jcout : out std_logic;
92         enA : out std_logic;
93         enB : out std_logic;
94         selA : out std_logic;
95         selB : out std_logic;
96         read1 : out std_logic
97     );
98 end component;
99
100 signal s_pc_next, s_pc : unsigned(2 downto 0);
101 signal s_enA, s_enB, s_jcout : std_logic;
102 signal counter : unsigned(1 downto 0) := "00";
103
104 begin
105
106     mr : MicroRom port map(
107         PC => s_pc,
108         pc_next => s_pc_next,
109         jcout => s_jcout,
110         enA => s_enA,
111         enB => s_enB,
112         selA => selA,
113         selB => selB,
114         read1 => read1
115     );
116
117     count_inc : process(data_ready)
118     begin
119         if(data_ready = '1') then
120             counter <= counter + 1;
121         end if;
122     end process;
123
124     reg_pc : process(clk)
125     begin
126         if(clk'event and clk = '1') then
127             if(start = '1') then

```

```
128      s_pc <= "001";
129      elsif(s_jcout = '0') then
130          s_pc <= s_pc_next;
131      else
132          if(cout = '1') then
133              s_pc <= "000";
134          end if;
135      end if;
136  end if;
137 end process;
138
139
140 enA <= s_enA when s_jcout = '0' else not(cout);
141 enB <= s_enB when s_jcout = '0' else cout;
142 sel <= std_logic_vector(counter);
143 end MicroProg;
```



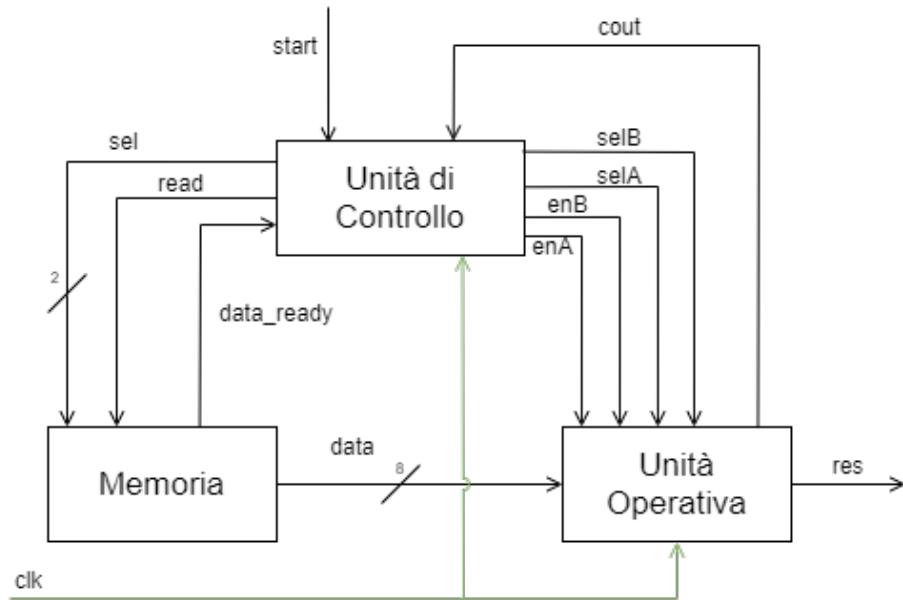


Figura 5.7: Architettura generale del progetto Modulo

Modulo

Modulo collega tutte le componenti definite in figura 5.7 con un approccio strutturale.

```

1  entity Modulo is
2      port (
3          clk : in std_logic;
4          start : in std_logic;
5          y : out std_logic_vector(7 downto 0)
6      );
7
8  end Modulo;
9
10 architecture Structural of Modulo is
11
12 component Memoria is
13     port (
14         sel : in std_logic_vector(1 downto 0);
15         read1 : in std_logic;
16         data : out std_logic_vector(7 downto 0);
17         data_ready : out std_logic
18     );
19 end component;
20
21 component Unità_Controllo is
22     port (
23         clk : in std_logic;
24         data_ready : in std_logic;
25         start : in std_logic;

```

```

26     cout : in std_logic;
27     read1 : out std_logic;
28     sel : out std_logic_vector(1 downto 0);      -- Seleziona dalla
29         memoria
30     selA : out std_logic;
31     selB : out std_logic;
32     enA : out std_logic;
33     enB : out std_logic
34 );
35 end component;
36 FOR ALL : Unita_Controllo USE ENTITY WORK.Unita_Controllo(MicroProg);
37
38 component Unita_Operativa is
39     port(
40         clk : in std_logic;
41         data : in std_logic_vector(7 downto 0);
42         selA : in std_logic;
43         selB : in std_logic;
44         enA : in std_logic;
45         enB : in std_logic;
46         cout : out std_logic;
47         res : out std_logic_vector(7 downto 0)
48 );
49 end component;
50
51 signal s_selA, s_selB, s_enA, s_enB, s_read1, s_data_ready, s_cout:
52     std_logic;
53 signal s_data : std_logic_vector(7 downto 0);
54 signal s_sel : std_logic_vector(1 downto 0);
55
56 begin
57     mem: memoria
58         port map(
59             sel => s_sel,
60             read1 => s_read1,
61             data => s_data,
62             data_ready => s_data_ready
63 );
64
65     uc : Unita_controllo
66         port map(
67             clk => clk,
68             data_ready => s_data_ready,
69             start => start,
70             cout => s_cout,
71             read1 => s_read1,
72

```

```
73     sel => s_sel,
74     selA => s_selA,
75     selB => s_selB,
76     enA => s_enA,
77     enB => s_enB
78 );
79
80
81 uo : Unita_Operativa
82     port map(
83         clk => clk,
84         data => s_data,
85         selA => s_selA,
86         selB => s_selB,
87         enA => s_enA,
88         enB => s_enB,
89         cout => s_cout,
90         res => y
91 );
92
93
94 end Structural;
```



5.3 Simulazione

Per effettuare la simulazione di entrambi gli approcci implementati, si è realizzato il testbench riportato qui di seguito. In questo caso, poiché era semplice valutare il corretto comportamento della macchina, non sono stati utilizzati gli assert. Avendo precaricato 4 valori in memoria, è stata avviata la macchina tramite il segnale di start per poterne valutare il corretto funzionamento.

Primo caso - 13 mod 3 = 1



Figura 5.8: Simulazione 13 mod 3

Da questa simulazione si può notare l'operazione modulo svolta dalla macchina. A 125ns il risultato dell'ALU è negativo ed a 135ns viene ripristinato il valore precedente, che corrisponde al risultato finale.

Secondo caso - 9 mod 5 = 4



Figura 5.9: Simulazione 9 mod 5

A 205ns il risultato dell'ALU è negativo ed a 215ns viene ripristinato il valore precedente, che corrisponde al risultato finale.

```

1 ENTITY modulo_tb IS
2 END modulo_tb;
3
4 ARCHITECTURE behavior OF modulo_tb IS
5
6   -- Component Declaration for the Unit Under Test (UUT)
7
8   COMPONENT Modulo
9     PORT (
10       clk : IN std_logic;
11       start : IN std_logic;
12       y : OUT std_logic_vector(7 downto 0)
13     );

```

```

14    END COMPONENT;
15
16    --Inputs
17    signal clk : std_logic := '0';
18    signal start : std_logic := '0';
19
20    --Outputs
21    signal y : std_logic_vector(7 downto 0);
22
23    -- Clock period definitions
24    constant clk_period : time := 10 ns;
25
26 BEGIN
27
28    -- Instantiate the Unit Under Test (UUT)
29    uut: Modulo PORT MAP (
30        clk => clk,
31        start => start,
32        y => y
33    );
34
35    -- Clock process definitions
36    clk_process :process
37    begin
38        clk <= '0';
39        wait for clk_period/2;
40        clk <= '1';
41        wait for clk_period/2;
42    end process;
43
44
45    -- Stimulus process
46    stim_proc: process
47    begin
48        -- hold reset state for 100 ns.
49        wait for 30 ns;
50        start <= '1';
51        wait for 30 ns;
52        start <= '0';
53
54        wait for 50 ns;
55
56        wait for 30 ns;
57        start <= '1';
58        wait for 30 ns;
59        start <= '0';
60
61        wait for 50 ns;
62

```

```
63      wait for 30 ns;  
64      start <= '1';  
65      wait for 30 ns;  
66      start <= '0';  
67  
68  
69  
70      -- insert stimulus here  
71  
72      wait;  
73  end process;  
74  
75 END;
```



Capitolo 6

Comunicazione tra sistemi mediante buffer

6.1 Traccia

Progettare un sistema per inviare due parole da 16 bit ciascuna da una unità A ad una unità B. Le due unità non sono dotate di un collegamento diretto costituito da un bus di 16 bit, ma l'unità A possiede un bus di 4 bit in uscita e l'unità B possiede un bus di 8 bit in ingresso. Per questa ragione, il trasferimento deve avvenire in più passi facendo uso di un buffer.

A questo scopo, una unità di controllo si occupa di trasferire ognuna delle due parole di 16 bit da A al buffer in 4 blocchi successivi da 4 bit ciascuno, e successivamente dal buffer a B in 2 blocchi successivi da 8 bit ciascuno.

6.2 Soluzione

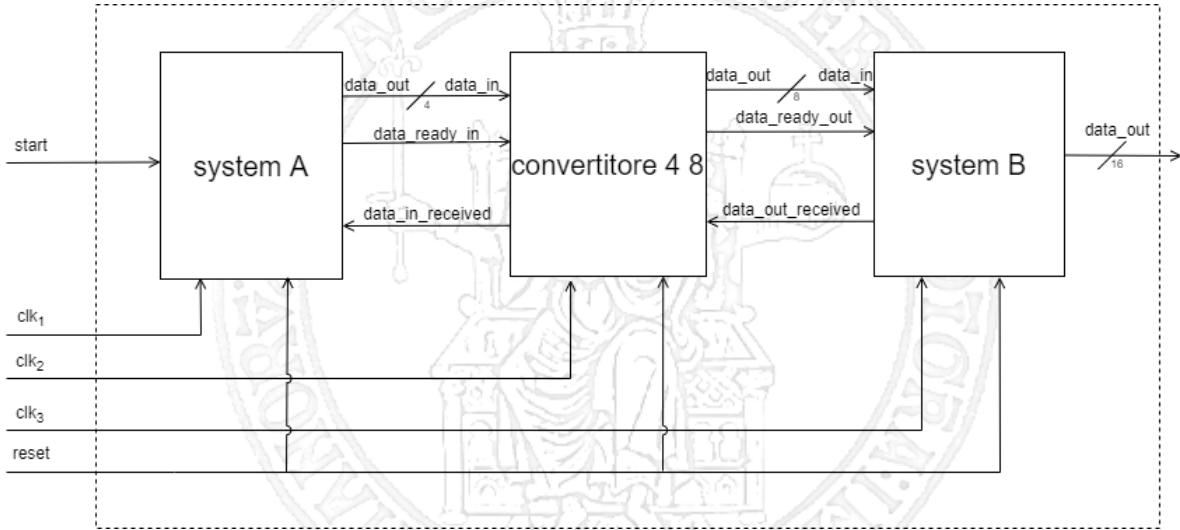


Figura 6.1: Architettura generale del sistema

Il sistema generale è composto da tre moduli: *System A*, *System B* e *Convertitore 4/8*.

Il *Convertitore 4/8* è stato inserito per permettere la comunicazione tra il sistema A e il sistema B, poiché la dimensione del bus di uscita del primo sistema è diversa rispetto al bus di ingresso

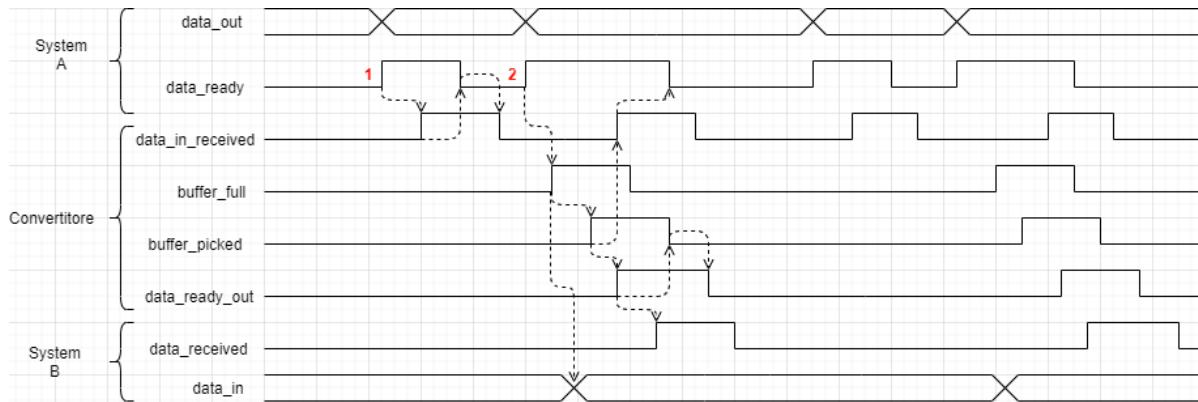


Figura 6.2: Protocollo di comunicazione tra i sistemi ed il buffer

dal secondo.

Per la corretta comunicazione, è stato progettato un protocollo di handshaking asincrono introducendo opportuni segnali di stato tra i sistemi ed il convertitore. In figura 6.2 è mostrato il diagramma temporale dei segnali generati per l'invio di 16 bit dal sistema A al sistema B. Come si può osservare dallo schema, il sistema A genera il segnale *data_ready* dopo aver trasferito i primi 4 bit sul segnale *data_out*.

Successivamente, il convertitore, dopo aver caricato tali bit nel buffer da 8 bit, genera il segnale di ack *data_in_received*.

Al trasferimento del secondo blocco di 4 bit dal sistema A, prima di generare nuovamente l'*ack*, il convertitore *svuota* il buffer caricando i dati sul bus di uscita verso il sistema B ed è pronto per ricevere nuovi dati. Il segnale *data_ready_out* notifica al sistema B la presenza di dati da leggere sul bus. Dopo averli letti, il sistema B invia un *ack* al convertitore con *data_received*.

Questa procedura si ripete ogni volta che si chiede il trasferimento di 8 bit da A a B, ossia due volte per ogni word.

Ogni componente del sistema realizza un automa a stati finiti, opportunatamente progettati per implementare la comunicazione appena descritta.

Infine, nello schema in figura 6.1 sono rappresentate le connessioni delle componenti appena descritte. È da notare che *System A*, *System B* e *Convertitore 4/8*, in linea di principio, possono lavorare su clock con frequenze diverse poiché il protocollo di comunicazione implementato è asincrono.

6.2.1 System A

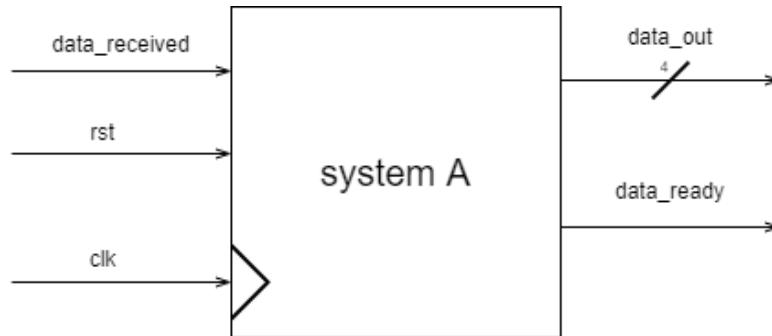


Figura 6.3: Interfaccia del sistema A

Il sistema A presenta 3 ingressi: *data_received*, *reset* e *clock*. All'interno della macchina sono state memorizzate le due word da inviare al sistema B.

In uscita il sistema presenta il segnale *data_ready* e il bus da 4 bit *data_out* che si interfacciano con il convertitore. I segnali *data_ready* e *data_received*, come mostrato in precedenza, sono necessari per l'handshaking asincrono, notificando rispettivamente la presenza dei dati disponibili al convertitore e l'avvenuta lettura di quest'ultimi.

Il sistema A presenta i seguenti stati di evoluzione:

WAIT START Stato di idle in cui si attende il segnale di start per iniziare l'operazione di trasferimento dati verso il convertitore.

SendData Vengono caricati i dati sul bus di uscita e viene alzato il segnale *DATA_READY* per notificare al convertitore la presenza di dati consistenti in ingresso.

WAIT DataReceived Il sistema si pone in attesa del segnale *DATA_RECEIVED* che notifica l'avvenuta ricezione dei dati da parte del convertitore.

DataReceived In questo stato il sistema abbassa *DATA_READY* e attende che il convertitore abbassi a sua volta il segnale *DATA_RECEIVED*. Se sono stati inviati tutti i 16 bit, ovvero l'operazione è stata eseguita 4 volte, il sistema ritorna in **WAIT START** se *start* = 0, altrimenti il sistema evolve nello stato **WAIT START = 0**; se invece vi sono altri dati da inviare, il sistema si pone nello stato **SendData**.

WAIT START = 0 Si attende *START* = 0 prima di ritornare nello stato di idle per evitare comunicazioni spurie.

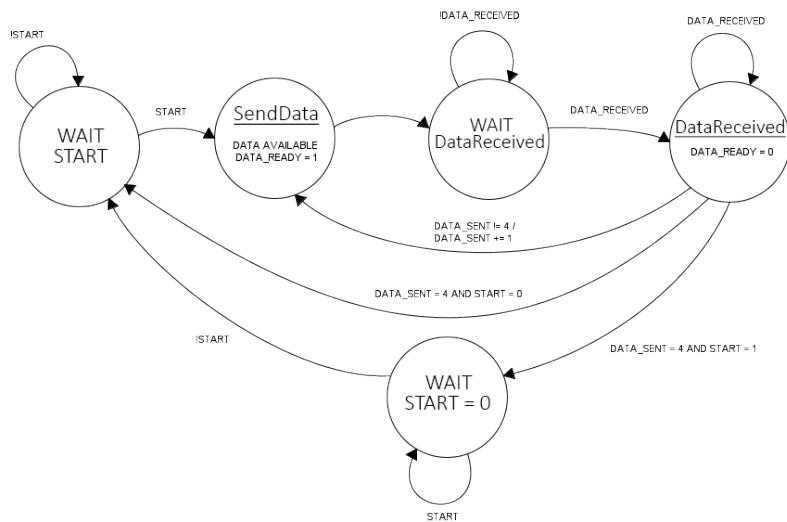


Figura 6.4: Grafo dell'automa del sistema A

6.2.2 System B

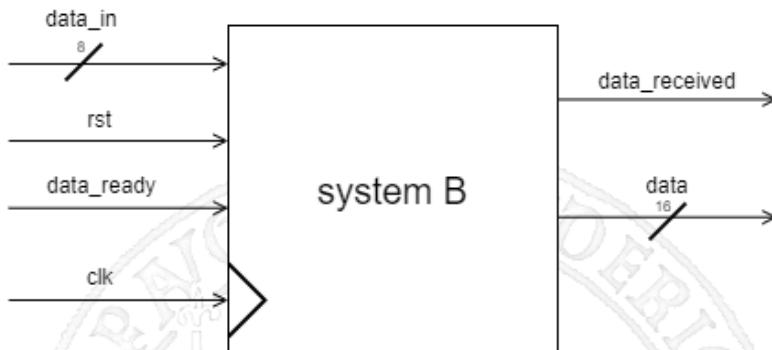


Figura 6.5: Interfaccia del sistema B

Il sistema B presenta 4 ingressi: *data_in*, *reset*, *data_ready* e *clock*.

In uscita il sistema presenta il segnale *data_received* e il bus *data* da 16 bit.

I segnali *data_ready* e *data_received*, sono necessari per l'handshaking asincrono e notificano rispettivamente la presenza dei dati disponibili dal convertitore e l'avvenuta lettura di questi da parte del sistema stesso.

L'uscita *data* da 16 bit rappresenta l'intera word inviata dal sistema A.

Il sistema B presenta i seguenti stati di evoluzione:

WAIT DataReady Il sistema si pone in attesa del segnale DATA_READY in uscita al convertitore che notifica la presenza di dati che possono essere caricati.

DataReceived In questo stato il sistema salva i dati provenienti dal convertitore. Alla prima ricezione, i dati vengono salvati negli 8 bit più significativi di un registro interno (*sel* = 0); nel secondo caso nei restanti 8 bit (*sel* = 1).

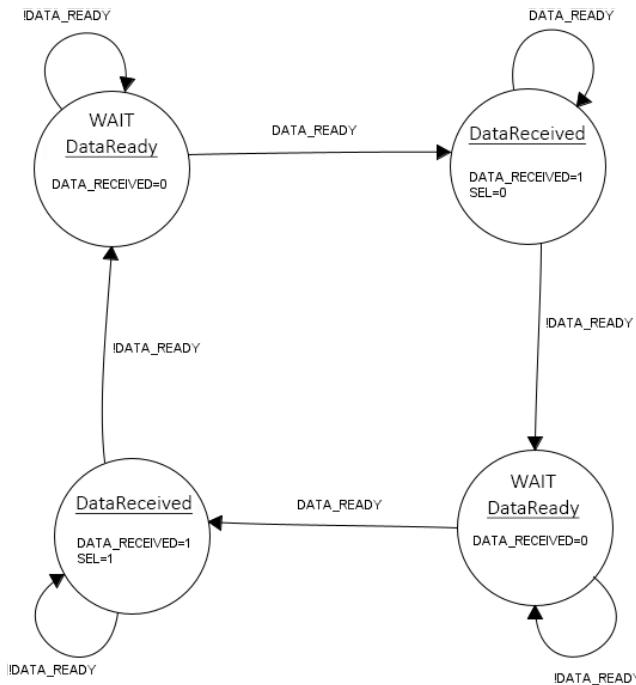


Figura 6.6: Grafo dell'automa del sistema B

6.2.3 Convertitore 4_8

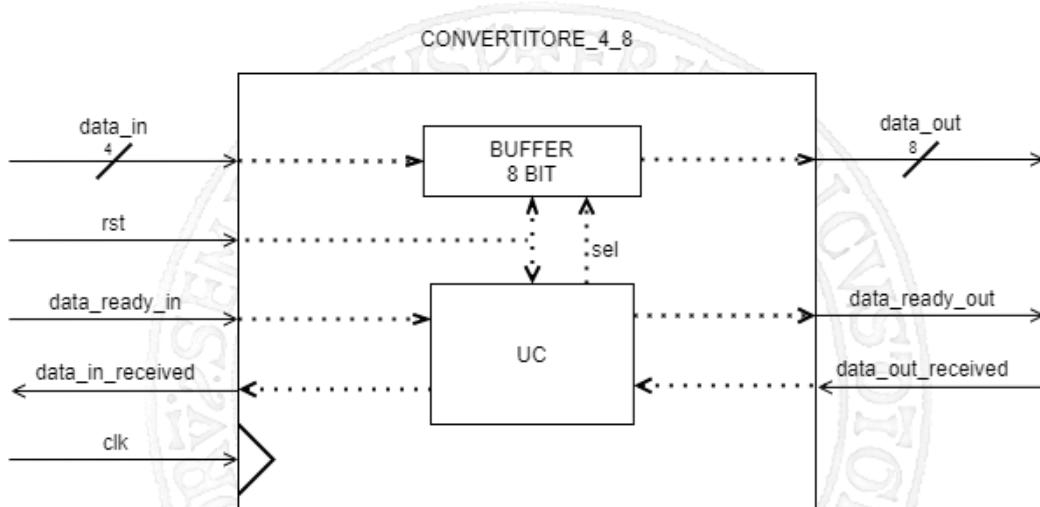


Figura 6.7: Interfaccia del convertitore 4_8

Il convertitore_4_8 racchiude tutti i segnali adibiti all'handshaking visti in precedenza, come mostrato in figura 6.7. All'interno del sistema è possibile distinguere due componenti fondamentali: l'unità di controllo e un registro buffer da 8 bit.

L'unità di controllo comunica al buffer in quali locazioni di memoria salvare i bit in ingresso (*data_in*), inserendoli prima nei 4 bit più significativi e poi nei 4 meno significativi.

A sua volta, l'unità di controllo si divide in due componenti, una che si occupa di ricevere dati dal

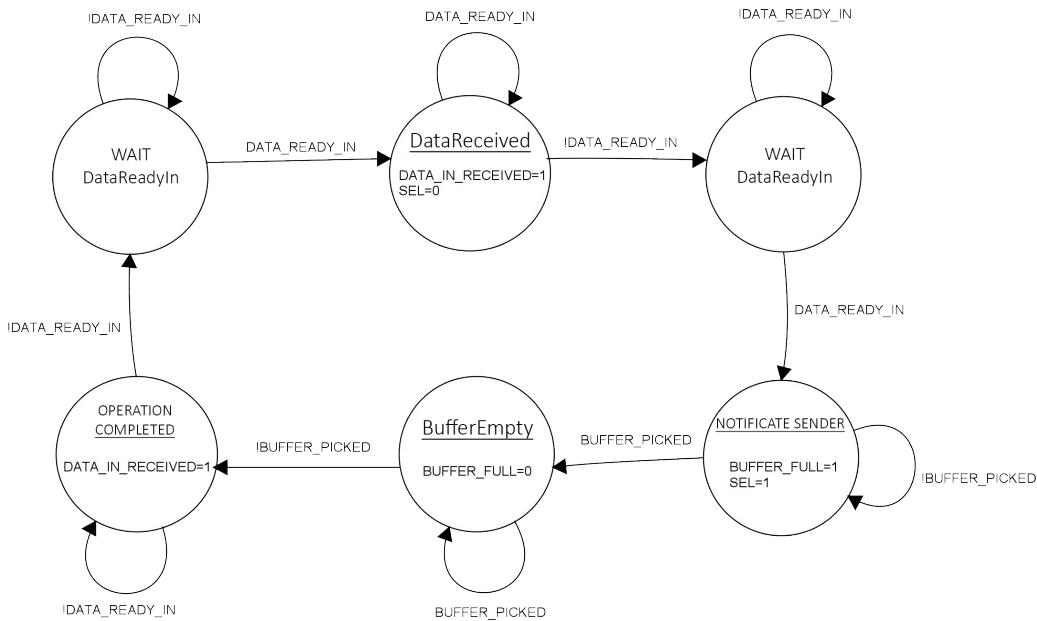


Figura 6.8: Grafo dell'automa del *receiver* del convertitore

sistema A, il *receiver*, ed una che si occupa di inviarli al sistema B, il *sender*.

Per quanto concerne il *receiver*, esso evolve secondo gli stati di evoluzione sottoesposti.

WAIT DataReadyIn Stato di attesa in cui il *receiver* aspetta che il sistema A gli notifichi la presenza di dati sul bus d'ingresso.

DataReceived Il *receiver* carica i primi 4 bit di dati più significativi ($sel = 0$) in un registro interno e informa il sistema A di averli acquisiti.

NOTIFICATE SENDER Vengono salvati gli altri 4 bit meno significativi ($sel = 1$) e viene notificato al *sender* che il buffer è pieno. Si attende in questo stato finché il *sender* non svuoti il buffer.

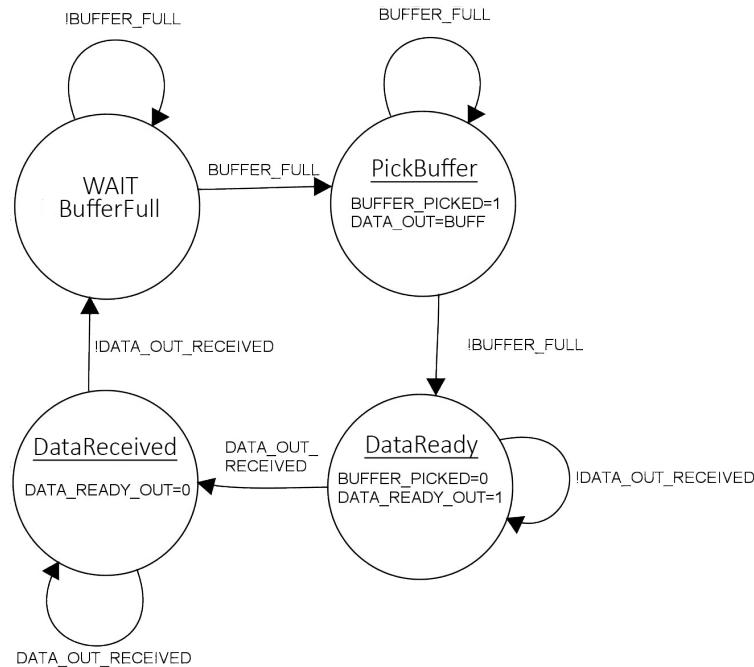
WAIT BufferEmpty Il buffer è stato svuotato e si pone $buffer_full = 0$ e si attende che si abbassi $buffer_picked$.

OPERATION COMPLETED Si notifica al sistema A che sono stati acquisiti gli altri 4 bit e si attende che $DATA_READY_IN$ si abbassi. Si noti che la notifica dell'acquisizione dei bit meno significativi avviene dopo che il buffer è stato svuotato e non all'atto della loro effettiva acquisizione, così da impedire al sistema A di inviare ulteriori bit quando il buffer è pieno.

Il *sender*, invece, evolve secondo i seguenti stati:

WAIT BufferFull Il *sender* attende che il buffer del convertitore contenga 8 bit da poter inviare al sistema B.

PickBuffer Stato in cui si pone il valore del buffer in uscita al convertitore e si avvisa il *receiver* di aver svuotato il registro interno e, di conseguenza, di poter acquisire ulteriori dati.


 Figura 6.9: Grafo dell'automa del *sender* del convertitore

DataReady Il *sender* comunica al sistema B che i dati sono stati caricati sul bus in uscita e si attende in questo stato finché non vengono acquisiti, ovvero quando $DATA_READY_OUT = 1$.

DataReceived Si attende in questo stato fino a quando $DATA_OUT_RECEIVED = 0$, così da concludere il protocollo di comunicazione con il sistema B.

6.2.4 Codice

Nella seguente implementazione sono stati utilizzati i costrutti *wait on* e *wait until* per il corretto funzionamento del protocollo di handshaking asincrono in fase di simulazione. Successivamente, affinché il sistema potesse essere sintetizzato sulla board, tali costrutti sono stati sostituiti con degli stati aggiuntivi. Il codice qui riportato è quello che presenta i costrutti di attesa.

System A

Il sistema A è stato implementato con un approccio comportamentale facendo riferimento all'automa rappresentato in figura 6.4. Al suo interno sono memorizzate le due word da 16 bit da inviare al sistema B. Il bus in uscita è da 4 bit.

```

1 entity system_a is
2 port (
3     clk      : in  std_logic;
4     data_received : in  std_logic;
5     rst      : in  std_logic;
6     start    : in  std_logic;
```

```

7      data_out      : out std_logic_vector(3 downto 0);
8      data_ready    : out std_logic
9  );
10 end system_a;
11
12 architecture Behavioral of system_a is
13
14
15 constant word_1 : std_logic_vector(15 downto 0) := "1111000000001111";
16 constant word_2 : std_logic_vector(15 downto 0) := "000011111110000";
17
18 type ROM_type is ARRAY(0 to 1) of std_logic_vector(15 downto 0);
19
20 constant ROM : ROM_type := (
21   0 => word_1,
22   1 => word_2
23 );
24
25
26 type state is (S0, S1, S2, S3, S4);
27 signal current_state : state := S0;
28
29
30 signal data_out_temp : std_logic_vector(3 downto 0);
31 signal data_ready_temp : std_logic := '0';
32
33
34 begin
35
36
37 data_out <= data_out_temp;
38 data_ready <= data_ready_temp;
39
40
41 p : process
42
43
44 variable count_word : unsigned(0 downto 0) := "0";
45 variable working : boolean := false;
46
47
48 begin
49
50   WAIT ON clk, rst;
51
52   if(rst = '1') then
53
54     current_state <= S0;
55     count_word := "0";

```

```

56
57     elsif(clk'event AND clk = '1') then
58
59         case current_state is
60
61             -- Attesa start alto
62             when S0 =>
63
64                 if(start = '1' AND not(working)) then
65                     current_state <= S1;
66                     working := true;
67                 end if;
68
69
70             when S1 =>
71
72                 data_out_temp <= rom(to_integer(count_word))(15 downto 12);
73                 data_ready_temp <= '1';
74                 WAIT UNTIL data_received = '1';
75                 data_ready_temp <= '0';
76                 WAIT UNTIL data_received = '0';
77                 current_state <= S2;
78
79             when S2 =>
80
81                 data_out_temp <= rom(to_integer(count_word))(11 downto 8);
82                 data_ready_temp <= '1';
83                 WAIT UNTIL data_received = '1';
84                 data_ready_temp <= '0';
85                 WAIT UNTIL data_received = '0';
86                 current_state <= S3;
87
88             when S3 =>
89
90                 data_out_temp <= rom(to_integer(count_word))(7 downto 4);
91                 data_ready_temp <= '1';
92                 WAIT UNTIL data_received = '1';
93                 data_ready_temp <= '0';
94                 WAIT UNTIL data_received = '0';
95                 current_state <= S4;
96
97             when S4 =>
98
99                 data_out_temp <= rom(to_integer(count_word))(3 downto 0);
100                data_ready_temp <= '1';
101                WAIT UNTIL data_received = '1';
102                data_ready_temp <= '0';
103                WAIT UNTIL data_received = '0';
104

```

```

105
106      if(start = '1') then
107          WAIT UNTIL start = '0';
108      end if;
109
110      working := false;
111      count_word := count_word + 1;
112      current_state <= S0;
113
114
115  end case;
116
117 end if;
118
119
120
121 end process;
122
123
124 end Behavioral;
```

System B

Il sistema B è stato implementato con un approccio comportamentale facendo riferimento all'automa rappresentato in figura 6.6. Riceve i dati provenienti indirettamente dal sistema A tramite un bus in ingresso da 8 bit. In uscita presenta la parola da 16 bit ottenuta da A.

```

1 entity system_b is
2 port(
3     clk      : in std_logic;
4     data_in   : in std_logic_vector(7 downto 0);
5     rst       : in std_logic;
6     data_ready : in std_logic;
7     data_received : out std_logic;
8     data_out    : out std_logic_vector(15 downto 0)
9 );
10 end system_b;
11
12 architecture Behavioral of system_b is
13
14
15 signal data_out_temp : std_logic_vector(15 downto 0) := (others => '0');
16
17 type state is (S0, S1);
18 signal current_state : state := S0;
19
20
21 signal data_received_temp : std_logic;
```

```

22
23
24 begin
25
26   data_received <= data_received_temp;
27   data_out <= data_out_temp;
28
29
30   p : process
31
32 begin
33
34   WAIT ON rst, clk;
35
36   if(rst = '1') then
37
38     data_out_temp <= (others => '0');
39     data_received_temp <= '0';
40     current_state <= S0;
41
42   elsif(clk'event AND clk = '1') then
43
44     case current_state is
45
46       when S0 =>
47
48         WAIT UNTIL data_ready = '1';
49         data_out_temp(15 downto 8) <= data_in;
50         data_received_temp <= '1';
51         WAIT UNTIL data_ready = '0';
52         data_received_temp <= '0';
53         current_state <= S1;
54
55       when S1 =>
56
57         WAIT UNTIL data_ready = '1';
58         data_out_temp(7 downto 0) <= data_in;
59         data_received_temp <= '1';
60         WAIT UNTIL data_ready = '0';
61         data_received_temp <= '0';
62         current_state <= S0;
63
64     end case;
65
66   end if;
67
68 end process;
69
70

```

71 | end Behavioral;

Convertitore 4/8

Il convertitore è stato realizzato con un approccio comportamentale. È stato progettato per permettere la comunicazione tra il bus in uscita del sistema A e il bus in ingresso del sistema B che hanno un parallelismo differente. La sua implementazione fa riferimento agli automi rappresentati in figura 6.8 e 6.9.

```

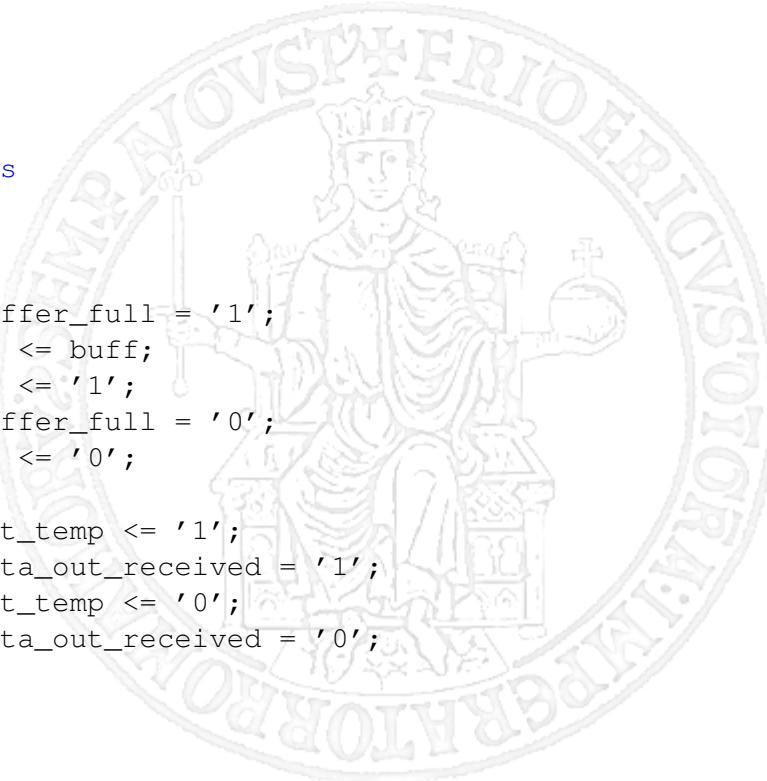
1  entity convertitore_8_16 is
2      port (
3          rst           : in  std_logic;
4          data_in       : in  std_logic_vector(3 downto 0);
5          data_ready_in : in  std_logic;
6          data_out_received: in  std_logic;
7          data_out       : out std_logic_vector(7 downto 0);
8          data_ready_out : out std_logic;
9          data_in_received : out std_logic
10     );
11 end convertitore_8_16;
12
13 architecture Behavioral of convertitore_8_16 is
14
15
16 signal buff : std_logic_vector(7 downto 0) := (others => '0');
17
18
19 signal data_out_temp      : std_logic_vector(7 downto 0) := (others => '0')
20 ;
21 signal data_ready_out_temp : std_logic := '0';
22 signal data_in_received_temp : std_logic := '0';
23 signal buffer_full, buffer_picked : std_logic := '0';
24
25 begin
26
27     data_out <= data_out_temp ;
28     data_ready_out <= data_ready_out_temp;
29     data_in_received <= data_in_received_temp;
30
31
32     receiver : process
33
34         variable data_count : unsigned(0 downto 0) := "0";
35
36     begin
37
38         WAIT UNTIL data_ready_in = '1';

```

```

39
40     if(data_count = 0) then
41
42         buff(7 downto 4) <= data_in;
43         data_in_received_temp <= '1';
44         WAIT UNTIL data_ready_in = '0';
45         data_in_received_temp <= '0';
46         data_count := data_count + 1;
47
48     elsif(data_count = 1) then
49
50         buff(3 downto 0) <= data_in;
51         buffer_full <= '1';
52         WAIT UNTIL buffer_picked = '1';
53         buffer_full <= '0';
54         WAIT UNTIL buffer_picked = '0';
55         data_in_received_temp <= '1';
56         WAIT UNTIL data_ready_in = '0';
57         data_in_received_temp <= '0';
58
59         data_count := data_count + 1;
60
61     end if;
62
63 end process;
64
65
66
67
68 sender : process
69
70 begin
71
72     WAIT UNTIL buffer_full = '1';
73     data_out_temp <= buff;
74     buffer_picked <= '1';
75     WAIT UNTIL buffer_full = '0';
76     buffer_picked <= '0';
77
78     data_ready_out_temp <= '1';
79     WAIT UNTIL data_out_received = '1';
80     data_ready_out_temp <= '0';
81     WAIT UNTIL data_out_received = '0';
82
83
84 end process;
85
86
87 end Behavioral;

```



System

Il componente *system*, realizzato con un approccio strutturale, connette tra loro i componenti sopraesposti come rappresentato in figura 6.1

```

1  entity system is
2    port (
3      clk      : in  std_logic;
4      start    : in  std_logic;
5      rst      : in  std_logic;
6      data     : out std_logic_vector(15 downto 0)
7    );
8  end system;
9
10 architecture Structural of system is
11
12   component system_a is
13
14     port (
15       clk          : in  std_logic;
16       data_received : in  std_logic;
17       rst          : in  std_logic;
18       start        : in  std_logic;
19       data_out     : out std_logic_vector(3 downto 0);
20       data_ready   : out std_logic
21     );
22
23   end component;
24
25   component system_b is
26
27     port (
28       clk          : in  std_logic;
29       data_in      : in  std_logic_vector(7 downto 0);
30       rst          : in  std_logic;
31       data_ready   : in  std_logic;
32       data_received : out std_logic;
33       data_out     : out std_logic_vector(15 downto 0)
34     );
35
36   end component;
37
38   component convertitore_8_16 is
39
40     port (
41       rst          : in  std_logic;
42       data_in      : in  std_logic_vector(3 downto 0);

```

```

43     data_ready_in    : in  std_logic;
44     data_out_received: in  std_logic;
45     data_out         : out std_logic_vector(7 downto 0);
46     data_ready_out   : out std_logic;
47     data_in_received : out std_logic
48 );
49
50 end component;
51
52
53 signal s_data_a          : std_logic_vector(3 downto 0) := (others => '0');
54 signal s_data_b          : std_logic_vector(7 downto 0) := (others => '0');
55 signal s_data_a_ready    : std_logic := '0';
56 signal s_data_b_ready    : std_logic := '0';
57 signal s_data_a_received : std_logic := '0';
58 signal s_data_b_received : std_logic := '0';
59
60
61 begin
62
63     sa : system_a
64     port map (
65         clk      => clk,
66         data_received => s_data_a_received,
67         rst      => rst,
68         start     => start,
69         data_out    => s_data_a,
70         data_ready   => s_data_a_ready
71     );
72
73
74     sb : system_b
75     port map (
76         clk      => clk,
77         data_in    => s_data_b,
78         rst      => rst,
79         data_ready   => s_data_b_ready,
80         data_received => s_data_b_received,
81         data_out    => data
82     );
83
84
85     bf : convertitore_8_16
86     port map (
87         rst      => rst,
88         data_in    => s_data_a,
89         data_ready_in  => s_data_a_ready,
90         data_out_received => s_data_b_received,
91         data_out    => s_data_b,

```

```

92     data_ready_out      => s_data_b_ready,
93     data_in_received   => s_data_a_received
94
95   );
96
97 end Structural;

```

Display Controller

Display controller è il componente che pilota i catodi e gli anodi del display a 7 segmenti installato sulla board in base alla parola da 16 bit in uscita da system.

```

1
2 entity display_controller is
3   port (
4     x      : in  std_logic_vector(15 downto 0);
5     rst    : in  std_logic;
6     clk    : in  std_logic;
7     anodo  : out std_logic_vector(7 downto 0);
8     catodo : out std_logic_vector(6 downto 0)
9   );
10 end display_controller;
11
12 architecture Behavioral of display_controller is
13
14
15 constant zero   : std_logic_vector(6 downto 0) := "0000001";
16 constant one    : std_logic_vector(6 downto 0) := "1001111";
17 constant two    : std_logic_vector(6 downto 0) := "0010010";
18 constant three  : std_logic_vector(6 downto 0) := "0000110";
19 constant four   : std_logic_vector(6 downto 0) := "1001100";
20 constant five   : std_logic_vector(6 downto 0) := "0100100";
21 constant six    : std_logic_vector(6 downto 0) := "0100000";
22 constant seven  : std_logic_vector(6 downto 0) := "0001111";
23 constant eight  : std_logic_vector(6 downto 0) := "0000000";
24 constant nine   : std_logic_vector(6 downto 0) := "0000100";
25 constant A      : std_logic_vector(6 downto 0) := "0001000";
26 constant b      : std_logic_vector(6 downto 0) := "1100000";
27 constant C      : std_logic_vector(6 downto 0) := "0110001";
28 constant d      : std_logic_vector(6 downto 0) := "1000010";
29 constant E      : std_logic_vector(6 downto 0) := "0110000";
30 constant F      : std_logic_vector(6 downto 0) := "0111000";
31 constant H      : std_logic_vector(6 downto 0) := "1001000";
32
33 signal count    : unsigned(2 downto 0)      := (others => '0');
34 signal anodo_temp : std_logic_vector(7 downto 0) := (others => '1');
35 signal catodo_temp : std_logic_vector(6 downto 0) := (others => '1');
36

```

```

37
38
39 begin
40
41     anodo <= anodo_temp;
42     catodo <= catodo_temp;
43
44     p : process(clk)
45     begin
46
47         if(clk'event AND clk = '1') then
48             if(rst = '1') then
49                 catodo_temp <= zero;
50
51             -- PRIMA CIFRA SECONDI
52             elsif(count = 0) then
53                 count <= count + 1;
54                 anodo_temp <= "11111110";
55
56             case x(3 downto 0) is
57
58                 when x"0" => catodo_temp <= zero;
59                 when x"1" => catodo_temp <= one;
60                 when x"2" => catodo_temp <= two;
61                 when x"3" => catodo_temp <= three;
62                 when x"4" => catodo_temp <= four;
63                 when x"5" => catodo_temp <= five;
64                 when x"6" => catodo_temp <= six;
65                 when x"7" => catodo_temp <= seven;
66                 when x"8" => catodo_temp <= eight;
67                 when x"9" => catodo_temp <= nine;
68                 when x"a" => catodo_temp <= A;
69                 when x"b" => catodo_temp <= b;
70                 when x"c" => catodo_temp <= C;
71                 when x"d" => catodo_temp <= d;
72                 when x"e" => catodo_temp <= E;
73                 when x"f" => catodo_temp <= F;
74                 when others => catodo_temp <= H;
75
76             end case;
77
78             -- SECONDA CIFRA SECONDI
79             elsif (count = 1) then
80                 count <= count + 1;
81                 anodo_temp <= "11111101";
82
83             case x(7 downto 4) is
84
85                 when x"0" => catodo_temp <= zero;

```

```

86      when x#"1" => catodo_temp <= one;
87      when x#"2" => catodo_temp <= two;
88      when x#"3" => catodo_temp <= three;
89      when x#"4" => catodo_temp <= four;
90      when x#"5" => catodo_temp <= five;
91      when x#"6" => catodo_temp <= six;
92      when x#"7" => catodo_temp <= seven;
93      when x#"8" => catodo_temp <= eight;
94      when x#"9" => catodo_temp <= nine;
95      when x#"a" => catodo_temp <= A;
96      when x#"b" => catodo_temp <= b;
97      when x#"c" => catodo_temp <= C;
98      when x#"d" => catodo_temp <= d;
99      when x#"e" => catodo_temp <= E;
100     when x#"f" => catodo_temp <= F;
101     when others => catodo_temp <= H;

102
103
104    end case;
105
106  -- PRIMA CIFRA MINUTI
107  elsif(count = 2) then
108    count <= count + 1;
109    anodo_temp <= "11111011";
110
111  case x(11 downto 8) is
112
113    when x#"0" => catodo_temp <= zero;
114    when x#"1" => catodo_temp <= one;
115    when x#"2" => catodo_temp <= two;
116    when x#"3" => catodo_temp <= three;
117    when x#"4" => catodo_temp <= four;
118    when x#"5" => catodo_temp <= five;
119    when x#"6" => catodo_temp <= six;
120    when x#"7" => catodo_temp <= seven;
121    when x#"8" => catodo_temp <= eight;
122    when x#"9" => catodo_temp <= nine;
123    when x#"a" => catodo_temp <= A;
124    when x#"b" => catodo_temp <= b;
125    when x#"c" => catodo_temp <= C;
126    when x#"d" => catodo_temp <= d;
127    when x#"e" => catodo_temp <= E;
128    when x#"f" => catodo_temp <= F;
129    when others => catodo_temp <= H;

130
131  end case;
132
133
134

```

```

135      -- SECONDA CIFRA MINUTI
136      elsif (count = 3) then
137          count <= (others => '0');
138          anodo_temp <= "11110111";
139
140          case x(15 downto 12) is
141
142              when x"0" => catodo_temp <= zero;
143              when x"1" => catodo_temp <= one;
144              when x"2" => catodo_temp <= two;
145              when x"3" => catodo_temp <= three;
146              when x"4" => catodo_temp <= four;
147              when x"5" => catodo_temp <= five;
148              when x"6" => catodo_temp <= six;
149              when x"7" => catodo_temp <= seven;
150              when x"8" => catodo_temp <= eight;
151              when x"9" => catodo_temp <= nine;
152              when x"a" => catodo_temp <= A;
153              when x"b" => catodo_temp <= b;
154              when x"c" => catodo_temp <= C;
155              when x"d" => catodo_temp <= d;
156              when x"e" => catodo_temp <= E;
157              when x"f" => catodo_temp <= F;
158              when others => catodo_temp <= H;
159
160      end case;
161
162
163      end if;
164
165  end if;
166
167 end process;
168
169 end Behavioral;

```

Display Buffer

In *Display Buffer* sono state collegate tutte le componenti come in figura 6.11 e in uscita vi sono i segnali uscenti da *Display Controller* che consentono di mostrare sul display l'ultimo dato trasferito dal sistema A al sistema B.

```

1 entity display_buffer is
2
3 port (
4     rst : in std_logic;
5     clk : in std_logic;
6     start : in std_logic;

```

```

7      an     : out std_logic_vector(7 downto 0);
8      cat   : out std_logic_vector(6 downto 0)
9  );
10
11 end display_buffer;
12
13
14 architecture Structural of display_buffer is
15
16 component base_tempi is
17   port(
18     reset    : in  std_logic;
19     clk_in   : in  std_logic;
20     clk_out  : out std_logic_vector(0 to 3)
21   );
22 end component;
23
24
25 component system is
26   port(
27     clk      : in  std_logic_vector(0 to 2);
28     start    : in  std_logic;
29     rst      : in  std_logic;
30     data     : out std_logic_vector(15 downto 0)
31   );
32 end component;
33
34 component display_controller is
35   port (
36     x       : in  std_logic_vector(15 downto 0);
37     rst    : in  std_logic;
38     clk    : in  std_logic;
39     anodo  : out std_logic_vector(7 downto 0);
40     catodo : out std_logic_vector(6 downto 0)
41   );
42 end component;
43
44
45 signal s_data : std_logic_vector(15 downto 0);
46 signal s_clk  : std_logic_vector(0 to 3) := (others => '0');
47
48
49 begin
50
51   bt : base_tempi
52
53     port map (
54

```

```
56      reset    => rst,
57      clk_in   => clk,
58      clk_out  => s_clk
59
60  );
61
62
63  sy : system
64
65  port map (
66
67      clk    => s_clk(0 to 2),
68      start  => start,
69      rst    => rst,
70      data   => s_data
71
72  );
73
74
75  dc : display_controller
76
77  port map (
78
79      x      => s_data,
80      rst   => rst,
81      clk   => s_clk(3),
82      anodo => an,
83      catodo=> cat
84
85  );
86
87
88 end Structural;
```



6.3 Simulazione

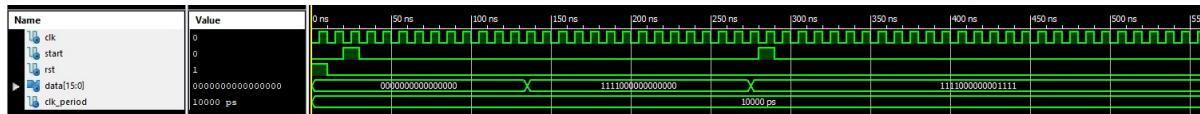


Figura 6.10: Simulazione

Per effettuare la simulazione di entrambi gli approcci implementati, si è realizzato il testbench qui di seguito riportato. In questo caso, poiché era semplice valutare il corretto comportamento della macchina, non sono stati utilizzati gli assert.

Per verificare il corretto funzionamento, sono state caricate due word da 16 bit nel sistema A e, tramite il segnale di start, viene dato il via al loro trasferimento verso il sistema B. In simulazione si può notare che il valore di uscita corrisponde all'ultima word trasferita.

```

1 begin
2
3     rst <= '1';
4     wait for 10 ns;
5
6     rst <= '0';
7     wait for 10 ns;
8
9     start <= '1';
10    wait for 10 ns;
11
12    start <= '0';
13    wait for 250 ns;
14
15    start <= '1';
16    wait for 10 ns;
17
18    start <= '0';
19    wait for 50 ns;
20
21
22    wait;
23 end process;
24
25 END;
```

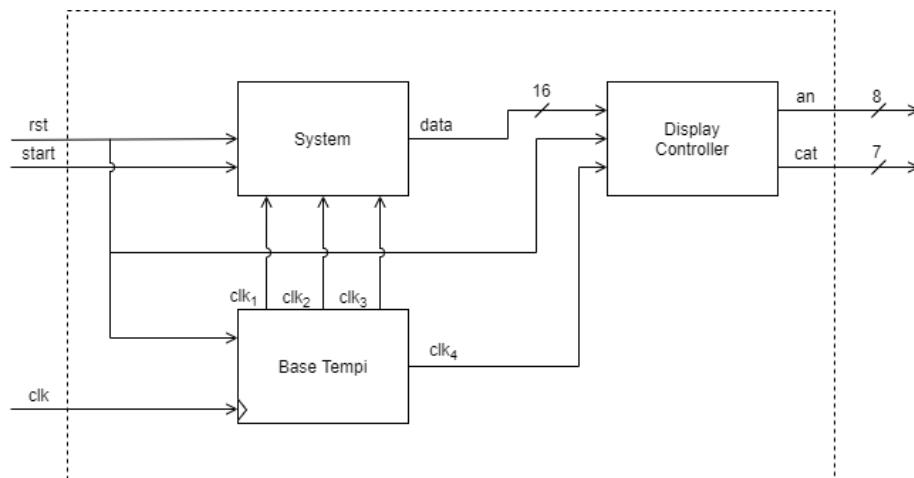


Figura 6.11: Architettura implementata sull'FPGA della board

6.4 Sintesi

Per caricare il sistema sulla scheda sono stati fatti dei cambiamenti sostanziali al codice. La principale modifica fatta al codice dei tre componenti fondamentali del sistema (system A, B e Convertitore_4_8) è stata quella di eliminare tutti i costrutti *WAIT* utilizzati, poiché simulabili ma non sintetizzabili. I costrutti *WAIT* sono quindi stati sostituiti da degli stati che ne simulano il funzionamento.

Per testare il corretto funzionamento della macchina, è stato deciso di visualizzare l'ultima word trasferita al sistema B tramite il display a 7 segmenti. La codifica scelta per la visualizzazione dei dati è quella esadecimale, in modo da rappresentare i 16 bit su 4 cifre. Per effettuare ciò è stato inserito il divisore di frequenza fornito nel materiale didattico modificando la frequenza di clock da 100Hz a 1000 Hz, inoltre è stato deciso di attivare il reset quando il segnale è alto. Il divisore di frequenza è stato inoltre utilizzato per fornire a *system A*, *system B* e al *convertitore 4_8* clock a frequenze diverse. In figura 6.11 è possibile osservare l'architettura progettata per implementare quanto appena esposto.



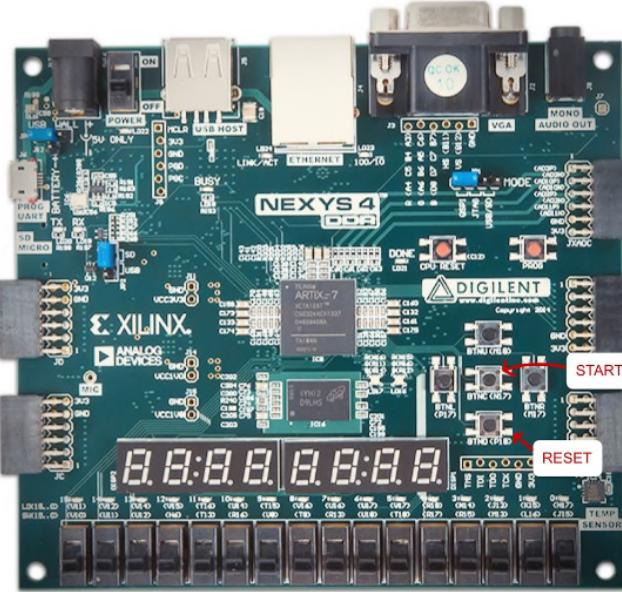


Figura 6.12: Mapping sulla scheda

6.4.1 Mapping su scheda

Di seguito è riportato il file *UCF* che mappa i segnali di *Display Buffer* sui dispositivi di I/O della scheda. Si è scelto di utilizzare il bottone centrale per dare lo *START*, mentre il bottone di sotto per resettare la macchina. Infine, per la visualizzazione dei dati si è scelto di utilizzare la codifica esadecimale, in modo tale da rappresentare i 16 bit sulle ultime 4 cifre del display.

```

1  NET "clk"      LOC = "E3" | IOSTANDARD = "LVCMOS33"; #Bank = 35, Pin name = #
     IO_L12P_T1_MRCC_35,
2
3  ## 7 segment display
4  NET "cat<6>"          LOC=T10 | IOSTANDARD=LVCMOS33; #
     IO_L24N_T3_A00_D16_14
5  NET "cat<5>"          LOC=R10 | IOSTANDARD=LVCMOS33; #IO_25_14
6  NET "cat<4>"          LOC=K16 | IOSTANDARD=LVCMOS33; #IO_25_15
7  NET "cat<3>"          LOC=K13 | IOSTANDARD=LVCMOS33; #IO_L17P_T2_A26_15
8  NET "cat<2>"          LOC=P15 | IOSTANDARD=LVCMOS33; #IO_L13P_T2_MRCC_14
9  NET "cat<1>"          LOC=T11 | IOSTANDARD=LVCMOS33; #
     IO_L19P_T3_A10_D26_14
10 NET "cat<0>"          LOC=L18 | IOSTANDARD=LVCMOS33; #IO_L4P_T0_D04_14
11 #NET "cat<7>"          LOC=H15 | IOSTANDARD=LVCMOS33; #
     IO_L19N_T3_A21_VREF_15
12
13 NET "an<0>"          LOC=J17 | IOSTANDARD=LVCMOS33; #IO_L23P_T3_FOE_B_15
14 NET "an<1>"          LOC=J18 | IOSTANDARD=LVCMOS33; #IO_L23N_T3_FWE_B_15
15 NET "an<2>"          LOC=T9 | IOSTANDARD=LVCMOS33; #IO_L24P_T3_A01_D17_14
16 NET "an<3>"          LOC=J14 | IOSTANDARD=LVCMOS33; #IO_L19P_T3_A22_15
17 NET "an<4>"          LOC=P14 | IOSTANDARD=LVCMOS33; #IO_L8N_T1_D12_14
18 NET "an<5>"          LOC=T14 | IOSTANDARD=LVCMOS33; #IO_L14P_T2_SRCC_14

```

```

19 NET "an<6>"          LOC=K2 | IOSTANDARD=LVC MOS33; #IO_L23P_T3_35
20 NET "an<7>"          LOC=U13 | IOSTANDARD=LVC MOS33; #IO_L23N_T3_A02_D18_14
21
22 ## Buttons
23 #NET "cpu_resetn"     LOC=C12 | IOSTANDARD=LVC MOS33; #IO_L3P_T0_DQS_AD1P_15
24
25 NET "start"           LOC=N17 | IOSTANDARD=LVC MOS33; #IO_L9P_T1_DQS_14
26 NET "rst"              LOC=P18 | IOSTANDARD=LVC MOS33; #IO_L9N_T1_DQS_D13_14
27 #NET "btnl"            LOC=P17 | IOSTANDARD=LVC MOS33; #IO_L12P_T1_MRCC_14
28 #NET "btnr"            LOC=M17 | IOSTANDARD=LVC MOS33; #IO_L10N_T1_D15_14
29 #NET "btnu"            LOC=M18 | IOSTANDARD=LVC MOS33; #IO_L4N_T0_D05_14

```

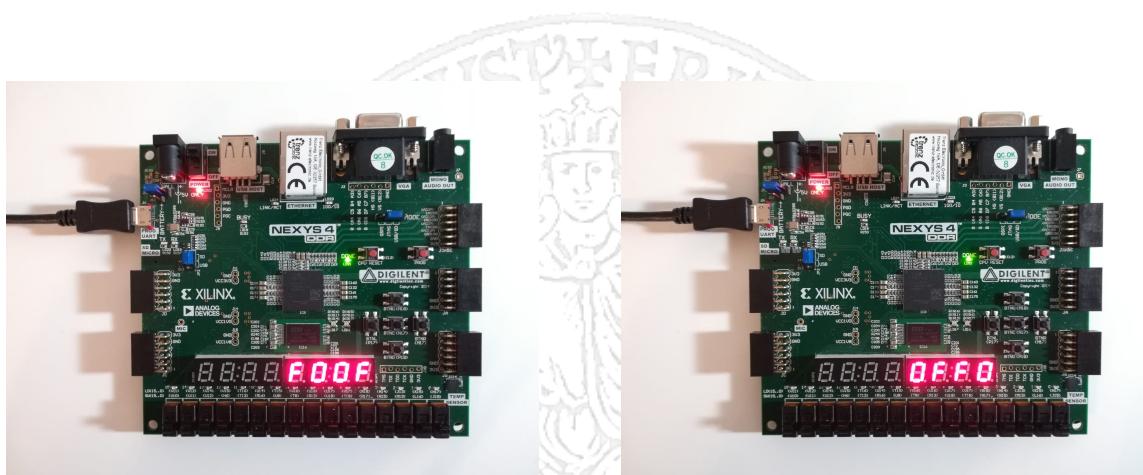
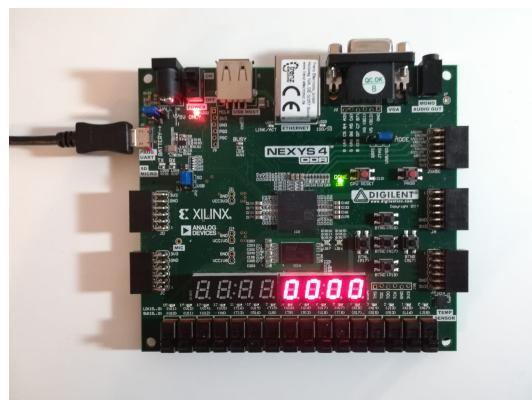


Figura 6.13: Implementazione del progetto sulla board. In alto la macchina in stato di reset. In basso le due word trasferite dal sistema A al sistema B.

Capitolo 7

Prodotto Scalare

7.1 Traccia

Progettare un sistema in grado di calcolare il prodotto scalare fra 2 vettori A e B di M elementi, ciascuno codificato su N bit (M ed N a scelta dello studente).

$$\sum_{i=0}^{M-1} A_i \cdot B_i$$

Il sistema deve essere alimentato con k copie di vettori A e B diversi (cioè $[A_0B_0], [A_1B_1], \dots, [A_kB_k]$), forniti in uno dei modi seguenti (a scelta dello studente):

1. Tutti i vettori A_j e B_j ($j=1, \dots, k$) sono precaricati in una ROM, e ciascuna coppia è fornita alla macchina in parallelo;
2. Tutti i vettori sono precaricati, e la macchina riceve serialmente gli elementi di ciascuna coppia di vettori tramite l'ausilio di registri a scorrimento (es., nel caso di $M=3$, vengono forniti in sequenza $[A_0(0) B_0(0)]$, poi $[A_0(1)B_0(1)]$, e poi $[A_0(2)B_0(2)]$; successivamente, vengono forniti $[A_1(0)B_1(0)]$, $[A_1(1)B_1(1)]$ e $[A_1(2)B_1(2)]$, e così via);
3. Ciascuna coppia di vettori viene ricevuta da un'entità produtore mediante handshaking (e gestita in modalità parallela o seriale a seconda dell'architettura scelta).

Lo studente, inoltre, può scegliere di realizzare un datapath pipelined o meno, e di utilizzare la logica cablata o microprogrammata per l'unità di controllo.

7.2 Soluzione

Il prodotto scalare è un'operazione fondamentale nel campo dell'elaborazione dei segnali. Un sistema in grado di implementare tale operazione può essere utilizzato per calcolare, ad esempio, la trasformata di Fourier discreta (DFT) di un segnale o la trasformata coseno discreta (DCT).

L'architettura mostrata in questo elaborato è semplicemente una possibile implementazione hardware di tale operazione e può essere ottimizzata a seconda del contesto.

Nel descrivere l'architettura realizzata per il sistema prodotto scalare, sono riportate le scelte progettuali adottate che hanno influenzato l'intero sviluppo, ponendo l'accento sui pro e sui contro

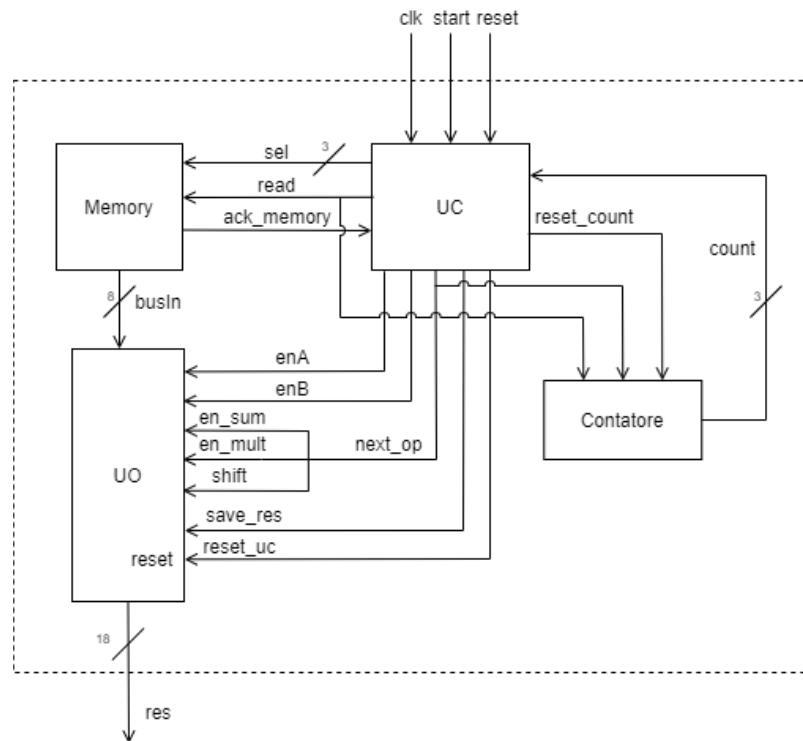


Figura 7.1: Architettura generale del sistema

delle scelte prese.

In figura 7.1 è possibile osservare l'intera architettura del sistema. I componenti da cui è costituito sono: un'unità di controllo, un'unità operativa, una memoria e un contatore.

Si è scelto di suddividere il problema in due unità fondamentali, ovvero nell'UC e nell'UO, così da separare quella che si occupa di effettuare il calcolo da quella che gestisce i segnali di controllo. Inoltre, è stata aggiunta una memoria al cui interno vi sono salvati i vettori sui cui il sistema effettua l'elaborazione. Ogni vettore caricato è formato da 3 numeri naturali codificati su 8 bit. Tali bit sono trasferiti all'unità operativa su un bus con parallelismo da 8. Il trasferimento viene gestito dall'unità di controllo tramite due protocolli: uno handshaking asincrono con la memoria e l'altro sincrono con l'unità operativa (figura 7.2). Entrambi consentono la corretta acquisizione dei dati da parte dell'unità operativa.

Infine, vi è un contatore che supporta l'unità di controllo nelle sue operazioni. Tutti i componenti sono descritti in dettaglio nei paragrafi che seguono.

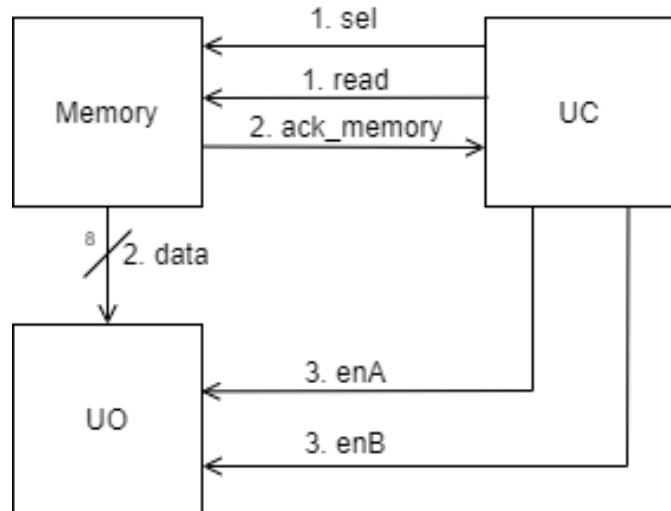


Figura 7.2: Segnali scambiati tra le entità per il protocollo di comunicazione

7.2.1 Unità Operativa

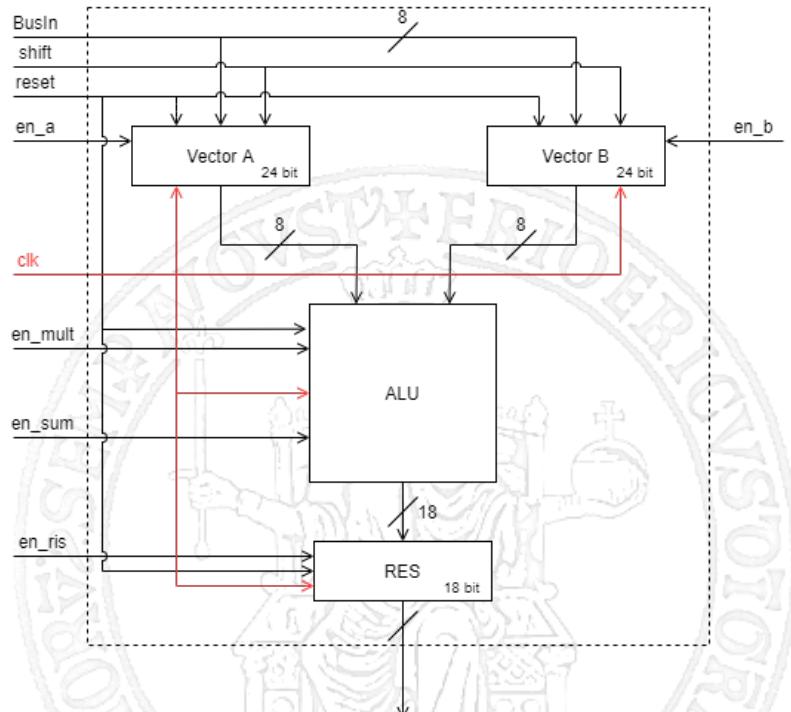


Figura 7.3: Architettura dell'unità operativa

L'unità operativa è composta da 2 registri tampone, contenenti i vettori di cui si desidera calcolare il prodotto scalare, da un'unità logico aritmetica che effettua il calcolo vero e proprio, e dal registro *res* che contiene il risultato finale dell'operazione. Tutti i segnali di ingresso, fatta eccezione per *BusIn*, sono segnali di controllo generati dall'UC per la gestione dei registri.

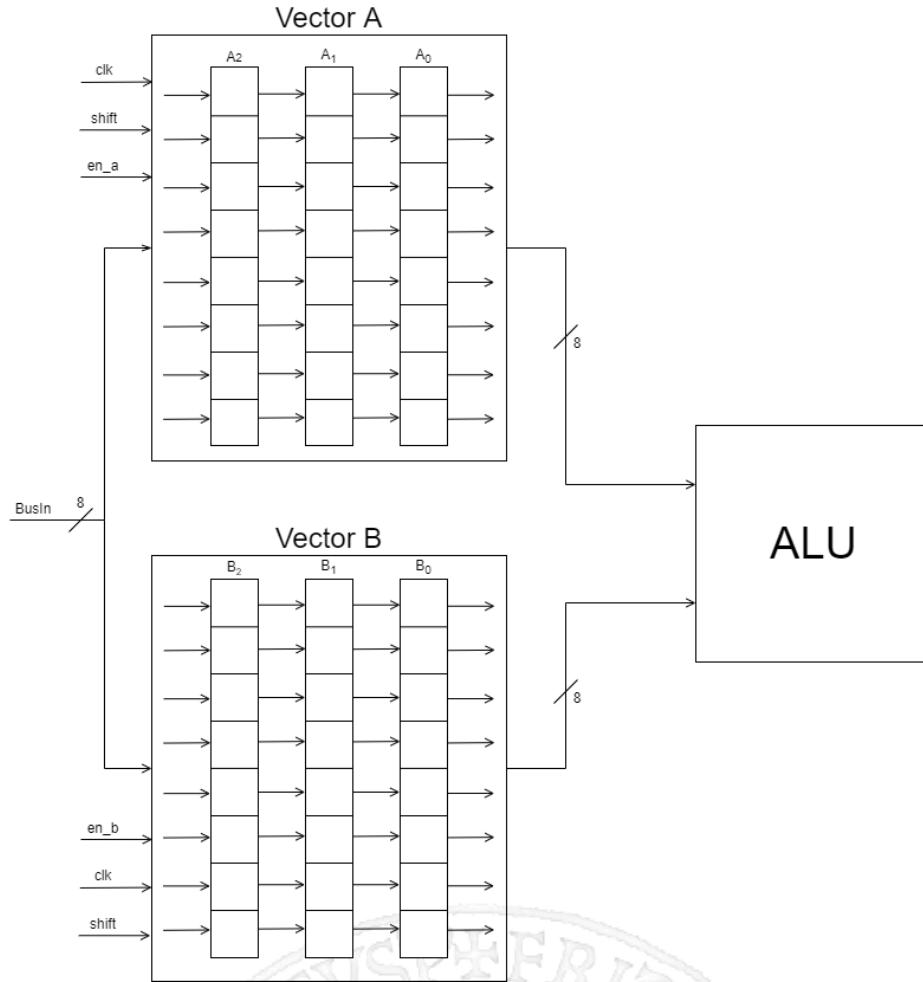


Figura 7.4: Architettura interna del componente vector

7.2.1.1 Vector

Al suo interno Vector presenta 8 registri a scorrimento da 3 bit che permettono di acquisire 24 bit in tre colpi di clock sul fronte di discesa. Sono impiegati per memorizzare i vettori dai quali sarà calcolato il prodotto scalare.

Vector presenta in ingresso 4 segnali:

- *busIn* da 8 bit per la ricezione dei dati;
- *En_a/En_b*, segnali di abilitazione per l'acquisizione dei dati nei registri A_2/B_2 e lo scorrimento dei registri;
- il *clock*;
- *shift* che ha il compito di far scorrere gli elementi da 8 bit nei registri interni di Vector.

Il segnale d'uscita comprende gli 8 bit uscenti dai registri a scorrimento, nel momento in cui è alto il segnale di *shift*, e vengono caricati in ingresso alla ALU che si occuperà di effettuare l'elaborazione.

Gli elementi vengono caricati con un approccio seriale. Tale scelta permette sia di non avere un

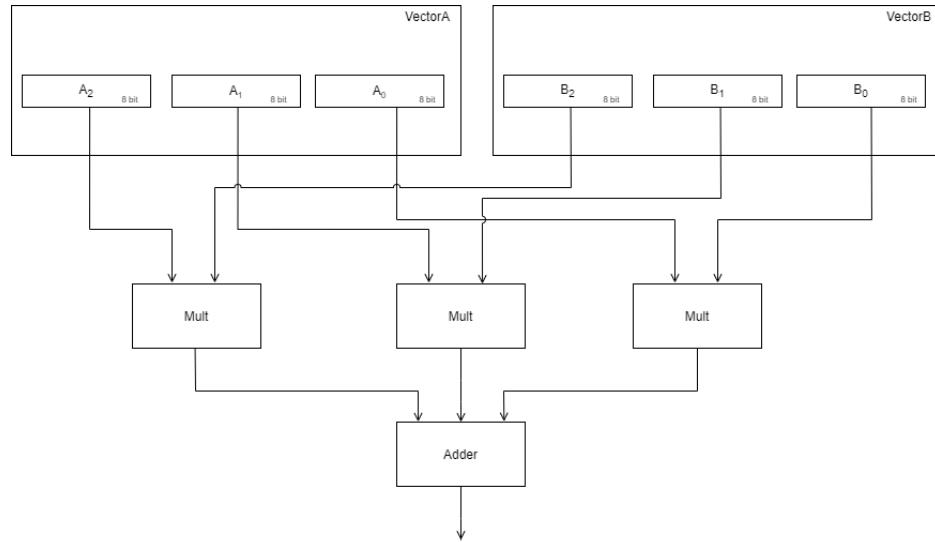


Figura 7.5: Approccio parallelo per lo sviluppo dell'ALU

bus interno con un parallelismo direttamente proporzionale alla grandezza del vettore, sia di non dover utilizzare componenti aggiuntivi, che sarebbero stati necessari in un approccio parallelo. Inoltre questa scelta permette di ottenere un'architettura più scalabile, potendo utilizzare la stessa componente anche per vettori di lunghezza maggiore modificando l'unità di controllo.

7.2.1.2 ALU

Per lo sviluppo dell'ALU è possibile seguire sia un approccio parallelo, come mostrato in figura 7.5, che uno seriale. Una struttura parallela ha il vantaggio di elaborare i dati più velocemente rispetto a quella seriale, ma presenta lo svantaggio di occupare un'area maggiore. Essa è costituita da 3 unità che effettuano la moltiplicazione, una per ogni elemento del vettore, ed un addizionatore che somma i risultati dei moltiplicatori. È chiaro che all'aumentare degli elementi del vettore aumenta sia il numero di moltiplicatori sia la complessità dell'addizionatore. Sicché si è preferito adottare un approccio seriale versione pipelined per incrementare la produttività del sistema, come mostrato in figura 7.6.

In riferimento a questa figura è possibile osservare la struttura interna della ALU. *MULT* e *ADDER* sono le componenti che effettuano rispettivamente l'operazione di moltiplicazione e addizione; in particolare, la prima coinvolge numeri interi positivi codificati su 8 bit, mentre la seconda tra interi codificati su 18 bit per evitare overflow nel caso critico ($255 \cdot 255 \cdot 3$).

REG e *ACC* sono i registri che contengono i risultati delle operazioni precedenti; inoltre, nel registro *REG* sono aggiunti due zeri nei due bit più significativi per inviare all'adder un vettore da 18 bit.

REG è un registro tampone che ha il compito di disaccoppiare l'operazione di moltiplicazione con quella di addizione; infatti, con una corretta tempificazione è possibile sommare il risultato della moltiplicazione con quello precedente mentre *MULT* è occupato ad effettuare un nuovo calcolo. *ACC* invece è un registro di appoggio che consente di salvare il risultato dell'addizione precedente e di poterlo fornire sia in retroazione che in uscita.

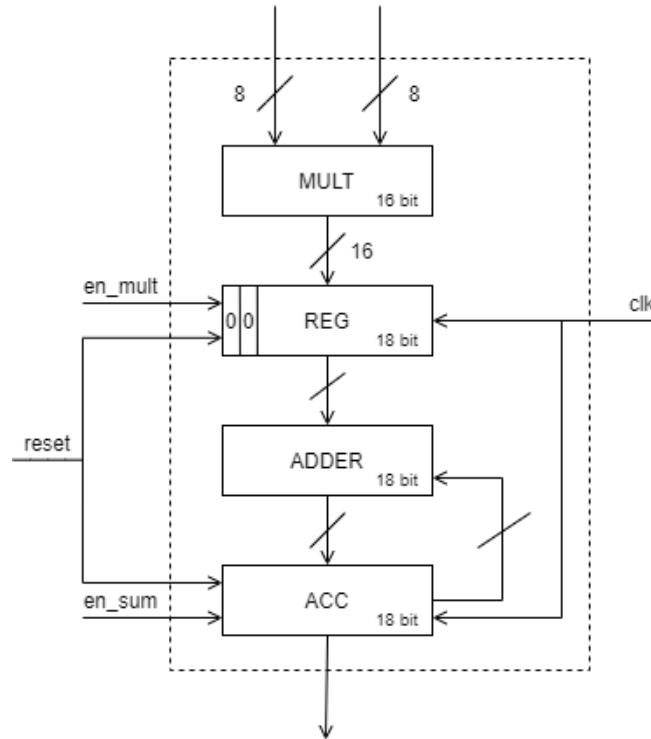


Figura 7.6: Approccio seriale pipelined per lo sviluppo dell'ALU

I registri salvano i valori in ingresso sul fronte di discesa del clock quando i rispettivi segnali di *enable* sono alti.

7.2.2 Unità di Controllo

In figura 7.7 è rappresentata l'interfaccia dell'unità di controllo. I segnali di ingresso sono:

- *ack_mem*: segnale di ACK ricevuto dalla memoria durante il protocollo asincrono.
- *reset*: segnale per riportare la macchina in uno stato noto.
- *start*: segnale per avviare il calcolo del prodotto scalare.
- *count*: uscita del contatore.
- *clk*.

I segnali in uscita sono:

- *sel*: segnale in ingresso alla memoria per selezionare la locazione da leggere.
- *read*: segnale per iniziare la lettura dalla memoria della locazione selezionata.
- *enA*: segnale per salvare nel vettore A il valore di *busIn*.
- *enB*: segnale per salvare nel vettore B il valore di *busIn*.

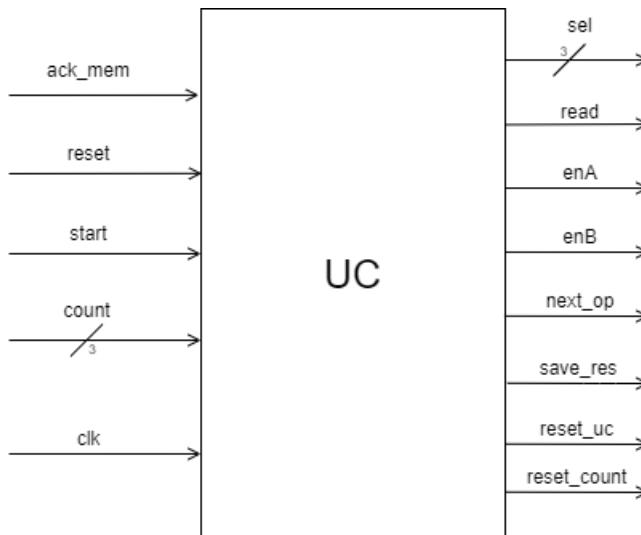


Figura 7.7: Interfaccia dell'unità di controllo

- *next_op*: segnale utilizzato per shiftare i registri all'interno di vector e abilitare in lettura i registri dell'ALU.
- *save_res*: segnale per caricare in un registro il risultato dell'ALU.
- *reset_uc*: segnale di reset generato dall'unità di controllo per svuotare i registri dell'unità operativa.
- *reset_count*: segnale per resettare il contatore.

L'unità di controllo è stata realizzata con una logica microprogrammata. Per la realizzazione dell'unità di controllo è stato progettato un diagramma di flusso per mostrare l'algoritmo per effettuare il calcolo. Tale algoritmo è stato tradotto in una sequenza di microistruzioni.

7.2.2.1 Automi dell'unità di controllo

L'automa in figura 7.8 descrive il funzionamento generale dell'Unità di Controllo. È possibile scendere ad un livello di dettaglio più basso rispetto a quello rappresentato poiché i nodi di **inA** e **inB** possono essere ulteriormente espansi.

L'automa è caratterizzato dai seguenti stati di evoluzione:

Start L'unità di controllo attende il segnale di *start* che dà il via all'operazione di prodotto scalare.

Reset In questo stato sono generati i segnali di reset per riportare l'unità di controllo e il contatore in uno stato noto. L'utilità del contatore è discussa in seguito.

inA Vengono generati i segnali che consentono di salvare in *VectorA* il vettore letto dalla memoria. La macchina evolve nello stato successivo quando il contatore raggiunge il valore 3, ovvero quando sono stati caricati tutti e 3 gli elementi del vettore.

inB Vengono generati i segnali che consentono di salvare in *VectorB* il vettore letto dalla memoria. Il passaggio di stato è analogo a quello di **inA**.

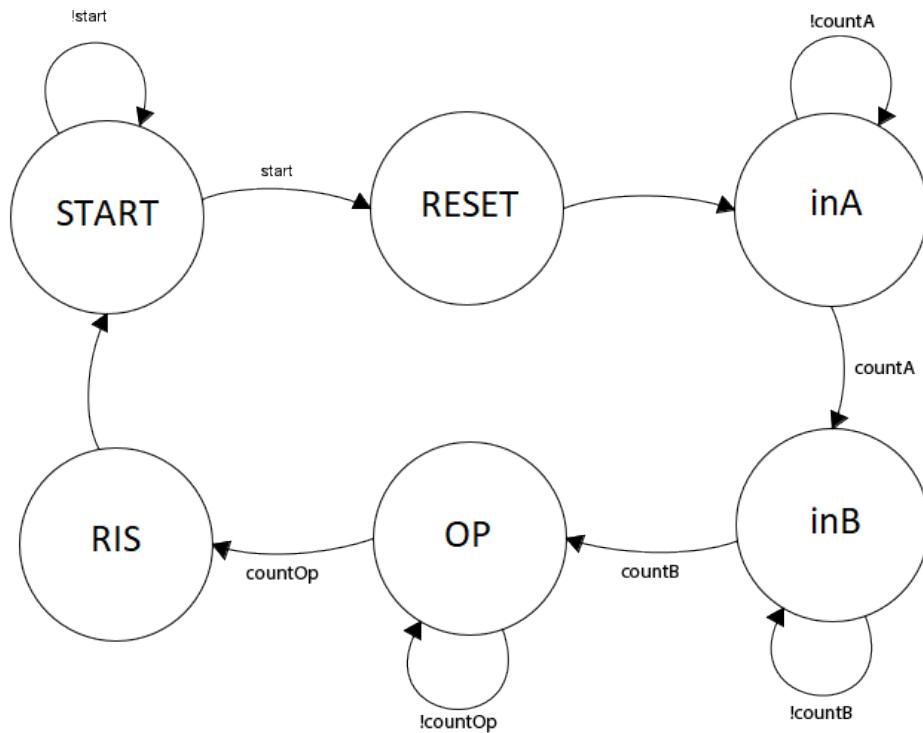


Figura 7.8: Automa che descrive il comportamento dell'unità di controllo

OP In questo stato è resettato il contatore e vengono generati i segnali per effettuare l'operazione di prodotto scalare. La macchina permane in questo stato fino a quando il contatore non raggiungere il valore 4 che sancisce il completamento dell'operazione da parte dell'Unità Operativa. È stato pertanto posto un vincolo: in 4 colpi di clock l'Unità Operativa deve restituire il risultato del prodotto scalare.

Ris In quest'ultimo stato, viene posto in un registro esterno il risultato dell'ALU.

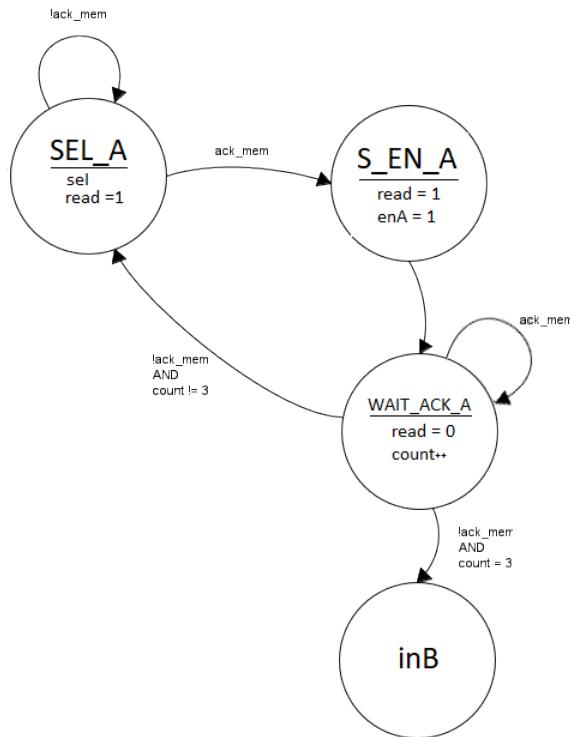
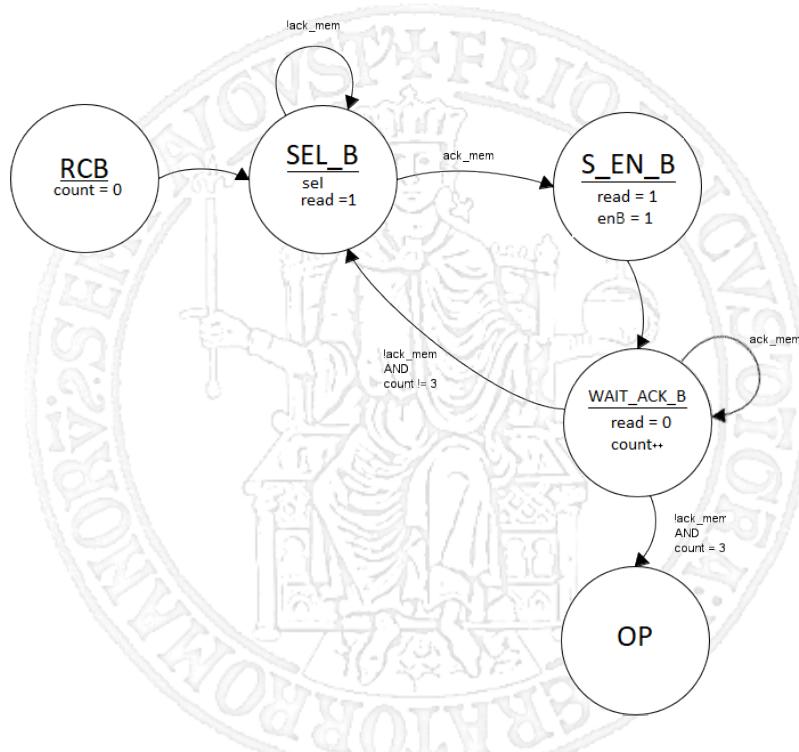
In figura 7.9 è rappresentato **inA**, espanso nei seguenti stati:

Sel A La macchina seleziona il byte che desidera leggere dalla memoria e avvia l'operazione di lettura attraverso il segnale di *read*. Il sistema rimane in questo stato fino a quando la memoria non notifica con un segnale di ACK il completamento dell'operazione.

S_EN_A Viene inviato a VectorA il segnale di abilitazione per la lettura degli 8 bit in uscita dalla memoria.

Wait ack A Caricati i dati nel registro tampone dell'ALU, viene abbassato il segnale di *read* e incrementato il contatore. Se la sequenza è stata eseguita 3 volte, la macchina evolve nello stato **inB**, altrimenti ritorna in **Sel A**

In figura 7.10 è rappresentato **inB** che risulta essere analogo ad **inA** aggiungendo uno stato iniziale RCB che serve a resettare il contatore.


 Figura 7.9: Analisi in dettaglio dello stato **inA**

 Figura 7.10: Analisi in dettaglio dello stato **inB**

NS 4 bit	JCount	JAck	base 3 bit	read	enA	enB	next op	save res	reset uc	reset count
-------------	--------	------	---------------	------	-----	-----	------------	-------------	-------------	----------------

Figura 7.11: Formato della microistruzione

	NS	Jcount	Jack	Base	Read	En_A	En_B	Next_Op	Save_Res	Reset_UC	Reset_Count
Idle	0000	0	0	000	0	0	0	0	0	0	0
Reset	0010	0	0	000	0	0	0	0	0	1	1
SelA	0011	0	1	000	0	0	0	0	0	0	0
S_enA	0100	0	0	000	1	1	0	0	0	0	0
Wait_Ack_A	0101	1	1	000	0	0	0	0	0	0	0
Rc_B	0110	0	0	000	0	0	0	0	0	0	1
Sel_B	0111	0	1	011	1	0	0	0	0	0	0
S_en_B	1000	0	0	011	1	0	1	0	0	0	0
Wait_Ack_B	1001	1	1	011	0	0	0	0	0	0	0
Rc_Op	1010	0	0	000	0	0	0	0	0	0	1
Op	1011	1	0	000	0	0	0	1	0	0	0
Res	0000	0	0	000	0	0	0	0	1	0	0

Figura 7.12: Rappresentazione della Control Store

7.2.2.2 Microistruzioni

Poiché si è deciso di progettare l'unità di controllo in logica microprogrammata, la sequenza dei segnali di abilitazione è stata memorizzata in una *microRom*. Come anticipato dall'interfaccia in figura 7.7 è necessario che l'unità di controllo generi i seguenti segnali:

- sel
- read
- en_A
- en_b
- next_op
- save_res
- reset_uc
- reset_count

Inoltre, sono necessari ulteriori due segnali di flusso *Jcount* e *Jack* per, rispettivamente, gestire la lettura del valore di count e del segnale di ack proveniente dalla memoria. Infine, è stato inserito nella control word il segnale *NS* che rappresenta il prossimo indirizzo della microistruzione da eseguire. In figura 7.11 è presente la struttura della *control word*. In totale sono necessari 16 bit per generare tutti i segnali necessari per la gestione del sistema. Da notare che al posto di *sel* vi è il segnale *base*. Poiché nella memoria gli elementi di un vettore sono memorizzati in sequenza, nella *microRom* è conservato solamente l'indirizzo del primo elemento; *sel* è la somma tra *base* e il valore di conteggio.

Nella tabella in figura 7.12 è riportata la *Control Store*.

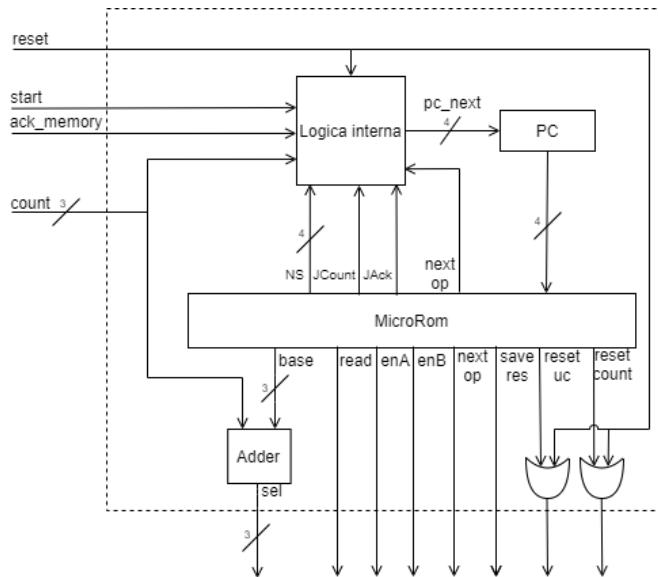


Figura 7.13: Architettura interna dell'unità di controllo

7.2.2.3 Struttura interna

L'unità di controllo è costituita da:

- un PC che contiene l'indirizzo della microistruzione da eseguire;
- la MicroRom illustrata nel paragrafo precedente;
- una logica interna per il calcolo della prossima istruzione;
- un Adder che ha il compito di sommare la base con il valore di conteggio.

La logica interna opera nel seguente modo:

- $JCount = 0$ e $JAck = 0$: il valore di Pc_next è NS.
- $JCount = 0$ e $JAck = 1$: se non è pervenuto il segnale di ack dalla memoria, viene rieseguita la stessa istruzione, altrimenti si passa all'istruzione successiva.
- $JCount = 1$ e $JAck = 0$: si passa all'istruzione successiva se il valore del contatore è 4, altrimenti viene eseguita l'istruzione corrente.
- $JCount = 1$ e $JAck = 1$: le microistruzioni che presentano entrambi questi valori alti devono attendere che il segnale di ACK si abbassi; dopodiché, se il valore di count è 3, il valore di PC è NS; altrimenti, viene eseguita la microistruzione SelA o SelB a seconda dello stato corrente.

Per semplificare la logica interna e per rendere l'unità di controllo più versatile, potevano essere aggiunti ulteriori bit alla control word per comunicare al contatore a quale valore di conteggio arrestarsi. In questo modo, l'Unità di Controllo non necessita di ricevere in ingresso il conteggio ma solamente un segnale che l'avvisi che il conteggio ha raggiunto la soglia indicata. Tale approccio avrebbe quindi semplificato l'Unità di Controllo a discapito di un contatore leggermente più complesso.

7.3 Codice

Nella seguente implementazione sono stati utilizzati i componenti necessari di una logica micro-programmata.

Sono stati utilizzati i costrutti Wait Until per implementare l'handshaking asincrono tra unità di controllo e memoria.

Memoria

Nella memoria sono memorizzati 2 vettori uguali da 3 elementi codificati su 8 bit: [1, 2, 3]. Pertanto, il prodotto scalare è la norma al quadrato del vettore: $\|[1, 2, 3]\|^2 = 14$. Attraverso il costrutto *wait* è implementato il protocollo di comunicazione con l'unità di controllo.

```

1  entity Memoria is
2      port(
3          sel : in std_logic_vector(2 downto 0);
4          read1 : in std_logic;
5          data : out std_logic_vector(7 downto 0);
6          data_ready : out std_logic
7      );
8  end Memoria;
9
10 architecture Behavioral of Memoria is
11 SUBTYPE data_type is integer range 0 to 255;
12 TYPE memory_type is array(0 to 5) of data_type;
13
14 constant memory : memory_type := (
15     0 => 1,
16     1 => 2,
17     2 => 3,
18     3 => 1,
19     4 => 2,
20     5 => 3
21 );
22
23 begin
24     mem : process
25         variable index : integer;
26         begin
27             WAIT UNTIL read1 = '1';
28             index := to_integer(unsigned(sel));
29             data <= std_logic_vector(to_unsigned(memory(index), data'length));
30             data_ready <= '1';
31             WAIT UNTIL read1 = '0';
32             data_ready <= '0';
33         end process;
34     end Behavioral;

```

Contatore

Il contatore è stato implementato utilizzando un approccio comportamentale. Presenta due modalità di funzionamento: quando *count_fe* si alza, il contatore incrementa il valore di conteggio sul suo fronte di discesa; quando è alto *count_clk*, il valore di conteggio viene calcolato ad ogni fronte di salita del clock.

```

1  entity contatore is
2      port(
3          clk      : in std_logic;
4          reset    : in std_logic;
5          count_fe  : in std_logic;    -- conta i Falling Edge
6          count_clk : in std_logic;   -- conta i colpi di clock
7          count     : out std_logic_vector(2 downto 0)
8      );
9  end contatore;
10
11 architecture Behavioral of contatore is
12
13 signal counter : unsigned(2 downto 0) := (others => '0');
14
15 begin
16
17     count <= std_logic_vector(counter);
18
19     p_clk : process
20     begin
21         wait on clk, reset, count_fe;
22
23         if(reset = '1') then
24             counter <= (others => '0');
25
26         elsif(count_fe = '1') then
27             wait until count_fe = '0';
28             counter <= counter + 1;
29
30         elsif(count_clk = '1') then
31
32             if(clk'event AND clk = '1') then
33                 counter <= counter + 1;
34             end if;
35
36
37         end if;
38
39     end process;
40
41
42
43 end Behavioral;
```

Vector

La componente Vector è stata implementata con un approccio comportamentale. Se l'abilitazione è alta memorizza il valore in ingresso in R_2 sul fronte di discesa del clock, mentre se il segnale di shift è alto trasla i valori R_2 , R_1 ed R_0 .

```

1  entity vector is
2
3      port(
4          clk    : in std_logic;
5          shift   : in std_logic;
6          en     : in std_logic;
7          rst    : in std_logic;
8          data_in : in std_logic_vector(7 downto 0);
9          data_out : out std_logic_vector(7 downto 0)
10     );
11
12 end vector;
13
14 architecture Behavioral of vector is
15 signal r0 : std_logic_vector(7 downto 0) := (others => '0'); -- registro 0
16 signal r1 : std_logic_vector(7 downto 0) := (others => '0'); -- registro 1
17 signal r2 : std_logic_vector(7 downto 0) := (others => '0'); -- registro 2
18 signal s_data_out : std_logic_vector(7 downto 0) := (others => '0');
19
20 begin
21     data_out <= s_data_out;
22
23     p : process(clk, rst)
24
25     begin
26
27         if(rst = '1') then
28             r0 <= (others => '0');
29             r1 <= (others => '0');
30             r2 <= (others => '0');
31
32         elsif(clk'event AND clk = '0') then
33
34             if(en = '1' AND shift = '0') then
35                 r2 <= data_in;
36                 r1 <= r2;
37                 r0 <= r1;
38
39             elsif(en = '0' AND shift = '1') then
40                 r2 <= (others => '0');
41                 r1 <= r2;

```

```

42      r0 <= r1;
43      s_data_out <= r0;
44
45    end if;
46
47  end if;
48
49 end process;
50
51 end Behavioral;

```

ALU

L'ALU è stata realizzata con un approccio comportamentale implementando la struttura in figura 7.6.

```

1 entity alu is
2   port(
3     inA      : in std_logic_vector(7 downto 0);
4     inB      : in std_logic_vector(7 downto 0);
5     en_mult  : in std_logic;
6     en_sum   : in std_logic;
7     clk      : in std_logic;
8     rst      : in std_logic;
9     res      : out std_logic_vector(17 downto 0)
10   );
11 end alu;
12
13 architecture Behavioral of alu is
14
15 -- REGISTRI
16 signal reg : unsigned(17 downto 0) := (others => '0');
17 signal acc : unsigned(17 downto 0) := (others => '0');
18
19 -- INTERCONNESSIONI
20 signal mult_out  : unsigned(15 downto 0) := (others => '0');
21 signal adder_out  : unsigned(17 downto 0) := (others => '0');
22 signal adder_retr : unsigned(17 downto 0) := (others => '0');
23
24 begin
25
26   mult_out <= unsigned(inA) * unsigned(inB);
27   adder_out <= adder_retr + reg;
28   adder_retr <= acc;
29   res <= std_logic_vector(acc);
30
31   p : process(clk, rst)
32     begin

```

```

33
34     if(rst = '1') then
35         reg <= (others => '0');
36         acc <= (others => '0');
37
38     elsif(clk'event AND clk = '0') then
39
40         if(en_mult = '1') then
41             reg <= "00" & mult_out;
42         end if;
43
44         if(en_sum = '1') then
45             acc <= adder_out;
46         end if;
47
48     end if;
49
50
51 end process;
52
53
54 end Behavioral;

```

Unità operativa

L'unità operativa è stata implementata utilizzando un approccio ibrido.

```

1 entity uo is
2     port (
3         busIn    : in std_logic_vector(7 downto 0);
4         clk      : in std_logic;
5         rst      : in std_logic;
6         en_a     : in std_logic;
7         en_b     : in std_logic;
8         en_ris   : in std_logic;
9         shift    : in std_logic;
10        en_mult : in std_logic;
11        en_sum  : in std_logic;
12        res     : out std_logic_vector(17 downto 0)
13    );
14
15 end uo;
16
17 architecture UnitaOperativa of uo is
18
19 component alu is
20     port (
21         inA      : in std_logic_vector(7 downto 0);
22         inB      : in std_logic_vector(7 downto 0);

```

```

23      en_mult    : in std_logic;
24      en_sum     : in std_logic;
25      clk        : in std_logic;
26      rst        : in std_logic;
27      res        : out std_logic_vector(17 downto 0)
28  );
29 end component;
30
31 component vector is
32
33 port(
34   clk      : in std_logic;
35   shift    : in std_logic;
36   en       : in std_logic;
37   rst      : in std_logic;
38   data_in  : in std_logic_vector(7 downto 0);
39   data_out : out std_logic_vector(7 downto 0)
40 );
41
42 end component;
43
44 signal regRis : std_logic_vector(17 downto 0) := (others => '0');
45 signal vectorA_out : std_logic_vector(7 downto 0) := (others => '0');
46 signal vectorB_out : std_logic_vector(7 downto 0) := (others => '0');
47 signal alu_out      : std_logic_vector(17 downto 0) := (others => '0');
48
49 begin
50
51   res <= regRis;
52
53   vectorA : vector
54     port map(
55       clk      => clk,
56       shift    => shift,
57       en       => en_a,
58       rst      => rst,
59       data_in  => busIn,
60       data_out  => vectorA_out
61     );
62
63   vectorB : vector
64     port map(
65       clk      => clk,
66       shift    => shift,
67       en       => en_b,
68       rst      => rst,
69       data_in  => busIn,
70       data_out  => vectorB_out
71     );

```

```

72
73     al : alu
74     port map(
75         inA    => vectorA_out,
76         inB    => vectorB_out,
77         en_mult => en_mult,
78         en_sum  => en_sum,
79         clk     => clk,
80         rst     => rst,
81         res     => alu_out
82     );
83
84
85     p : process(clk)
86     begin
87
88         if(clk'event AND clk = '0') then
89
90             if(en_ris = '1') then
91                 regRis <= alu_out;
92             end if;
93
94         end if;
95
96     end process;
97
98 end UnitaOperativa;

```

MicroRom

La microRom implementa la *control store* raffigurata dalla tabella in figura 7.12.

```

1  entity MicroRom is
2      port (
3          pc      : in std_logic_vector(3 downto 0);
4          next_state : out std_logic_vector(3 downto 0);
5          Jcount   : out std_logic;
6          Jack     : out std_logic;
7          base     : out std_logic_vector(2 downto 0);
8          read1    : out std_logic;
9          enA      : out std_logic;
10         enB      : out std_logic;
11         next_op   : out std_logic;
12         save_res  : out std_logic;
13         reset_count : out std_logic
14     );
15 end MicroRom;
16
17 architecture ImpMR of MicroRom is

```

```

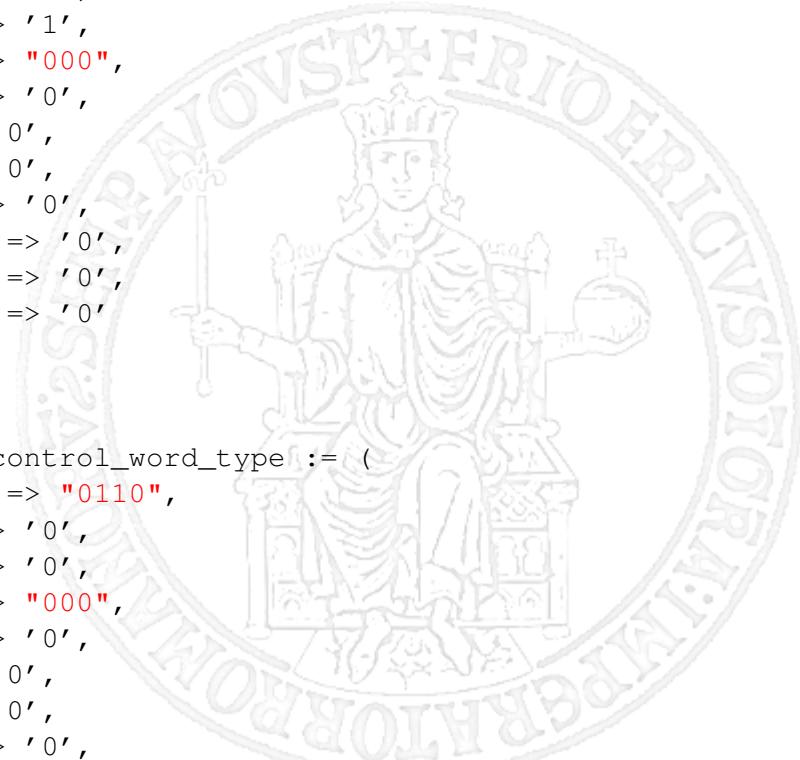
18
19 type control_word_type is record
20     next_state : std_logic_vector(3 downto 0);
21     Jcount      : std_logic;
22     Jack        : std_logic;
23     base        : std_logic_vector(2 downto 0);
24     read1       : std_logic;
25     enA         : std_logic;
26     enB         : std_logic;
27     next_op     : std_logic;
28     save_res    : std_logic;
29     reset_uc    : std_logic;
30     reset_count : std_logic;
31 end record;
32
33 constant idle : control_word_type := (
34     next_state  => "0000",
35     Jcount      => '0',
36     Jack        => '0',
37     base        => "000",
38     read1       => '0',
39     enA         => '0',
40     enB         => '0',
41     next_op     => '0',
42     save_res    => '0',
43     reset_uc    => '0',
44     reset_count => '0'
45 );
46
47 constant reset : control_word_type := (
48     next_state  => "0010",
49     Jcount      => '0',
50     Jack        => '0',
51     base        => "000",
52     read1       => '0',
53     enA         => '0',
54     enB         => '0',
55     next_op     => '0',
56     save_res    => '0',
57     reset_uc    => '1',
58     reset_count => '1'
59 );
60
61 constant selA : control_word_type := (
62     next_state  => "0011",
63     Jcount      => '0',
64     Jack        => '1',
65     base        => "000",
66     read1       => '1',

```

```

67     enA      => '0',
68     enB      => '0',
69     next_op   => '0',
70     save_res  => '0',
71     reset_uc  => '0',
72     reset_count => '0'
73 );
74
75 constant s_enA : control_word_type := (
76     next_state  => "0100",
77     Jcount      => '0',
78     Jack        => '0',
79     base         => "000",
80     read1       => '1',
81     enA         => '1',
82     enB         => '0',
83     next_op     => '0',
84     save_res    => '0',
85     reset_uc    => '0',
86     reset_count => '0'
87 );
88
89 constant wait_ack_a : control_word_type := (
90     next_state  => "0101",
91     Jcount      => '1',
92     Jack        => '1',
93     base         => "000",
94     read1       => '0',
95     enA         => '0',
96     enB         => '0',
97     next_op     => '0',
98     save_res    => '0',
99     reset_uc    => '0',
100    reset_count => '0'
101 );
102
103
104 constant rcB : control_word_type := (
105     next_state  => "0110",
106     Jcount      => '0',
107     Jack        => '0',
108     base         => "000",
109     read1       => '0',
110     enA         => '0',
111     enB         => '0',
112     next_op     => '0',
113     save_res    => '0',
114     reset_uc    => '0',
115     reset_count => '1'

```



```

116 );
117
118
119 constant selB : control_word_type := (
120     next_state    => "0111",
121     Jcount        => '0',
122     Jack          => '1',
123     base          => "011",
124     read1         => '1',
125     enA           => '0',
126     enB           => '0',
127     next_op       => '0',
128     save_res      => '0',
129     reset_uc      => '0',
130     reset_count   => '0'
131 );
132
133 constant s_enB : control_word_type := (
134     next_state    => "1000",
135     Jcount        => '0',
136     Jack          => '0',
137     base          => "011",
138     read1         => '1',
139     enA           => '0',
140     enB           => '1',
141     next_op       => '0',
142     save_res      => '0',
143     reset_uc      => '0',
144     reset_count   => '0'
145 );
146
147 constant wait_ack_b : control_word_type := (
148     next_state    => "1001",
149     Jcount        => '1',
150     Jack          => '1',
151     base          => "011",
152     read1         => '0',
153     enA           => '0',
154     enB           => '0',
155     next_op       => '0',
156     save_res      => '0',
157     reset_uc      => '0',
158     reset_count   => '0'
159 );
160
161 constant rc_op : control_word_type := (
162     next_state    => "1010",
163     Jcount        => '0',
164     Jack          => '0',

```

```

165     base      => "000",
166     read1     => '0',
167     enA       => '0',
168     enB       => '0',
169     next_op   => '0',
170     save_res  => '0',
171     reset_uc  => '0',
172     reset_count => '1'
173 );
174
175 constant op : control_word_type := (
176     next_state  => "1011",
177     Jcount      => '1',
178     Jack        => '0',
179     base        => "000",
180     read1       => '0',
181     enA         => '0',
182     enB         => '0',
183     next_op     => '1',
184     save_res    => '0',
185     reset_uc   => '0',
186     reset_count => '0'
187 );
188
189 constant res : control_word_type := (
190     next_state  => "0000",
191     Jcount      => '0',
192     Jack        => '0',
193     base        => "000",
194     read1       => '0',
195     enA         => '0',
196     enB         => '0',
197     next_op     => '0',
198     save_res    => '1',
199     reset_uc   => '0',
200     reset_count => '0'
201 );
202
203 type rom_type is array(0 to 11) of control_word_type;
204
205 constant rom : rom_type := (
206     0 => idle,
207     1 => reset,
208     2 => selA,
209     3 => s_enA,
210     4 => wait_ack_a,
211     5 => rcb,
212     6 => selB,
213     7 => s_enB,

```

```

214     8 => wait_ack_b,
215     9 => rc_op,
216    10 => op,
217    11 => res
218 );
219
220 signal cw : control_word_type;
221
222 begin
223
224     cw <= rom(to_integer(unsigned(pc)));
225
226     next_state <= cw.next_state;
227     Jcount      <= cw.jcount;
228     Jack        <= cw.jack;
229     base         <= cw.base;
230     read1       <= cw.read1;
231     enA         <= cw.enA;
232     enB         <= cw.enB;
233     next_op     <= cw.next_op;
234     save_res    <= cw.save_res;
235     reset_count <= cw.reset_count;
236
237 end ImpMR;

```

Unità di controllo

L'unità di controllo è stata realizzata utilizzando un approccio comportamentale.

```

1  entity uc is
2   port(
3     clk      : in std_logic;
4     reset    : in std_logic;
5     ack_memory : in std_logic;
6     count    : in std_logic_vector(2 downto 0);
7     start    : in std_logic;
8     sel      : out std_logic_vector(2 downto 0);
9     read1   : out std_logic;
10    enA     : out std_logic;
11    enB     : out std_logic;
12    next_op  : out std_logic;
13    save_res : out std_logic;
14    reset_uc : out std_logic;
15    reset_count : out std_logic
16  );
17 end uc;
18
19 architecture Behavioral of uc is
20

```

```

21 component MicroRom is
22   port(
23     pc      : in std_logic_vector(3 downto 0);
24     next_state : out std_logic_vector(3 downto 0);
25     Jcount    : out std_logic;
26     Jack      : out std_logic;
27     base      : out std_logic_vector(2 downto 0);
28     read1     : out std_logic;
29     enA       : out std_logic;
30     enB       : out std_logic;
31     next_op   : out std_logic;
32     save_res  : out std_logic;
33     reset_count : out std_logic
34   );
35 end component;
36
37
38 signal s_ns : std_logic_vector(3 downto 0) := (others => '0');
39 signal s_jcount, s_jack : std_logic := '0';
40 signal s_base : std_logic_vector(2 downto 0) := (others => '0');
41 signal s_pc : std_logic_vector(3 downto 0) := (others => '0');
42
43 signal pc_next : std_logic_vector(3 downto 0) := (others => '0');
44 signal s_next_op : std_logic := '0';
45
46 begin
47
48   mr : MicroRom
49     port map(
50       pc      => s_pc,
51       next_state  => s_ns,
52       Jcount    => s_jcount,
53       Jack      => s_jack,
54       base      => s_base,
55       read1     => read1,
56       enA       => enA,
57       enB       => enB,
58       next_op   => s_next_op,
59       save_res  => save_res,
60       reset_count => reset_count
61     );
62
63   -- ADDER
64   sel <= std_logic_vector(unsigned(s_base) + unsigned(count));
65   next_op <= s_next_op;
66
67   l11 : process(reset, clk)
68   begin
69

```

```

70      if(reset = '1') then
71          s_pc <= (others => '0');
72
73      elsif(clk'event AND clk = '0') then
74
75          if(start = '1') then
76              s_pc <= "0001";
77
78          else
79              s_pc <= pc_next;
80          end if;
81      end if;
82
83  end process;
84
85  li2 : process(clk)
86 begin
87
88     if(clk'event AND clk = '1') then
89
90         if(s_jcount = '0' AND s_jack = '0') then
91             pc_next <= s_ns;
92
93         elsif(s_jcount = '0' AND s_jack = '1') then
94             if(ack_memory = '1') then
95                 pc_next <= s_ns;
96             end if;
97
98             -- STATO DI op
99         elsif(s_jcount = '1' and s_jack = '0') then
100
101             -- STATO OP
102             if(s_next_op = '1' AND count = "100") then
103                 pc_next <= s_ns;
104             end if;
105
106             -- STATO DI WAIT_ACK
107         else
108
109             if(ack_memory = '0') then
110                 if(count = "011") then
111                     pc_next <= s_ns;
112                 else
113                     -- WAIT_ACK_A
114                     if(s_pc = "0100") then
115                         pc_next <= "0010";
116
117                     -- WAIT_ACK_B
118                     elsif(s_pc = "1000") then

```

```

119         pc_next <= "0110";
120     end if;
121   end if;
122 end if;
123
124 end if;
125
126 end if; -- CLK
127 end process;
128
129
130 end Behavioral;

```

Sistema Completo

Nel sistema completo, sviluppato con un approccio strutturale, sono state collegate in modo opportuno le componenti descritte in precedenza, come mostrato in figura

```

1 entity sistema_prodotto_scalare is
2   port (
3     clk : in std_logic;
4     start : in std_logic;
5     reset : in std_logic;
6     res   : out std_logic_vector(17 downto 0)
7   );
8 end sistema_prodotto_scalare;
9
10 architecture Structural of sistema_prodotto_scalare is
11
12 component Memoria is
13   port (
14     sel      : in std_logic_vector(2 downto 0);
15     read1    : in std_logic;
16     data     : out std_logic_vector(7 downto 0);
17     data_ready : out std_logic
18   );
19 end component;
20
21 component uc is
22   port (
23     clk      : in std_logic;
24     reset    : in std_logic;
25     ack_memory : in std_logic;
26     count    : in std_logic_vector(2 downto 0);
27     start    : in std_logic;
28     sel      : out std_logic_vector(2 downto 0);
29     read1   : out std_logic;
30     enA     : out std_logic;

```

```

31      enB      : out std_logic;
32      next_op   : out std_logic;
33      save_res  : out std_logic;
34      reset_uc  : out std_logic;
35      reset_count : out std_logic
36  );
37 end component;
38
39 component uo is
40  port(
41    busIn   : in std_logic_vector(7 downto 0);
42    clk     : in std_logic;
43    rst     : in std_logic;
44    en_a    : in std_logic;
45    en_b    : in std_logic;
46    en_ris  : in std_logic;
47    shift   : in std_logic;
48    en_mult : in std_logic;
49    en_sum  : in std_logic;
50    res     : out std_logic_vector(17 downto 0)
51  );
52
53 end component;
54
55 component contatore is
56  port(
57    clk      : in std_logic;
58    reset    : in std_logic;
59    count_fe : in std_logic; -- conta i Falling Edge
60    count_clk : in std_logic; -- conta i colpi di clock
61    count    : out std_logic_vector(2 downto 0)
62  );
63 end component;
64
65 signal s_reset_for_cont, s_read1, s_reset_uc, s_ack_memory, s_enA, s_enB,
66   s_next_op, s_save_res, s_reset_count, s_reset : std_logic := '0';
67 signal s_sel : std_logic_vector(2 downto 0) := (others => '0');
68 signal busIn : std_logic_vector(7 downto 0) := (others => '0');
69 signal s_count : std_logic_vector(2 downto 0) := (others => '0');
70
71 begin
72
73   mem : Memoria
74   port map(
75     sel      => s_sel,
76     read1   => s_read1,
77     data     => busIn,
78     data_ready => s_ack_memory

```

```

79      );
80
81  unita_controllo : uc
82    port map(
83      clk      => clk,
84      reset    => reset,
85      ack_memory => s_ack_memory,
86      count     => s_count,
87      start     => start,
88      sel       => s_sel,
89      read1    => s_read1,
90      enA      => s_enA,
91      enB      => s_enB,
92      next_op   => s_next_op,
93      save_res  => s_save_res,
94      reset_uc  => s_reset_uc,
95      reset_count=> s_reset_count
96    );
97
98  s_reset <= s_reset_uc OR reset;
99  unita_operativa : uo
100   port map(
101     busIn    => busIn,
102     clk      => clk,
103     rst      => s_reset,
104     en_a     => s_enA,
105     en_b     => s_enB,
106     en_ris   => s_save_res,
107     shift    => s_next_op,
108     en_mult  => s_next_op,
109     en_sum   => s_next_op,
110     res      => res
111   );
112
113  s_reset_for_cont <= s_reset_uc OR s_reset_count OR reset;
114  cont : contatore
115    port map(
116      clk      => clk,
117      reset    => s_reset_for_cont,
118      count_fe  => s_read1,
119      count_clk => s_next_op,
120      count     => s_count
121   );
122
123 end Structural;

```

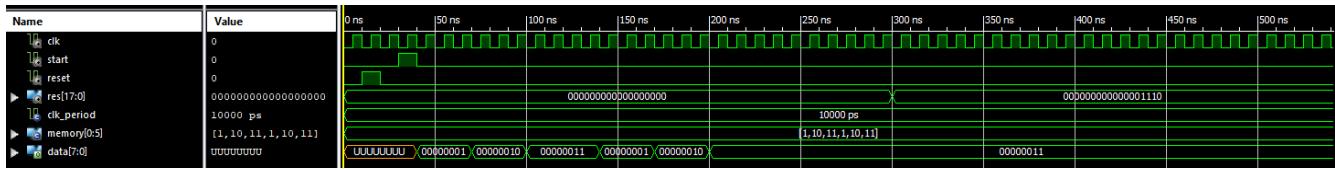


Figura 7.14: Simulazione del sistema

7.4 Simulazione

Per effettuare la simulazione di entrambi gli approcci implementati, si è realizzato il testbench riportato qui di seguito. In questo caso è stato utilizzato un assert per verificare la correttezza del risultato finale. In figura 7.14 è rappresentato il risultato della simulazione, in particolare sono stati riportati i seguenti segnali:

- Res: Mostra il risultato finale.
- Memory: Mostra i valori contenuti nella memoria.
- Data: Segnale che passa i valori contenuti in memoria all'Unità Operativa.

```

1 stim_proc: process
2 begin
3     -- hold reset state for 100 ns.
4     wait for 10 ns;
5     reset <= '1';
6
7     wait for 10 ns;
8     reset <= '0';
9
10    wait for 10 ns;
11    start <= '1';
12
13    wait for 10 ns;
14    start <= '0';
15
16    wait for 300 ns;
17    assert res = "0000000000000110" report "Risultato Errato" severity
18        warning;
19
20    wait;
21 end process;
22
END;
```

Capitolo 8

Processore Mic-1

8.1 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.

8.2 Programma

Per effettuare l'analisi dell'architettura del processore MIC-1 si è sviluppato un semplice programma C il cui codice è qui di seguito riportato.

```
1 int a = 5;
2 int b = 10;
3 int c;
4 if(a < b)
5     c = 1;
6 else
7     c = 0;
```

Lo scopo è quello di confrontare due variabili: a e b. In c viene salvato 1 se a è più piccolo b, altrimenti 0. A questo punto, il codice è stato tradotto in linguaggio assembler con le istruzioni IJVM.

```
1 .main
2 .var
3 a
4 .endvar
5 BIPUSH 0x05      #A
6 BIPUSH 0x07      #B
```

```

7 ISUB
8 IFLT L1      #IF A < B SAVE 1 ELSE SAVE 0
9 BIPUSH 0x00
10 GOTO L2
11 L1:
12   BIPUSH 0x01
13 L2:
14   ISTORE a
15 .endmethod

```

Come è possibile osservare dal codice, viene allocata una variabile locale a; dopodiché, attraverso l'istruzione BIPUSH si inseriscono in cima allo stack i valori 5 e 7. Per effettuare il confronto tra questi due valori, l'idea è quella di sottrarre il primo elemento con il secondo; se il risultato di tale operazione è negativo allora 5 è minore di 7. Per implementare il costrutto if..then..else si utilizzano delle istruzioni di salto. In particolare, è stata utilizzata l'istruzione IFLT che effettua una pop dallo stack e salta alla label L1 se il valore estratto è negativo, altrimenti si prosegue con l'istruzione successiva. Infine, attraverso l'istruzione ISTORE, si estraе la parola in cima allo stack e la si memorizza nella variabile locale a.

8.2.1 RAM

Compreso come da un linguaggio di programmazione si è passato al relativo codice assembler, affrontiamo come da un programma assembler viene generato un file binario, eseguibile dal MIC-1, col compilatore assembler per l'ISA della IJVM.

La stringa di bit viene memorizzata in una RAM dalla quale il processore legge direttamente le istruzioni da eseguire. L'indirizzo della prossima istruzione è memorizzato nel registro PC del processore. In figura 8.1 è rappresentato il codice binario relativo al programma sviluppato. Nel codice binario presente in figura 8.2, i bit sono stati raggruppati a gruppi di 8 ed è stato inserita la corrispondente istruzione IJVM. Leggendo da destra verso sinistra e dall'alto verso il basso, è possibile notare come le istruzioni seguano fedelmente il programma assembler. Da notare come la word 0 sia l'istruzione che consente l'allocazione delle variabili locali specificando il limite inferiore (primi 16 bit) e il limite superiore (ultimi 16 bit) degli indirizzi delle variabili allocate. In questo caso, vi è un'unica variabile locale allocata, pertanto il limite superiore ed inferiore coincidono con 0x01.

Un'altra particolarità sono le istruzioni di salto. Infatti, leggendo la word 2 dove è presente

```

128 => "00000000000000000000000000000000"
0 => "00000001000000000000000010000000",
1 => "000001100010000000010100010000",
2 => "0000100000000001001110101011100",
3 => "0000000010100111000000000010000",
4 => "001101100000001000100000000101",
5 => "00000000000000000000000000000001",

```

Figura 8.1: Traduzione del programma assembler in bit memorizzato nella RAM

```

128 => "00000000000000000000000000000000",
0 => "00000001 00000000 00000001 00000000", -- Allocazione variabile a
1 => "(0x7 00000111) (BIPUSH 00010000) (0x05 00000101) (BIPUSH 00010000)",
2 => "(JMP 8 00001000) 00000000 (IFLT 10011101) (ISUB 01011100)",
3 => "00000000 (GOTO 10100111) (0x00 00000000) (BIPUSH 00010000)",
4 => "(ISTORE 00110110) (0x01 00000001) (BIPUSH 00010000) (JMP 5 00000101)",
5 => "00000000 00000000 00000000 (VAR a 00000001)",

```

Figura 8.2: Codice binario del programma commentato

l'istruzione IFLT, è possibile notare come dopo un'istruzione di salto segue sempre un byte nullo e l'offset del salto. Considerazione analoga per l'istruzione GOTO nella word 3.

8.3 Simulazione del programma sul processore Mic-1

Per analizzare l'architettura del processore Mic-1 si è sfruttato il programma descritto nel paragrafo precedente.

Questo paragrafo descrive una prima simulazione effettuata che non ha dato il risultato atteso. Segue pertanto un'analisi per identificare il problema e la soluzione adottata.

Per le rappresentazioni delle forme d'onda dei segnali generati, è stato utilizzato il software *GKWave*.

8.3.1 Prima simulazione

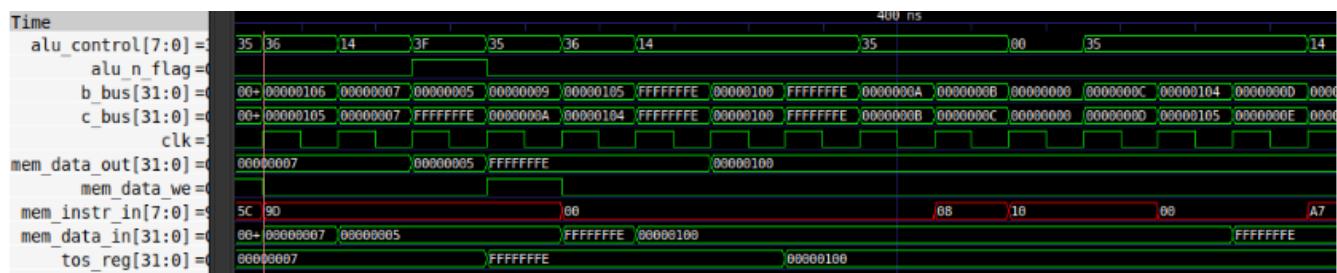


Figura 8.3: Segnali generati nella prima simulazione

Simulando il programma mostrato precedentemente, si è subito notato un problema dell'architettura per quanto riguarda i salti condizionati. Infatti, dato che sullo stack sono stati memorizzati i valori 5 e 7, ci si aspettava che in *a* venisse salvato 1 piuttosto che 0. Invece, come è possibile osservare in figura 8.3, in *a* viene salvato 0.

Concentrandosi sul segnale *mem_instr_in* che indica l'istruzione entrante nel processore nel registro MBR, si possono osservare tutte le istruzioni eseguite.

Partendo dall'istruzione 0x9D, ovvero IFLT, seguono il byte nullo come spiegato precedentemente e l'offset di salto; dopodiché vi è una BIPUSH (0x10) seguita dal valore 0x00 quando, in realtà, ci si aspetta 0x01. Osservando con attenzione il segnale *alu_n_flag*, si può notare che il segnale di stato assume un'unica volta il valore 1, ovvero durante l'esecuzione dell'istruzione ISUB (0x5C). Da qui è possibile comprendere come mai il programma non dia il risultato atteso: se il bit N non è alto durante l'esecuzione di IFLT, il salto condizionato non avverrà e sarà inserito nello stack il valore 0x00.

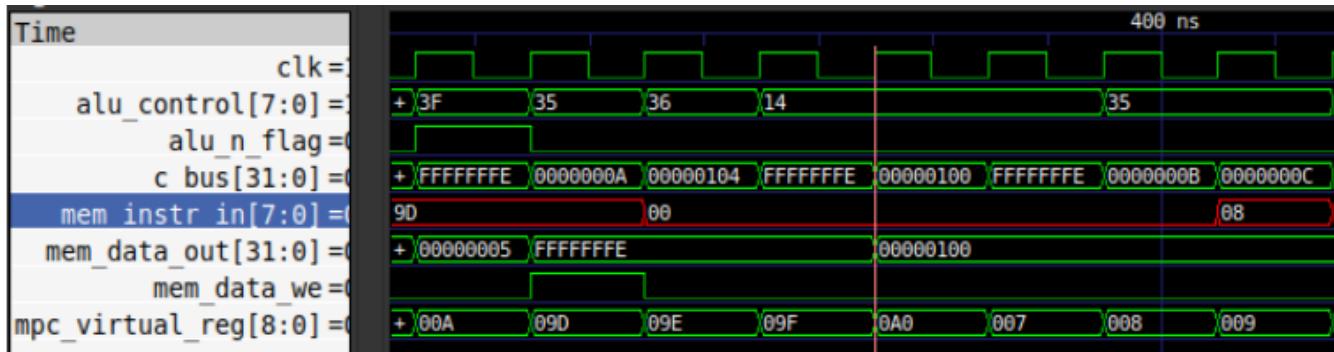


Figura 8.4: Simulazione che evidenzia l’evoluzione del MicroProgramCounter e del flag N

In aggiunta, a rafforzare l’ipotesi di un errore nell’architettura del sistema, vi è un’ulteriore osservazione. Quando sul bus C transita il valore negativo -2 (FFFFFFFE) in uscita dall’ALU, *alu_n_flg* assume erroneamente un valore basso.

8.3.2 Identificazione e risoluzione del problema

```

1 iflt = 0x9D:
2     MAR = SP = SP - 1; rd
3     OPC = TOS
4     TOS = MDR
5     N = OPC; if (N) goto T; else goto F

```

Dato che il problema è dovuto al salto condizionato con l’istruzione IFLT, è stato analizzato in dettaglio il suo funzionamento.

Prima di tutto, per effettuare il POP del valore in testa allo stack, si decrementa lo stack-pointer e si effettua la lettura del nuovo valore puntato. Successivamente, in OPC viene inserito il valore salvato in TOS, ovvero il valore dell’elemento in testa allo stack, e in TOS viene salvato il nuovo valore appena letto dalla memoria.

Nell’ultima microistruzione si controlla il valore salvato in OPC. Se esso è negativo, ovvero se $N = 1$, si salta all’istruzione T, altrimenti all’istruzione F. Da quest’analisi ci si aspetta che durante l’esecuzione delle istruzioni 0x9E e 0xA0, poiché in cima allo stack è salvato il valore -2, il bit N sia pari ad 1. Ciò non avviene come è possibile osservare dalla simulazione in figura 8.4. L’unica volta in cui il valore di N è uguale a 1 è durante l’istruzione 0x0A che è una microistruzione di ISUB che effettua la differenza tra il primo valore in cima allo stack e il successivo. Pertanto, si può ipotizzare che l’errore sia proprio nell’ALU, poiché quando un valore negativo transita dal bus B al bus C, passando inevitabilmente per l’ALU, il bit N non assume il valore atteso.

Fatta questa osservazione, si è analizzato il codice VHDL che implementa l’ALU. Poiché il problema si limita ad una scorretta gestione del flag N, ci si è concentrati sull’evoluzione del segnale negative_flag. Esso, come è possibile notare dalla riga 121 del codice VHDL, dipende dal 32esimo bit di t_sum che, con un abuso di linguaggio, rappresenta il bit di segno.

t_sum è un segnale interno che rappresenta la somma algebrica tra gli operandi in ingresso all’ALU ed un segnale di incremento $F_0F_1 = 11$. Questi sono il quinto ed il quarto bit del segnale di controllo in ingresso all’ALU che determinano l’operazione da eseguire.

F ₀	F ₁	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	\bar{A}
1	0	1	1	0	0	\bar{B}
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B - A
1	1	0	1	1	0	B - 1
1	1	1	0	1	1	-A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	-1

Figura 8.5: Segnali di controllo ALU e relative funzioni¹

Il valore di t_sum viene posto a 0 qualora l'ALU effettui operazioni diverse dalla somma. È proprio questo che accade quando un elemento dal bus B transita inalterato al bus C: l'unità di controllo invia il segnale di controllo all'ALU 0x14 (00 010100), dove i primi due bit più significativi controllano lo shift register. Traducendo tale segnale con la tabella in figura 8.5, l'ALU esegue l'istruzione B, ovvero fa transitare inalterato il valore presente sul bus B verso il bus C. Pertanto, il controllo del bit di negatività non deve avvenire solamente quando si effettua un'operazione di somma, ma ogni qual volta il valore di uscita dell'ALU è interpretato come negativo. Quindi, il codice in linea 121 è stato sostituito con:

$$\text{negative_flag} \leq t_result(31)$$

```

113
114  with fn select t_result <=
115    t_and  when alu_fn_and,
116    t_or   when alu_fn_or,
117    t_not_b when alu_fn_not_b,
118    t_sum   when others;
119
120  -- ALU flags
121  negative_flag <= t_result(31); -- precedentemente t_sum
122  zero_flag     <= '1' when t_result = x"00000000" else '0';
123
124  -- Shifter
125  with sh select sh_result <=
126    t_result(23 downto 0) & x"00"      when alu_sh_sll8,
127    t_result(31) & t_result(31 downto 1) when alu_sh_sra1,
128    t_result                           when others;
129

```

Figura 8.6: Codice VHDL per l'implementazione dell'ALU dell'Ingegnere Alberto Moriconi (modificato)



Figura 8.7: Segnali generati nella seconda simulazione

8.3.3 Seconda simulazione

Dopo aver modificato il codice VHDL dell'ALU è possibile notare dalla simulazione la corretta esecuzione della condizione di salto, infatti nella variabile a viene pushato il valore corretto, cioè 0x01.

Inoltre, il segnale *alu_n_flag* assume valore 1 non soltanto durante l'esecuzione di ISUB, come avveniva prima della modifica, bensì ogni volta che sul bus C è transitato un valore negativo, come 0xFFFFFFF.



¹Immagine tratta dal materiale del corso

ADDR		JAM		ALU				C				MEM		B														
NEXT ADDRESS		J m P C	J A M N	J m P C	S L R 8	S R L 1	F ₀	F ₁	E N A	E N B	I N V A	I N C	H	O P C	T O S	C P P	L V	S P	P C	M D R	M A R	W R D	F E T CH	B B US				
0	0	1	0	1	1	1	0	1	0	0	0	1	1	0	1	1	0	0	0	0	1	0	0	1	0	0	0	0
0	0	1	0	1	1	1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	1	1	0	0	1	0	0	1	0	0	0	0	0

Figura 8.8: Sequenza delle microistruzioni per il codice operativo ISUB

8.4 Flusso di esecuzione dell'istruzione ISUB

L'istruzione ISUB (0x5C) sostituisce le due parole in cima allo stack con la loro differenza. Di seguito è riportata la sua notazione mal.

```

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

```

Viene innanzitutto decrementato lo stack-pointer, così da puntare e leggere l'elemento successivo alla testa. Lo stack-pointer decrementato viene inserito anche nel registro MAR, poiché punta alla locazione che non solo contiene il secondo elemento in cima allo stack ma è la locazione di memoria dove sarà salvato il risultato della differenza.

Con la microistruzione seguente, viene caricata in H la cima dello stack, memorizzata nel registro TOS.

Infine, viene eseguita l'effettiva sottrazione e il risultato è caricato sia in TOS che in MDR, in modo tale che TOS possa sempre contenere la copia dell'elemento in cima allo stack. Segue un'operazione di scrittura in memoria nella locazione puntata da MAR. Nella tabella in figura 8.8 è riportato il relativo codice binario delle tre microistruzioni. La regione interessata della *control store* inizia all'indirizzo 0x5C e termina all'indirizzo 0x5E.

0x5C MAR = SP = SP - 1; rd

- NEXT ADDRESS 001011101 = 0x5D: la prossima microistruzione è la successiva.
- JAM 000: non è necessario controllare alcun flag per effettuare un salto condizionato.
- ALU 00110110: l'ALU effettua il decremento del valore posto sul bus B.
- Bus C, SP = 1 e MAR = 1: il dato posto sul bus C è memorizzato nei registri SP e MAR.
- MEM RD = 1: si invia alla memoria il comando di lettura.
- Bus B 0100: il registro SP è abilitato a scrivere sul bus B.

0x5D H = TOS

- NEXT ADDRESS 001011110 = 0x5E: la prossima microistruzione è la successiva.
- JAM 000: non è necessario controllare alcun flag per effettuare un salto condizionato.

- ALU 00010100: l'ALU lascia passare invariato il valore scritto sul bus B.
- Bus C, H = 1: il dato posto sul bus C è memorizzato nel registro H.
- MEM 000: non viene effettuata alcuna operazione di prelievo o di scrittura in memoria.
- Bus B 0111: il registro TOS è abilitato a scrivere sul bus B.

0x5E MDR = TOS = MDR - H; wr; goto main

- NEXT ADDRESS 00000101 = 0x05: la prossima microistruzione è il main.
- JAM 000: non è necessario controllare alcun flag per effettuare un salto condizionato.
- ALU 00111111: l'ALU effettua la sottrazione B - A.
- Bus C TOS = 1: il dato posto sul bus C è memorizzato nel registro TOS.
- MEM WR = 1: si invia alla memoria il comando di scrittura.
- B BUS 0100: il registro MDR è abilitato a scrivere sul bus B.

Invece, nelle figure successive sono rappresentati graficamente i movimenti tra i registri all'interno dell'unità operativa.

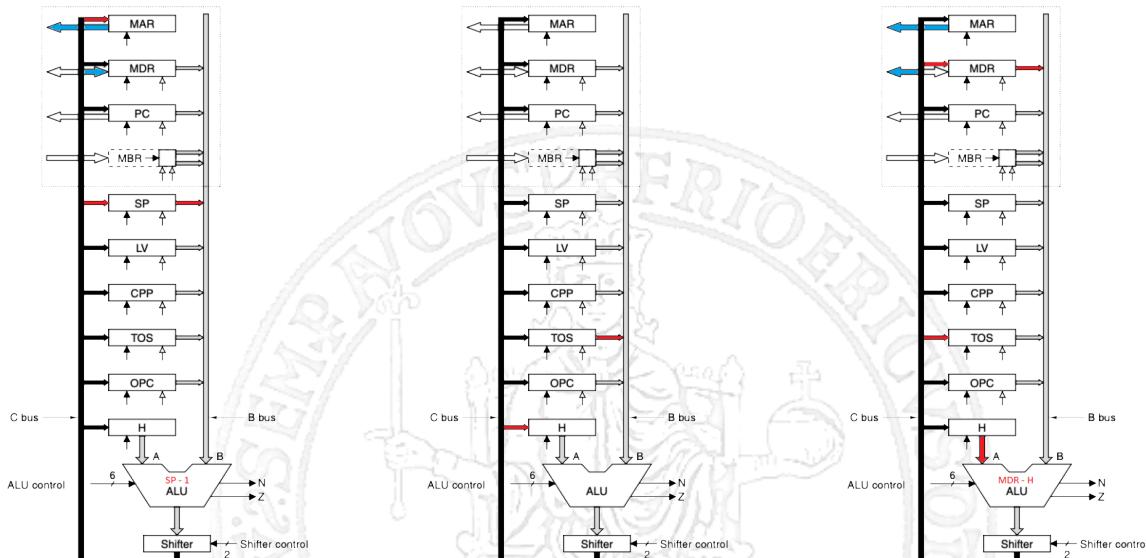


Figura 8.9: Flusso di esecuzione di ISUB

ADDR		JAM		ALU				C				MEM		B																
NEXT ADDRESS		J m P C	J A M C	J m P C	S L A 8	S R A 1	F ₀	F ₁	E N A	E N B	I N V A	I N C	H	O P C	T O S	C P P	L V	S P	P C	M D R	M A R	W R	R D	F E T CH	B B US					
0	0	0	1	1	0	1	1	1	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1	0	1			
0	0	0	1	1	1	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	
0	0	0	1	1	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	1
0	0	0	1	1	1	0	1	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0	0	1	0	0	1	0	0	0
0	0	0	1	1	1	0	1	1	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0

Figura 8.10: Sequenza delle microistruzioni per il codice operativo ISTORE

8.5 Flusso di esecuzione dell'istruzione ISTORE

L'istruzione ISTORE (0x36) estraie la parola in cima allo stack e la memorizza in una variabile locale e come operando richiede il nome della variabile stessa, che è un offset rispetto ad LV.

```

1 istore = 0x36:
2     H = LV
3     MAR = MBRU + H
4 istore_cont:
5     MDR = TOS; wr
6     SP = MAR = SP - 1; rd
7     PC = PC + 1; fetch
8     TOS = MDR; goto main

```

Per cominciare, l'indirizzo salvato in LV, che punta alla base dell'area di memoria delle variabili locali, viene caricato nel registro H. L'istruzione successiva somma l'indirizzo inserito in H con l'offset presente nel registro MBRU, all'interno del quale vi è salvato l'operando di ISTORE nel momento dell'esecuzione di questa microistruzione. Tale somma, che rappresenta l'indirizzo di memoria della variabile locale, viene salvata nel MAR. In seguito, in MDR viene copiato TOS, così da poter scrivere nell'area di memoria puntata da MAR il valore presente sulla cima dello stack. Segue il decremento dello stack-pointer, così da effettuare correttamente la pop dallo stack del valore salvato in memoria, e la lettura del valore puntato da SP-1 così da poterlo salvare nel registro TOS che deve sempre contenere il valore puntato dallo SP alla fine dell'istruzione. Prima di effettuare quest'ultima operazione e tornare al main, si incrementa il PC e si esegue l'operazione di fetch, così da avere in MBR l'indirizzo della prossima istruzione successiva ad ISTORE. Se non si effettuasse questa operazione, ritornati nel main si inserirebbe nel MPC l'operando della ISTORE piuttosto che l'indirizzo della prossima istruzione da eseguire, con effetti impredicibili. Nella tabella in figura 8.10 è riportato il relativo codice binario delle microistruzioni che compongono ISTORE. La regione interessata della *control store* comincia all'indirizzo 0x36 e termina all'indirizzo 0x3B.

0x36 H = LV

- NEXT ADDRESS 000110111 = 0x37: la prossima microistruzione è la successiva.
- JAM 000: non è necessario controllare alcun flag per effettuare un salto condizionato.

- ALU 00010100: l'ALU lascia passare invariato il valore scritto sul bus B.
- Bus C, H = 1: il dato posto sul bus C è memorizzato nel registro H.
- MEM 000: non viene effettuata alcuna operazione di prelievo o di scrittura in memoria.
- Bus B 0101: il registro LV è abilitato a scrivere sul bus B.

0x37 MAR = MBRU + H

- NEXT ADDRESS 000111000 = 0x38: la prossima microistruzione è la successiva.
- JAM 000: non è necessario controllare alcun flag per effettuare un salto condizionato.
- ALU 00111100: l'ALU effettua l'operazione di somma A + B.
- Bus C, MAR = 1: il dato posto sul bus C è memorizzato nel registro MAR.
- MEM 000: non viene effettuata alcuna operazione di prelievo o di scrittura in memoria.
- Bus B 0011: il registro MBRU è abilitato a scrivere sul bus B.

0x38 MDR = TOS; wr

- NEXT ADDRESS 000111001 = 0x39: la prossima microistruzione è la successiva.
- JAM 000: non è necessario controllare alcun flag per effettuare un salto condizionato.
- ALU 00010100: l'ALU lascia passare invariato il valore scritto sul bus B.
- Bus C, MDR = 1: il dato posto sul bus C è memorizzato nel registro MDR.
- MEM WE = 1: si invia alla memoria il comando di scrittura.
- Bus B 0111: il registro TOS è abilitato a scrivere sul bus B.

0x39 SP = MAR = SP - 1; rd

- NEXT ADDRESS 000111010 = 0x3A: la prossima microistruzione è la successiva.
- JAM 000: non è necessario controllare alcun flag per effettuare un salto condizionato.
- ALU 00110110: l'ALU decrementa di 1 il valore scritto sul bus B.
- Bus C, SP = 1 MAR = 1: il dato posto sul bus C è memorizzato nei registri SP e MAR.
- MEM RD = 1: si invia alla memoria il comando di lettura.
- Bus B 0100: il registro SP è abilitato a scrivere sul bus B.

0x3A PC = PC + 1; fetch

- NEXT ADDRESS 000111011 = 0x3B: la prossima microistruzione è la successiva.
- JAM 000: non è necessario controllare alcun flag per effettuare un salto condizionato.
- ALU 00110101: l'ALU incrementa di 1 il valore scritto sul bus B.
- Bus C, PC = 1: il dato posto sul bus C è memorizzato nel registro PC.
- MEM FETCH = 1: si esegue l'operazione fetch.

- Bus B 0001: il registro PC è abilitato a scrivere sul bus B.

0x3B TOS = MDR; goto main

- NEXT ADDRESS 000001010 = 0x0A: la prossima microistruzione è il main.
- JAM 000: non è necessario controllare alcun flag per effettuare un salto condizionato.
- ALU 00010100: l'ALU lascia passare invariato il valore scritto sul bus B.
- Bus C, TOS = 1: il dato posto sul bus C è memorizzato nel registro TOS.
- MEM 000: non viene effettuata alcuna operazione di prelievo o di scrittura in memoria.
- Bus B 0000: il registro MDR è abilitato a scrivere sul bus B.

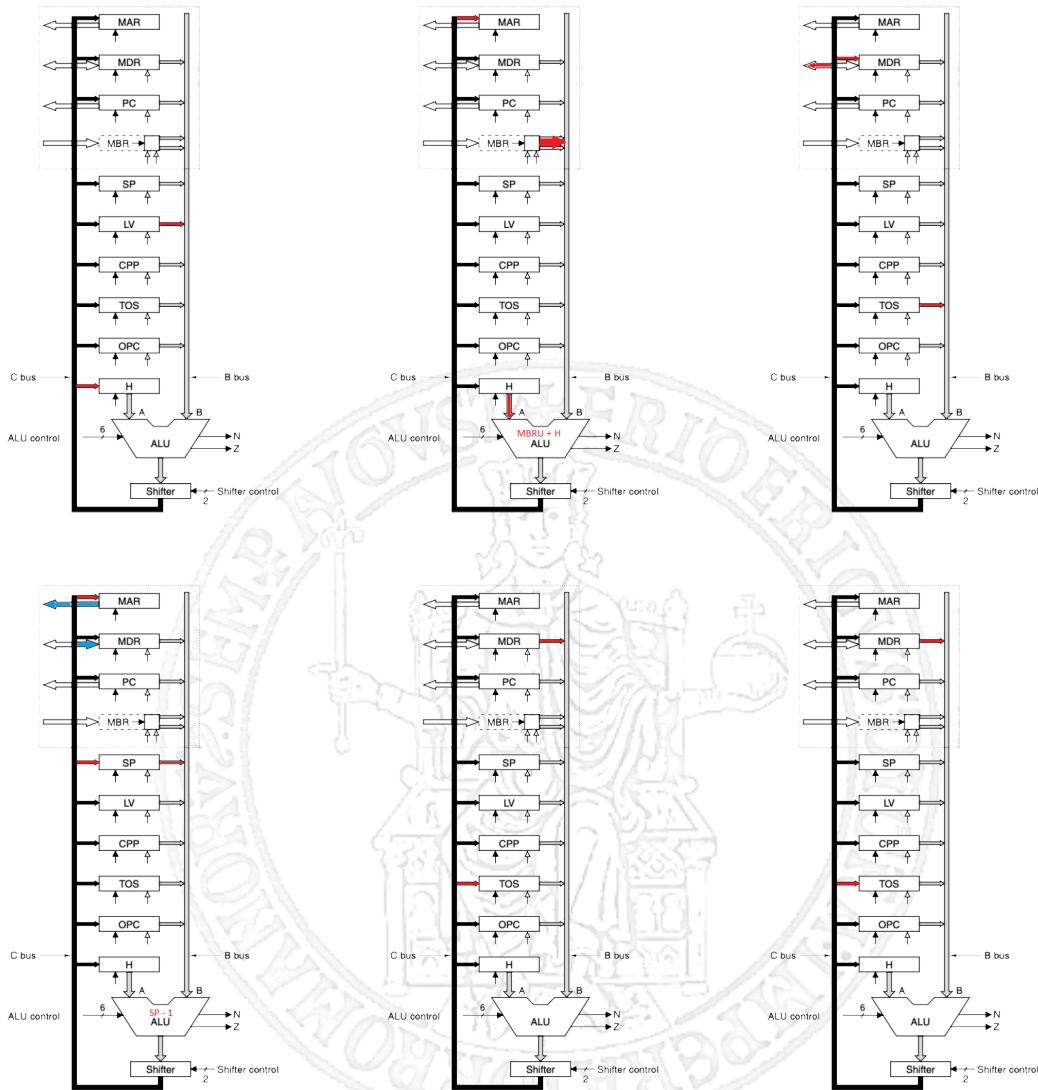


Figura 8.11: Flusso di esecuzione di ISTORE

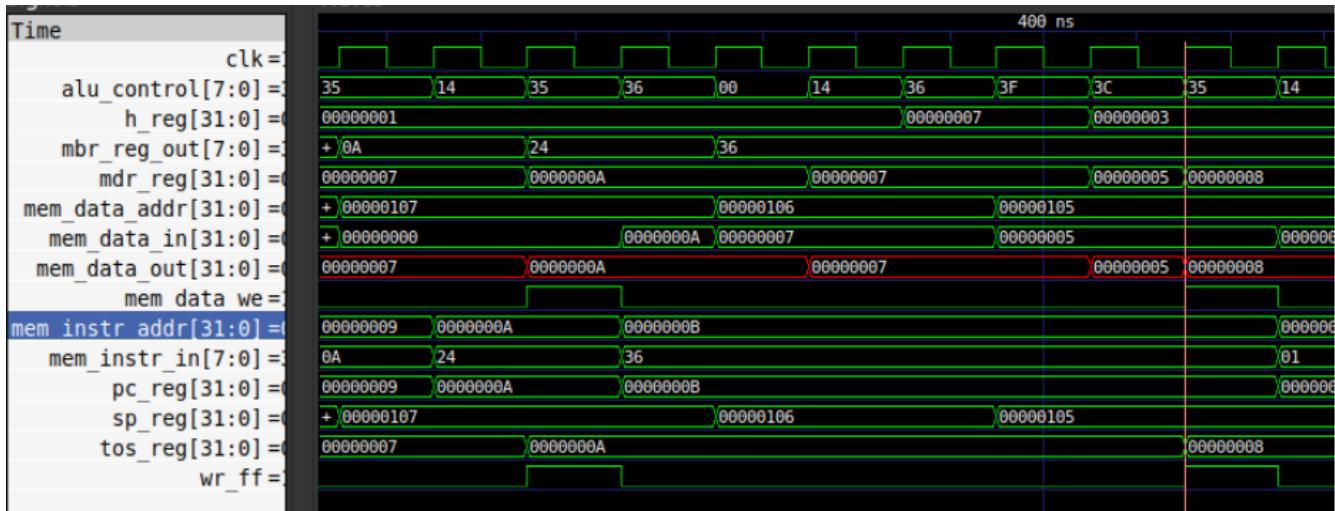


Figura 8.12: Sequenza delle microistruzioni per il codice operativo ISUB

8.6 Aggiunta di un codice operativo

Per la modifica di un codice operativo è stata creata una nuova istruzione in notazione MAL all'interno del file *ajvm.mal*: AS3 (Add Sub 3 elements). Tale istruzione esegue la pop dei primi 3 elementi in cima allo stack e effettua il push della somma del primo e del terzo sottratto col secondo estratto.

Innanzitutto bisogna individuare una locazione di memoria nella *control store* abbastanza grande da poter contenere le microistruzioni di AS3. Ispezionata tutta la control store, è stata scelta la locazione all'indirizzo 0x24. Qui di seguito è inserito il codice operativo aggiunto.

```

1 AS3 = 0x24:
2     MAR = SP = SP - 1; rd
3     empty
4     H = MDR          # b
5     MAR = SP = SP - 1; rd
6     H = TOS - H      # c - b
7     MDR = TOS = MDR + H; wr; goto main    # (c - b) + a

```

Viene effettuato la pop dei primi elementi decrementando lo stack pointer. Il registro H è utilizzato come registro accumulatore, attraverso il quale sono effettuate le operazioni di somma e di sottrazione come descritto precedentemente. Si osservi l'istruzione *empty* è un'istruzione di attesa, aggiunta poiché bisogna attendere un colpo di clock dopo aver avviato la lettura affinché nel registro MDR sia effettivamente presente il dato da leggere. Al termine dell'istruzione, il risultato viene pushato sullo stack e si ritorna al main per eseguire la successiva istruzione.

Implementato il codice operativo, vi è un ulteriore problema da risolvere. Poiché non è possibile modificare l'assemblatore IJVM, non sarà possibile tradurre il nuovo codice operativo nel suo suo corrispettivo codice binario. Per risolvere tale problema si è deciso di utilizzare una istruzione conosciuta dal compilatore, IADD, come *placeholder*; successivamente il suo codice binario salvato nella RAM è stato sostituito con quello dell'istruzione AS3, 0x24.

Ecco il programma realizzato che fa uso dell'istruzione AS3.

```
1 .main
2 .var
3 a
4 .endvar
5 BIPUSH 0x05 #A
6 BIPUSH 0x07 #B
7 BIPUSH 0x0A #C
8 AS3
9 ISTORE a
10 HALT
11 .endmethod
```

In figura 8.12 è presente la simulazione del programma. Dal segnale *mem_data_in* si nota come in MDR sono caricati i 3 valori estratti dalla cima dello stack uno dopo l'altro: 10, 7 e 5. Al termine dell'istruzione, in *mem_data_out* è presente il risultato atteso dell'operazione, ovvero $10 - 7 + 5 = 8$. È possibile inoltre osservare che, al termine, in TOS è memorizzato 8, il nuovo valore presente in cima allo stack.

8.7 Istruzioni Input/Output

È possibile individuare due metodi per eseguire operazioni di input e di output: *Memory Mapped* e *I/O Mapped*.

1) Approccio Memory Mapped

Nell'approccio *Memory Mapped*, lo spazio di indirizzamento della memoria centrale e dei registri dell'interfaccia I/O è lo stesso. Ciò consente di utilizzare istruzioni di trasferimenti dati, come per esempio una MOVE, come se si spostassero dati in memoria centrale.

Per raggiungere questo obiettivo, le periferiche effettuano *snooping* sul bus degli indirizzi della CPU e reagiscono quando riconoscono l'indirizzo a cui sono associate; tale indirizzo è memorizzato nei registri hardware interni alla periferica.

La principale problematica è che l'intero bus degli indirizzi deve essere completamente decodificato per ogni periferica, ciò rende questo approccio più lento rispetto all'*I/O Mapped*.

Il vantaggio di utilizzare questo metodo è che il programmatore ha a disposizione tutti i metodi di indirizzamento offerti per accedere alla memoria.

2) Approccio I/O Mapped

Nell'*I/O Mapped* viene assegnata uno spazio di indirizzamento separato dalla memoria, dedicato esclusivamente per l'*I/O*. Questo approccio permette di avere vantaggi in termini di velocità rispetto alla soluzione precedente anche se risulta più dispendioso in termini di area e di implementazione. In più c'è bisogno di definire delle istruzioni dedicate per le operazioni di *I/O* e ciò rende la realizzazione più complessa.

Nonostante entrambi gli approcci abbiano i loro vantaggi e svantaggi, nel processore MIC-1 è stato pensato di utilizzare l'approccio *Memory Mapped* per non dover inserire ulteriori istruzioni che dovrebbero essere dedicate per le operazioni di *I/O* e per utilizzare lo stesso spazio di indirizzamento della memoria.

Per quanto riguarda le operazioni di scrittura e lettura nei registri dell'interfaccia di *I/O*, possono essere effettuate con le solite istruzioni ISA. È necessario prevedere un'interfaccia fisica verso la periferica. Poiché tale interfaccia deve far fronte all'eterogeneità dei dispositivi *I/O*, esse risultano essere standardizzate (RS232C, USB...). È possibile quindi utilizzare un *UART* per la trasmissione seriale per l'invio e la ricezione fisica dei dati da e verso la periferica.

Data un'idea concettuale di come possano lavorare processore e periferiche, bisogna definire con quale metodologia il processore controlla lo stato della periferica.

Una prima tecnica è quella che prende il nome di *polling*: il processore effettua un'attesa attiva controllando i registri di stato della periferica periodicamente, bruciando numerosi cicli di clock essendo le periferiche normalmente molto più lente. Il vantaggio è che potrebbe essere implementato senza alcuna modifica hardware del Mic; infatti, il programma assembler può eseguire un loop infinito fin quando la periferica non comunica mediante i registri di stato la terminazione di un'operazione.

Una seconda tecnica, sicuramente molto più efficiente ma più complessa da sviluppare e da gestire, consiste nell'implementare un meccanismo di interruzione. L'interfaccia della periferica invia un segnale di *interrupt* al processore ed attende di essere servita.

Per implementare tale tecnica, nel Mic è sicuramente necessario modificare il ciclo del processore, quindi la *control store*. Infatti, alla fine di ogni istruzione ISA, piuttosto che saltare all'istruzione main bisognerebbe verificare che non vi sia un'interruzione pendente; qualora ci fosse, va servita tramite un ISR prima di tornare nel main ed eseguire l'operazione di fetch per la prossima istruzione.



Capitolo 9

Comunicazione Seriale

9.1 Traccia

Sfruttando l'implementazione fornita dalla Digilent di un dispositivo UART (componente RS232RefComp.vhd), progettare ed implementare in VHDL i seguenti componenti:

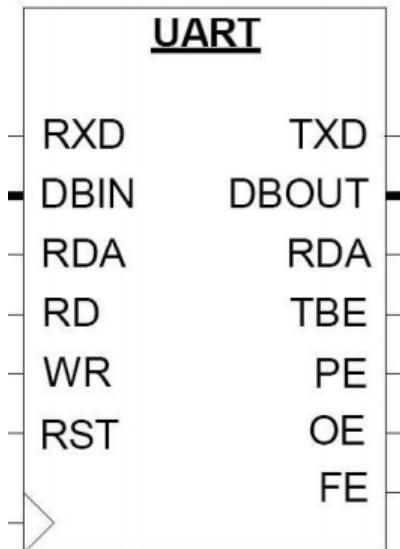
- (a) **UART_TAPPO**: il componente acquisisce una stringa di 8 bit (fornita attraverso gli switch della board di sviluppo) e la serializza tramite la sezione di trasmissione del dispositivo UART; l'output seriale della UART viene re-inviato in ingresso alla sezione di ricezione dello stesso dispositivo (configurazione a tappo), e il dato deserializzato viene visualizzato sui led della board di sviluppo.
- (b) **2_UART**: il componente acquisisce una stringa di 8 bit (fornita dall'utente tramite gli switch della board di sviluppo), la serializza tramite la sezione di trasmissione di un primo dispositivo UART, la deserializza tramite la sezione di ricezione di un secondo dispositivo UART collegato a valle del primo, e mostra le stringa led della board di sviluppo.
- (c) **UART_PC** (facoltativo): il componente realizza la comunicazione fra la board di sviluppo e un terminale seriale in esecuzione su PC (es. Termite), previa connessione di PC e board tramite dispositivo fisico RS232 (uno degli endpoint di comunicazione è rappresentato dal PC). Il componente deve poter acquisire una stringa di 8 bit che rappresenta un carattere in codifica ASCII (fornita attraverso gli switch della board di sviluppo), ed inviarla tramite il dispositivo UART al terminale in esecuzione sul PC, in cui il carattere viene visualizzato. Allo stesso modo, il componente deve essere in grado di ricevere attraverso lo stesso dispositivo UART (oppure una seconda UART) un carattere trasmesso dal terminale e mostrarlo sui led.

9.2 Interfaccia UART

Prima di poter presentare la soluzione del punto *a* e *b* della traccia, è necessario analizzare l'interfaccia del dispositivo da utilizzare, ovvero l'UART (Universal Asynchronous Receiver-Transmitter). “Lo UART è un dispositivo hardware, di uso generale o dedicato, che converte flussi di bit di dati da un formato parallelo a un formato seriale asincrono o viceversa”¹.

Senza entrare nei dettagli del protocollo di trasmissione dei dati, un dispositivo UART consente

¹ Cozzolino Giovanni, Comunicazione tra dispositivi tramite interfaccia seriale, lucidi del corso

Figura 9.1: Interfaccia del componente UART²

agevolmente una conversione parallelo/seriale -> seriale/parallelo, permettendo al sistema trasmettitore e al sistema ricevitore di disinteressarsene.

Esso può essere suddiviso in due parti distinte: *transmitter* e *receiver*. Il transmitter riceve i dati in parallelo sulla porta DBIN e li invia serialmente sulla porta TXD. Una volta caricati i dati sulla porta DBIN, per avviare la trasmissione bisogna alzare il segnale di WR. Una volta che il dato è stato inviato, ovvero il buffer di trasmissione è vuoto, si alza il bit TBE.

Il *receiver* riceve dati in serie sulla porta RXD e li fornisce in uscita in parallelo sulla porta DBOUT. Informa il sistema ricevente della ricezione di un carattere alzando il bit RDA. L'UART fornisce diversi bit per informare il sistema ricevente di eventuali errori:

- PE (parity error): l'uscita di questa porta è alta quando il bit di parità non corrisponde con la parità dei dati.
- OE (overrun error) : questo errore si verifica quando dei dati inviati vengono sovrascritti da nuovi dati senza dare il tempo al ricevitore di leggerli. Per informare il componente che la lettura è stata effettuata, bisogna dargli il segnale di RD.
- FE (framing error): errore che si verifica quando la linea non è alta quando è previsto il bit di stop.

Infine, per quanto riguarda il clock, il datasheet del componente afferma che è necessaria una frequenza di clock pari a 50MHz per il suo funzionamento.

9.3 Problematiche e soluzioni UART Tappo

Per lo sviluppo di un UART a tappo che consente di inviare dati provenienti dagli switch della board e allo stesso tempo di riceverli per trasferirli sulla porta DBOUT dello stesso dispositivo, è possibile collegare la porta di trasmissione seriale (TXD) con la porta di ricezione seriale (RXD).

²Immagine tratta dal materiale del corso

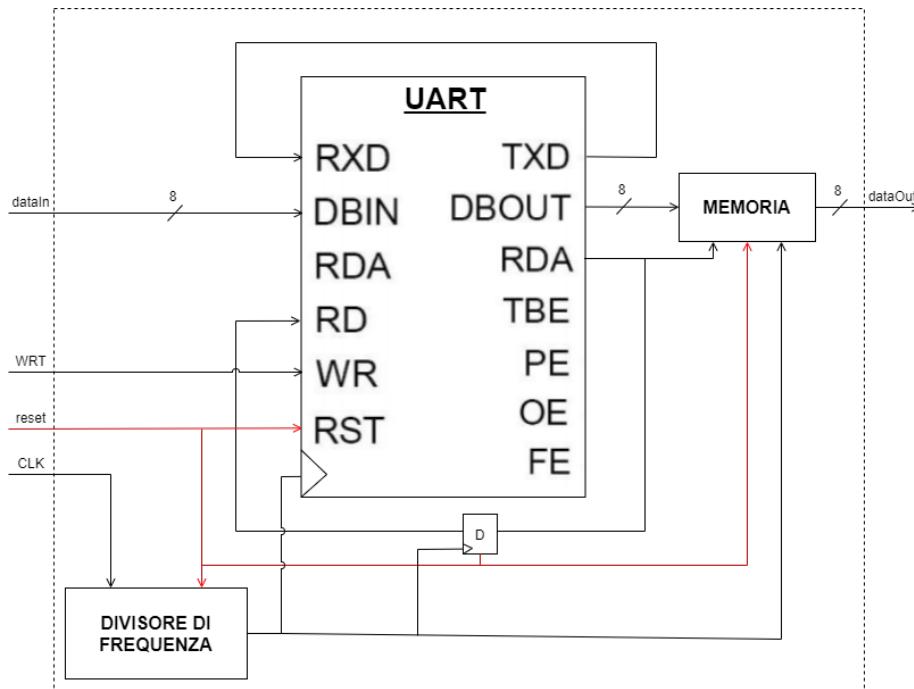


Figura 9.2: Architettura del sistema per la risoluzione del punto (a)

Da notare che ciò è possibile poiché UART prevede una comunicazione *full duplex*; infatti, vi sono due registri a scorrimento distinti: uno dedicato alla memorizzazione dei dati da trasmettere e l'altra dedicato alla ricezione; pertanto, non ci saranno problemi di collisione tra dati inviati e ricevuti.

Per rendere il progetto più semplice e snello, concentrandosi esclusivamente sul componente UART e la trasmissione seriale, è stato scelto di non utilizzare un'unità di controllo che regolasse la trasmissione, la ricezione ed il salvataggio di questi in una memoria direttamente collegata ai led della board.

Dal punto di vista del trasmettitore, non vi sono particolari accorgimenti da adottare: DBIN raccoglie i dati direttamente dagli switch della board e, tramite un segnale di WR collegato ad un bottone, la trasmissione viene avviata. Non vi è alcun controllo da fare su TBE.

Per quanto riguarda il ricevitore, sono state adottate delle soluzioni per la memorizzazione dei dati trasmessi per comunicare al UART l'avvenuta lettura dei dati. Una volta che l'intero carattere è stato ricevuto, esso viene trasmesso sulla porta di uscita DBOUT che è direttamente collegata ad una memoria che salva il dato sul fronte di discesa del clock quando l'abilitazione è alta. In assenza di un'unità di controllo, bisognava comprendere quale fosse un segnale adatto a svolgere il compito di abilitazione della memoria. Per questo scopo si adatta bene il segnale RDA che si alza una volta che il carattere è stato ricevuto ed è quindi collegato all'abilitazione della memoria.

Un'altra problematica da gestire era quella di informare il ricevitore dell'avvenuta acquisizione del carattere attraverso la porta RD. Inizialmente si era pensato di collegare direttamente il segnale di RDA con RD. Tuttavia in fase di simulazione è sorto il seguente problema: i dati su DBOUT non rimanevano abbastanza a lungo in uno stato consistente da consentire la corretta lettura. Questo poiché prima che arrivasse il fronte di discesa del clock, che consente la memorizzazione dei dati in ingresso alla memoria, RDA, il segnale di abilitazione, si abbassava. Ciò è dovuto al fatto che il componente riceve il segnale RD e, in maniera asincrona, resetta il segnale RDA. Pertanto, nel

momento in cui RDA si alzava, si abbassava quasi immediatamente.

Per evitare che questo accadesse, è stato inserito un elemento che introduce del ritardo, ovvero un Flip Flop D. In questa maniera, quando RDA si alza, sul fronte di discesa del clock avviene sia la memorizzazione del dato sia la commutazione del flip flop D. In questo momento, RD riceve il segnale di completa ricezione e abbassa RDA.

In figura 9.2 è possibile osservare lo schema della struttura appena descritta. È stato inserito un divisore di frequenza per dimezzare la frequenza di clock e portarla a 50MHz come suggerito dalla Digilent.

L'interfaccia è composta dai seguenti segnali di ingresso:

- dataIn: segnale da 8 bit da trasmettere.
- WRT: avvia la trasmissione del segnale dataIn alla porta d'uscita.
- reset: segnale per il reset dei componenti.
- clk: il segnale di clock.

9.3.1 Codice UART Tappo

Per l'implementazione dell'architettura sono stati riutilizzati il *Divisore di Frequenza*, fornito dal materiale del corso, e il UART fornito dalla Digilent. La memoria è stata implementata con un semplice processo.

```

1  entity UART_Tappo is
2      port (
3          DATA_IN    : in  std_logic_vector (7 downto 0);
4          WRT        : in  std_logic;
5          RST        : in  std_logic;
6          CLK        : in  std_logic;
7          DATA_OUT   : out std_logic_vector (7 downto 0)
8      );
9
10
11 end UART_Tappo;
12
13 architecture Structural of UART_Tappo is
14
15 component UARTcomponent is
16     Generic (
17         --@48MHz
18         BAUD_DIVIDE_G : integer := 1;    --115200 baud
19         BAUD_RATE_G   : integer := 16
20
21         --@26.6MHz
22         --BAUD_DIVIDE_G : integer := 14;  --115200 baud
23         --BAUD_RATE_G   : integer := 231
24     );
25

```

```

26      port (
27          TXD    : out    std_logic    := '1';           -- Transmitted serial
28          data output
29          RXD    : in     std_logic;                  -- Received serial data
30          input
31          CLK    : in     std_logic;                  -- Clock signal
32          DBIN   : in     std_logic_vector(7 downto 0); -- Input parallel
33          data to be transmitted
34          DBOUT  : out    std_logic_vector(7 downto 0); -- Recevived
35          parallel data output
36          RDA    : inout   std_logic;                 -- Read Data Available
37          TBE    : out    std_logic := '1';           -- Transfer Buffer Emty
38          RD     : in     std_logic := '0';           -- Read Strobe
39          WR     : in     std_logic;                 -- Write Strobe
40          PE     : out    std_logic;                 -- Parity error
41          FE     : out    std_logic;                 -- Frame error
42          OE     : out    std_logic;                 -- Overwrite error
43          RST    : in     std_logic := '0';           -- Reset signal
44      );
45
46 end component;
47
48 component clock_filter is
49     generic(
50         clock_frequency_in : integer := 50000000;
51         clock_frequency_out : integer := 5000000
52     );
53     Port (
54         clock_in : in STD_LOGIC;
55         reset_n : in STD_LOGIC;
56         clock_out : out STD_LOGIC
57     );
58
59 end component;
60
61 signal memoria : std_logic_vector(7 downto 0) := (others => '0');
62 signal s_mem   : std_logic_vector(7 downto 0) := (others => '0');
63 signal s_rxd_txd : std_logic := '1';
64 signal s_clk   : std_logic := '0';
65 signal s_rda   : std_logic := '0';
66 signal s_ff    : std_logic := '0';
67
68 begin
69
70     UART : UARTcomponent port map (
71         TXD    => s_rxd_txd,
72         RXD    => s_rxd_txd,
73         CLK    => s_clk,
74         DBIN   => DATA_IN,

```

```

71      DBOUT => s_mem,
72      RDA => s_rda,
73      RD    => s_ff,
74      WR    => WRT,
75      RST   => RST
76  );
77
78 DIV : clock_filter generic map (
79     clock_frequency_in => 100000000,
80     clock_frequency_out => 50000000
81 )
82
83     port map(
84         clock_in  => clk,
85         reset_n   => rst,
86         clock_out  => s_clk
87 );
88
89 p : process(clk, rst)
90 begin
91
92     if(rst = '1') then
93         memoria <= (others => '0');
94
95
96     elsif(clk'event and clk = '0' and s_rda = '1') then
97         memoria <= s_mem;
98     end if;
99
100
101
102 end process;
103 DATA_OUT <= memoria;
104
105 ff : process(clk, rst)
106 begin
107
108     if(rst = '1') then
109         s_ff <= '0';
110     elsif(clk'event and clk = '0') then
111         s_ff <= s_rda;
112     end if;
113
114
115 end process;
116 end Structural;

```

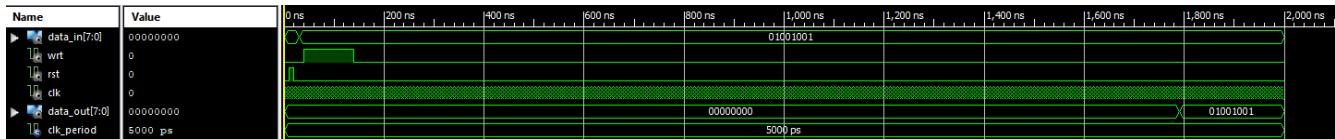


Figura 9.3: Simulazione del sistema UART tappo

9.3.2 Simulazione UART Tappo

Per la simulazione è stato generato il testbench qui riportato.

La macchina viene resettata alzando il segnale RST. Dopodiché, in input è stata inserita la stringa 01001001 ed è stata avviata la trasmissione con il segnale WRT.

È possibile osservare come tale segnale si replica su DATA_OUT e ciò verifica la corretta trasmissione del dato.

```

1 ff : process(clk, rst)
2 begin
3
4     if(rst = '1') then
5         s_ff <= '0';
6     elsif(clk'event and clk = '0') then
7         s_ff <= s_rda;
8     end if;
9
10    end process;
11 end Structural;
```



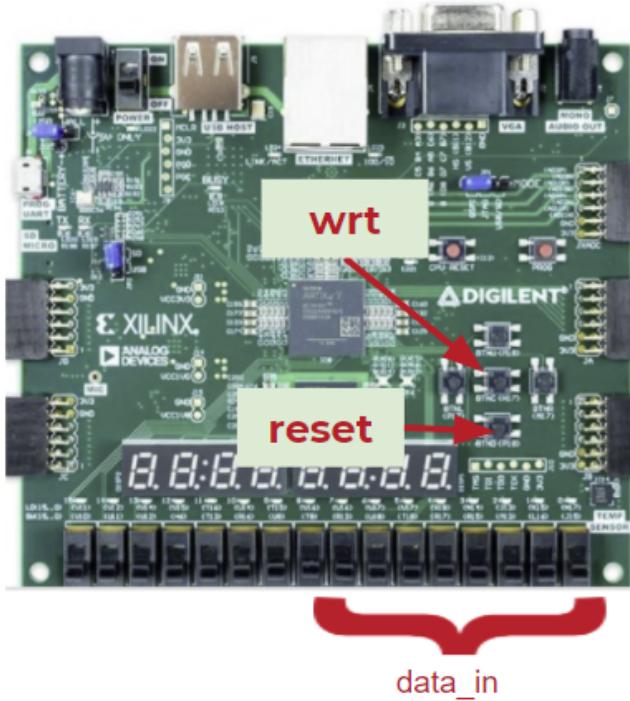


Figura 9.4: Mapping dei segnali sui dispositivo I/O della board

9.3.3 Mapping sulla scheda

Gli input del sistema sono stati mappati sulla Nexys 4DDR100T nel seguente modo:

- dataIn: l'ingresso è mappato sugli ultimi 8 switch.
- WRT: il bottone centrale è stato predisposto per l'avvio della trasmissione.
- reset: il reset è stato mappato sul bottone di sotto.
- dataOut: l'uscita è visualizzata sui led.

In figura 9.6 vi è una rappresentazione grafica del mapping.

```

1 NET "CLK"    LOC = "E3"    | IOSTANDARD = "LVCMOS33";           #Bank = 35, Pin
2      name = #IO_L12P_T1_MRCC_35,                                Sch name = clk100mhz
3 ## Switches
4 NET "DATA_IN<0>"          LOC=J15  | IOSTANDARD=LVCMS33; #IO_L24N_T3_RS0_15
5 NET "DATA_IN<1>"          LOC=L16  | IOSTANDARD=LVCMS33; #
6     IO_L3N_T0_DQS_EMCCCLK_14
7 NET "DATA_IN<2>"          LOC=M13  | IOSTANDARD=LVCMS33; #
8     IO_L6N_T0_D08_VREF_14
9 NET "DATA_IN<3>"          LOC=R15  | IOSTANDARD=LVCMS33; #IO_L13N_T2_MRCC_14
10 NET "DATA_IN<4>"          LOC=R17  | IOSTANDARD=LVCMS33; #IO_L12N_T1_MRCC_14
11 NET "DATA_IN<5>"          LOC=T18  | IOSTANDARD=LVCMS33; #IO_L7N_T1_D10_14
12 NET "DATA_IN<6>"          LOC=U18  | IOSTANDARD=LVCMS33; #
13     IO_L17N_T2_A13_D29_14
14 NET "DATA_IN<7>"          LOC=R13  | IOSTANDARD=LVCMS33; #IO_L5N_T0_D07_14

```

```

11 ## Buttons
12 NET "WRT"          LOC=N17 | IOSTANDARD=LVC MOS33; #IO_L9P_T1_DQS_14
13 NET "RST"          LOC=P18 | IOSTANDARD=LVC MOS33; #IO_L9N_T1_DQS_D13_14
14 ## LEDs
15 NET "DATA_OUT<0>"    LOC=H17 | IOSTANDARD=LVC MOS33; #IO_L18P_T2_A24_15
16 NET "DATA_OUT<1>"    LOC=K15 | IOSTANDARD=LVC MOS33; #IO_L24P_T3_RS1_15
17 NET "DATA_OUT<2>"    LOC=J13 | IOSTANDARD=LVC MOS33; #IO_L17N_T2_A25_15
18 NET "DATA_OUT<3>"    LOC=N14 | IOSTANDARD=LVC MOS33; #IO_L8P_T1_D11_14
19 NET "DATA_OUT<4>"    LOC=R18 | IOSTANDARD=LVC MOS33; #IO_L7P_T1_D09_14
20 NET "DATA_OUT<5>"    LOC=V17 | IOSTANDARD=LVC MOS33; #
   IO_L18N_T2_A11_D27_14
21 NET "DATA_OUT<6>"    LOC=U17 | IOSTANDARD=LVC MOS33; #
   IO_L17P_T2_A14_D30_14
22 NET "DATA_OUT<7>"    LOC=U16 | IOSTANDARD=LVC MOS33; #
   IO_L18P_T2_A12_D28_14

```

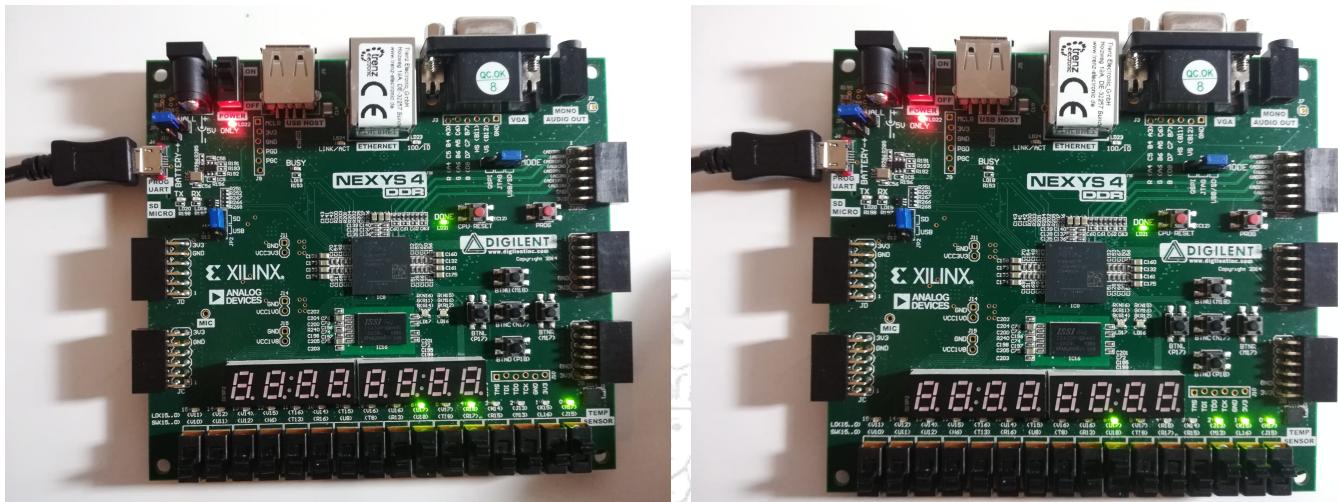


Figura 9.5: Foto dell'implementazione del progetto sulla board



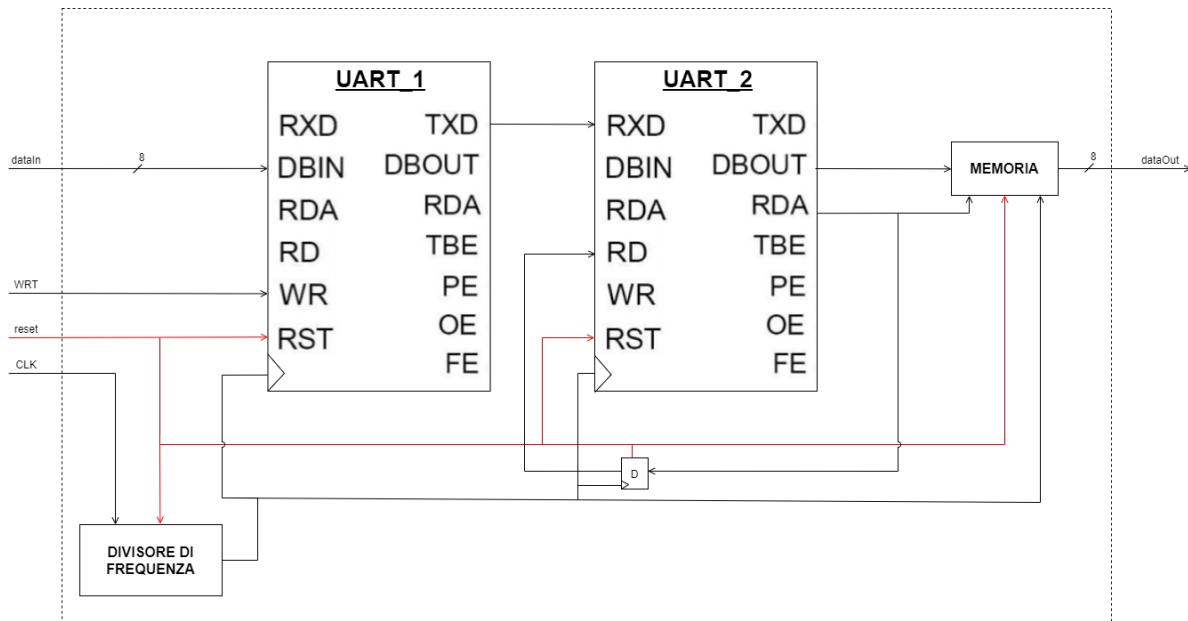


Figura 9.6: Architettura del sistema per la risoluzione del punto (b)

9.4 Problematiche e soluzioni 2 UART

L'architettura del sistema per la risoluzione del punto *b* è molto simile a quella per il punto *a*. Ciò è dovuto alla separazione dei compiti che ogni UART deve svolgere. In altri termini, se nel punto *a* *receiver* e *transmitter* degenerano in un unico UART a tappo, il punto *b* è una semplice scissione dei due compiti.

Questa può essere la tipica configurazione di due sistemi che comunicano tramite un'interfaccia seriale. Non è necessario che i clock dei due dispositivi abbiano la stessa frequenza, però è comunque suggerito dal datasheet della Digilent di fornire al dispositivo UART una frequenza pari a 50MHz. Fatta questa premessa, in questa soluzione è stata utilizzata una base dei tempi comune.

Da notare che in questa configurazione soltanto UART1 può trasmettere i dati al UART2 poiché non vi era alcun motivo per implementare una comunicazione full-duplex. Se invece si desiderava che la comunicazione potesse essere realizzata in ambo i versi, era necessario collegare l'uscita TXD del UART2 con l'entrata RXD del UART1. Anche qui persiste l'idea di evitare l'utilizzo di un'unità di controllo per semplificare il progetto e concentrarsi solamente sulla trasmissione; pertanto, vengono adottati tutti gli accorgimenti visti nei paragrafi precedenti.

Per quanto riguarda la simulazione del dispositivo e la sua implementazione sulla board, non cambia nulla rispetto a quanto visto nei paragrafi 9.3.2 e 9.3.3, poiché è stato deciso di lasciare l'interfaccia invariata.

9.4.1 Codice 2 UART

```

1  entity UART_2 is
2      port(
3          DATA_IN : IN std_logic_vector(7 downto 0);
4              WRT : IN std_logic;
5              RST : IN std_logic;
6              CLK : IN std_logic;
7              DATA_OUT : OUT std_logic_vector(7 downto 0)
8      );
9
10 end UART_2;
11
12 architecture Structural of UART_2 is
13
14 component UARTcomponent is
15     Generic (
16         --@48MHz
17         BAUD_DIVIDE_G : integer := 1;      --115200 baud
18         BAUD_RATE_G   : integer := 16
19
20         --@26.6MHz
21         --BAUD_DIVIDE_G : integer := 14;    --115200 baud
22         --BAUD_RATE_G   : integer := 231
23     );
24
25     port(
26         TXD      : out std_logic := '1';           -- Transmitted serial
27             data output
28         RXD      : in  std_logic := '1';           -- Received serial
29             data input
30         CLK      : in  std_logic;                  -- Clock signal
31         DBIN     : in  std_logic_vector(7 downto 0) := (others => '0');
32             -- Input parallel data to be transmitted
33         DBOUT    : out std_logic_vector(7 downto 0); -- Received
34             parallel data output
35         RDA      : inout std_logic;                -- Read Data Available
36         TBE      : out std_logic := '1';           -- Transfer Buffer Emty
37         RD       : in  std_logic := '0';           -- Read Strobe
38         WR       : in  std_logic := '0';           -- Write Strobe
39         PE       : out std_logic;                 -- Parity error
40         FE       : out std_logic;                 -- Frame error
41         OE       : out std_logic;                 -- Overwrite error
42         RST     : in  std_logic := '0';           -- Reset signal
43     );
44
45 end component;
46
47 component clock_filter is

```

```

44 generic(
45     clock_frequency_in : integer := 50000000;
46     clock_frequency_out : integer := 5000000
47 );
48 Port (
49     clock_in    : in STD_LOGIC;
50     reset_n     : in STD_LOGIC;
51     clock_out   : out STD_LOGIC
52 );
53
54 end component;
55
56
57 signal memoria : std_logic_vector(7 downto 0) := (others => '0');
58 signal s_mem    : std_logic_vector(7 downto 0) := (others => '0');
59 signal s_txd_rxd : std_logic := '1';
60 signal s_ff      : std_logic := '0';
61 signal s_rda     : std_logic := '0';
62 signal s_clk     : std_logic := '0';
63
64 begin
65
66 UART1 : UARTcomponent port map (
67     TXD    => s_txd_rxd,
68     CLK    => s_clk,
69     DBIN   => DATA_IN,
70     WR     => WRT,
71     RST   => RST
72 );
73
74 UART : UARTcomponent port map (
75     RXD    => s_txd_rxd,
76     CLK    => s_clk,
77     DBOUT  => s_mem,
78     RDA    => s_rda,
79     RD     => s_ff,
80     RST   => RST
81 );
82
83
84 DIV : clock_filter generic map (
85     clock_frequency_in => 100000000,
86     clock_frequency_out => 50000000
87 )
88
89     port map(
90         clock_in  => clk,
91         reset_n   => rst,
92         clock_out  => s_clk

```

```
93 );
94
95
96
97 p : process(clk, rst)
98 begin
99
100    if(rst = '1') then
101       memoria <= (others => '0');
102
103
104    elsif(clk'event and clk = '0' and s_rda = '1') then
105       memoria <= s_mem;
106    end if;
107
108
109
110   end process;
111 DATA_OUT <= memoria;
112
113 ff : process(clk, rst)
114 begin
115
116    if(rst = '1') then
117       s_ff <= '0';
118    elsif(clk'event and clk = '0') then
119       s_ff <= s_rda;
120    end if;
121
122
123
124
125 end Structural;
```



Capitolo 10

Reti di Interconnessione

10.1 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).
- b) (Opzionale) rimuovendo l'ipotesi di lavorare secondo uno schema a priorità fra i nodi e considerando una rete di 8 nodi, lo switch deve gestire eventuali conflitti generati da collisioni secondo un meccanismo a scelta (ad es. perdendo uno dei messaggi in conflitto).
- c) (Opzionale) Si implementi un protocollo di handshaking semplice regolato da una coppia di segnali (pronto a inviare/pronto a ricevere) per l'invio di ciascun messaggio fra due nodi.

10.2 Introduzione

Per implementare la rete di interconnessione è stata sfruttata la tecnica del *Perfect Shuffling*. “Il Perfect Shuffling è una tecnica che fa riferimento al modo con cui le carte di un mazzo vengono mischiate, infatti, se si divide il mazzo di carte in due parti uguali e si mischiano ‘perfettamente’, dopo $\log_2(n)$ volte si ritorna all’ordine iniziale, questa tecnica ci permette di ottimizzare i percorsi da seguire indicando i nodi di collegamento tra i diversi stadi.”

Utilizzando switch con 2 ingressi e 2 uscite, si può realizzare una rete composta da n stadi, dove $n = \log_2(\#nodi)$.

Il vantaggio del Perfect Shuffling è quello di realizzare uno switch multistadio tramite semplici componenti. Il sistema risulta facilmente scalabile poiché è possibile realizzare uno switch di dimensioni maggiori, senza modificare l’architettura, incrementando soltanto il numero di stadi e il numero di componenti.

La logica di controllo può essere distribuita tra i nodi della rete oppure realizzata tramite una componente che regola la politica di instradamento dei messaggi.

La problematica principale riguarda la gestione delle collisioni; esse possono essere causate da comunicazioni simultanee. Per far fronte a questo problema, in letteratura sono state proposte

varie tipologie di soluzioni.

Nelle architetture sviluppate in questo capitolo sono state utilizzate le seguenti strategie per gestire le collisioni:

- Esercizio 1: come richiesto dalla traccia è stata definita una rete a priorità, dove non possono esserci collisioni poiché vi è un controllo centralizzato in cui viene stabilito il singolo percorso da abilitare.
- Esercizio 2: non è stato utilizzato l'approccio con una rete a priorità e si è scelto di implementare una rete di controllo distribuita nei singoli nodi; in caso di collisione viene data la precedenza all'ingresso *in0*.
- Esercizio 3: è stata utilizzata la tecnica di *store & forward*, dove è stato implementato un protocollo di handshaking per la comunicazione dei nodi.



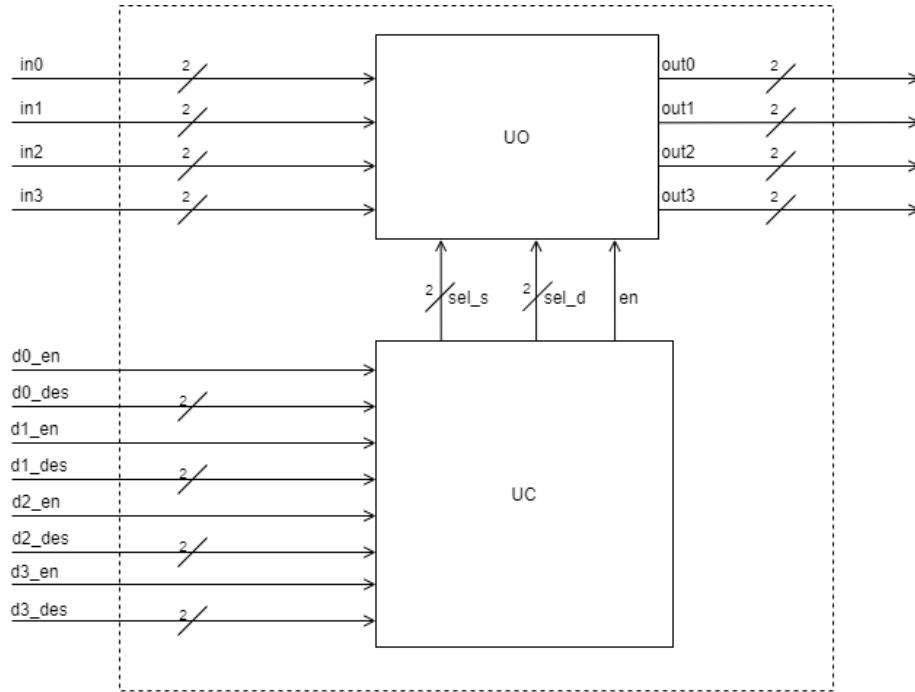


Figura 10.1: Suddivisione dello switch multistadio in unità operativa e unità di controllo

10.3 Rete di Interconnessione a Priorità

Come richiesto dal punto (a) della traccia, è stata implementata una rete a priorità ed è stato imposto il vincolo di non avere comunicazioni simultanee.

I nodi vengono serviti secondo uno schema a priorità determinata, dove il *nodo0* ha priorità massima e il *nodo3* ha priorità minima. In questo modo non vi è rischio che ci siano collisioni e dunque perdite di messaggi, con lo svantaggio di avere minore efficienza.

Inoltre, i dispositivi a priorità minore ricadono nel fenomeno di *starvation*, ossia possono non essere mai serviti se nodi a priorità maggiore monopolizzano la rete.

L'immagine in figura 10.1 mostra le componenti di cui è composto il dispositivo. L'unità operativa svolge il ruolo di uno switch 4:4: prende il dato da una delle linee *in* e lo inoltra in una linea *out*. L'unità di controllo è il componente che calcola la sorgente e la destinazione, secondo il meccanismo di priorità che è stato descritto precedentemente.

Alzando il bit *d*_en*, il dispositivo comunica l'intenzione di utilizzare la rete all'unità di controllo che, a seconda delle richieste, valuta a quale dispositivo spetti l'utilizzo. Inoltre i dispositivi comunicano la destinazione del messaggio tramite il segnale *d*_des*.

10.3.1 Switch 2:2

Il componente base per la rete di interconnessione dell'unità operativa è lo *switch 2:2*, rappresentato in figura 10.2. Il dispositivo non è stato progettato in logica tristate quindi, se non vi sono ingressi, l'uscita è bassa. Sono stati implementati un multiplexer e un demultiplexer che hanno un segnale di *enable* comune e un segnale di selezione per indicare rispettivamente la sorgente e la destinazione del dato.

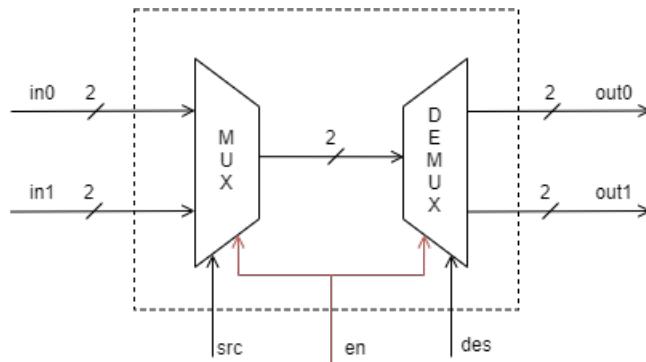


Figura 10.2: Architettura interna di uno switch 2:2

10.3.2 Architettura completa: Switch Multistadio

Lo schema in figura 10.3 rappresenta l’architettura completa, dove sono stati collegati opportunamente i componenti descritti in precedenza. I segnali di ingresso e di uscita dal sistema compongono l’interfaccia dello switch multistadio.

Come è possibile notare, non vi è alcun segnale di temporizzazione esterno: la rete è puramente combinatoria.

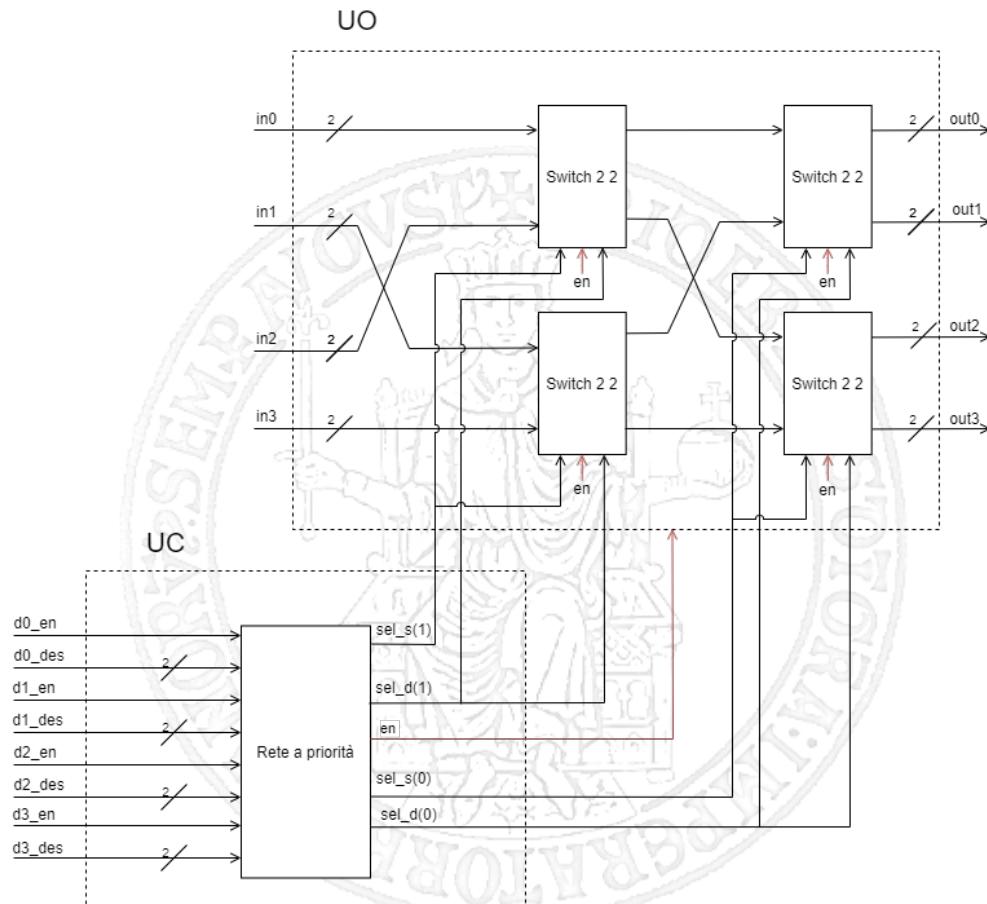


Figura 10.3: Architettura interna dello switch multistadio progettato

10.3.3 Codice della rete di interconnessione a priorità

Switch 2:2

Lo *Switch 2:2* è stato realizzato con un approccio strutturale, collegando opportunamente le componenti come mostrato in figura 10.2.

```

1  entity switch2_2 is
2      port(
3          in0: in std_logic_vector(1 downto 0);
4          in1: in std_logic_vector(1 downto 0);
5          sel_d: in std_logic;
6          sel_s: in std_logic;
7          en: in std_logic;
8          out0: out std_logic_vector(1 downto 0);
9          out1: out std_logic_vector(1 downto 0)
10     );
11
12 end switch2_2;
13
14 architecture Structural of switch2_2 is
15
16     component multiplexer
17         port(
18             in0: in std_logic_vector(1 downto 0);
19             in1: in std_logic_vector(1 downto 0);
20             en: in std_logic;
21             s: in std_logic;
22             out0: out std_logic_vector(1 downto 0)
23         );
24     end component;
25
26     component demultiplexer
27         port(
28             in0: in std_logic_vector(1 downto 0);
29             en: in std_logic;
30             d: in std_logic;
31             out0: out std_logic_vector(1 downto 0);
32             out1: out std_logic_vector(1 downto 0)
33         );
34     end component;
35
36     signal out_int: std_logic_vector(1 downto 0) :=(others=>'0');
37 begin
38     mux: multiplexer port map(
39         in0 => in0,
40         in1 => in1,
41         en => en,
42         s => sel_s,
43         out0 => out_int

```

```

44 );
45
46 demux: demultiplexer port map(
47   in0 => out_int,
48   en => en,
49   d => sel_d,
50   out0 => out0,
51   out1 => out1
52 );
53 end Structural;

```

Unità operativa

L'unità operativa è stata realizzata con un approccio strutturale. Sono stati utilizzati quattro *switch 2:2* e sono stati connessi secondo lo schema in figura 10.3. Era possibile istanziare gli *switch 2:2* con il costrutto *generate*; tuttavia, essendo pochi, si è preferito istanziarli uno alla volta.

```

1 entity uo is
2   port (
3     in0: in std_logic_vector(1 downto 0);
4     in1: in std_logic_vector(1 downto 0);
5     in2: in std_logic_vector(1 downto 0);
6     in3: in std_logic_vector(1 downto 0);
7
8     sel_s: in std_logic_vector(1 downto 0);
9     sel_d: in std_logic_vector(1 downto 0);
10
11    en: in std_logic;
12    out0: out std_logic_vector(1 downto 0);
13    out1: out std_logic_vector(1 downto 0);
14    out2: out std_logic_vector(1 downto 0);
15    out3: out std_logic_vector(1 downto 0)
16  );
17
18 end uo;
19
20 architecture Structural of uo is
21   component switch2_2 is
22     port (
23       in0: in std_logic_vector(1 downto 0);
24       in1: in std_logic_vector(1 downto 0);
25       sel_s: in std_logic;
26       sel_d: in std_logic;
27       en: in std_logic;
28       out0: out std_logic_vector(1 downto 0);
29       out1: out std_logic_vector(1 downto 0)
30     );
31   end component;

```

```

32
33
34 type data_2 is array (0 to 3) of std_logic_vector(1 downto 0);
35 signal s_switch_1: data_2;
36
37
38 begin
39     switch1_1: switch2_2 --LIVELLO 1 - SWITCH 1
40     port map(
41         in0 =>in0,
42         in1=> in2,
43         sel_s=> sel_s(1),
44         sel_d=> sel_d(1),
45         en=>en,
46         out0=>s_switch_1(0),
47         out1=>s_switch_1(1)
48     );
49     switch1_2: switch2_2 --LIVELLO 1 - SWITCH 2
50     port map(
51         in0 =>in1,
52         in1=>in3,
53         sel_s=>sel_s(1),
54         sel_d=>sel_d(1),
55         en=>en,
56         out0=>s_switch_1(2),
57         out1=>s_switch_1(3)
58     );
59
60     switch2_1: switch2_2 --LIVELLO 2 - SWITCH 1
61     port map(
62         in0 =>s_switch_1(0),
63         in1=>s_switch_1(2),
64         sel_s=>sel_s(0),
65         sel_d=>sel_d(0),
66         en=>en,
67         out0=>out0,
68         out1=>out1
69     );
70
71     switch2_21: switch2_2 --LIVELLO 2 - SWITCH 2
72     port map(
73         in0 =>s_switch_1(1),
74         in1=>s_switch_1(3),
75         sel_s=>sel_s(0),
76         sel_d=>sel_d(0),
77         en=>en,
78         out0=>out2,
79         out1=>out3
80     );

```

```

81
82
83 end Structural;

```

Unità di controllo

L'unità di controllo è una rete puramente combinatoria ed è stata implementata tramite l'approccio dataflow.

```

1  entity uc is
2    port (
3      d0_en: in std_logic;
4      d1_en: in std_logic;
5      d2_en: in std_logic;
6      d3_en: in std_logic;
7
8      d0_des: in std_logic_vector(1 downto 0);
9      d1_des: in std_logic_vector(1 downto 0);
10     d2_des: in std_logic_vector(1 downto 0);
11     d3_des: in std_logic_vector(1 downto 0);
12
13     en: out std_logic;
14     d_out: out std_logic_vector(1 downto 0);
15     s_out: out std_logic_vector(1 downto 0)
16   );
17
18 end uc;
19
20 architecture Dataflow of uc is
21
22 begin
23   d_out <= d0_des when d0_en='1' else
24     d1_des when d1_en='1' else
25     d2_des when d2_en='1' else
26     d3_des when d3_en='1' else
27     "00";
28
29   s_out <= "00" when d0_en='1' else
30     "01" when d1_en='1' else
31     "10" when d2_en='1' else
32     "11" when d3_en='1' else
33     "00";
34
35   en <= d0_en OR d1_en OR d2_en OR d3_en;
36
37 end Dataflow;

```

Switch Multistadio

Lo switch multistadio è stato realizzato con un approccio strutturale, collegando opportunamente unità di controllo e unità operativa.

```

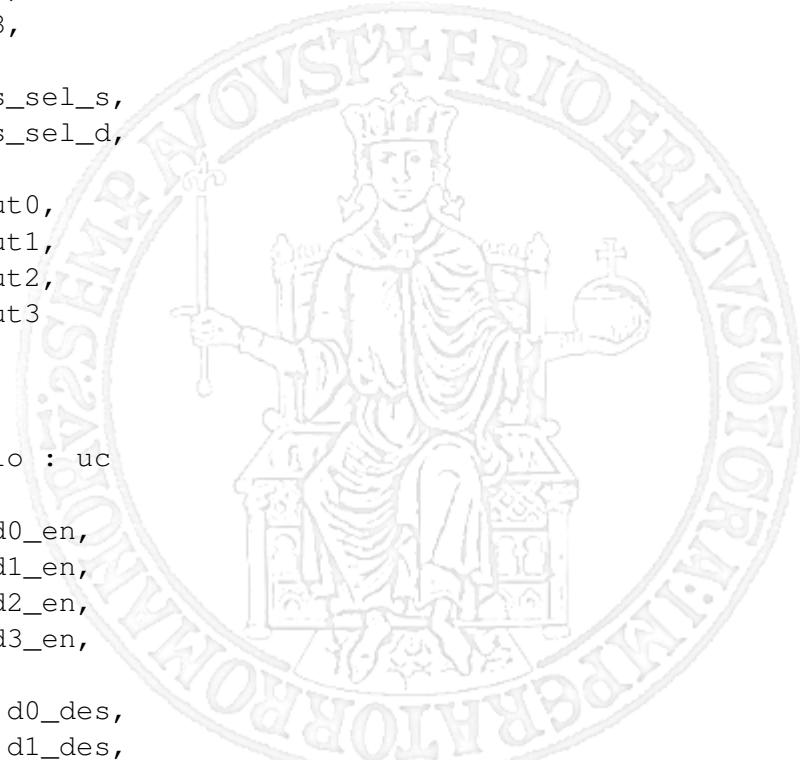
1  entity switch_multistadio is
2      port(
3          in0: in std_logic_vector(1 downto 0);
4          in1: in std_logic_vector(1 downto 0);
5          in2: in std_logic_vector(1 downto 0);
6          in3: in std_logic_vector(1 downto 0);
7
8          d0_en: in std_logic;
9          d1_en: in std_logic;
10         d2_en: in std_logic;
11         d3_en: in std_logic;
12
13         d0_des: in std_logic_vector(1 downto 0);
14         d1_des: in std_logic_vector(1 downto 0);
15         d2_des: in std_logic_vector(1 downto 0);
16         d3_des: in std_logic_vector(1 downto 0);
17
18         out0: out std_logic_vector(1 downto 0);
19         out1: out std_logic_vector(1 downto 0);
20         out2: out std_logic_vector(1 downto 0);
21         out3: out std_logic_vector(1 downto 0)
22     );
23 end switch_multistadio;
24
25 architecture Structural of switch_multistadio is
26
27     component uc is
28         port(
29             d0_en: in std_logic;
30             d1_en: in std_logic;
31             d2_en: in std_logic;
32             d3_en: in std_logic;
33
34             d0_des: in std_logic_vector(1 downto 0);
35             d1_des: in std_logic_vector(1 downto 0);
36             d2_des: in std_logic_vector(1 downto 0);
37             d3_des: in std_logic_vector(1 downto 0);
38             en: out std_logic;
39             d_out: out std_logic_vector(1 downto 0);
40             s_out: out std_logic_vector(1 downto 0)
41         );
42     end component;
43
44     component uo is
45         port(

```

```

46      in0: in std_logic_vector(1 downto 0);
47      in1: in std_logic_vector(1 downto 0);
48      in2: in std_logic_vector(1 downto 0);
49      in3: in std_logic_vector(1 downto 0);
50
51      sel_s: in std_logic_vector(1 downto 0);
52      sel_d: in std_logic_vector(1 downto 0);
53
54      en: in std_logic;
55      out0: out std_logic_vector(1 downto 0);
56      out1: out std_logic_vector(1 downto 0);
57      out2: out std_logic_vector(1 downto 0);
58      out3: out std_logic_vector(1 downto 0)
59  );
60 end component;
61
62 signal s_sel_s: std_logic_vector(1 downto 0) := (others => '0');
63 signal s_sel_d: std_logic_vector(1 downto 0) := (others => '0');
64 signal s_en: std_logic := '0';
65 begin
66     unita_operativa: uo
67     port map(
68         in0 => in0,
69         in1 => in1,
70         in2 => in2,
71         in3 => in3,
72
73         sel_s => s_sel_s,
74         sel_d => s_sel_d,
75         en=>s_en,
76         out0 => out0,
77         out1 => out1,
78         out2 => out2,
79         out3 => out3
80     );
81
82     unita_controllo : uc
83     port map(
84         d0_en => d0_en,
85         d1_en => d1_en,
86         d2_en => d2_en,
87         d3_en => d3_en,
88
89         d0_des => d0_des,
90         d1_des => d1_des,
91         d2_des => d2_des,
92         d3_des => d3_des,
93         en=>s_en,
94

```



```
95      d_out => s_sel_d,  
96      s_out => s_sel_s  
97  );  
98 end Structural;
```



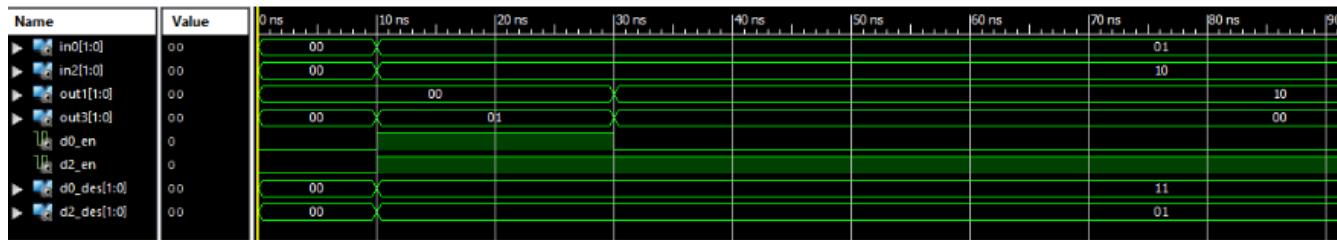


Figura 10.4: Simulazione della Rete di Interconnessione a Priorità

10.3.4 Simulazione

La simulazione rappresentata in figura 10.4 riporta solo i segnali coinvolti in modo da rendere più semplice la visualizzazione.

È stato utilizzato il testbench sottostante e, poiché è semplice valutare il comportamento della macchina, non sono stati definiti degli assert. Il nodo 0 vuole comunicare con il nodo 3 e il nodo 2 con il nodo 1. Entrambi i nodi alzano il relativo segnale di abilitazione.

Il primo dispositivo ad utilizzare la rete è 0 poiché ha priorità più alta. Infatti, tra 10ns e 30ns, out3 è pari a 01 mentre out1 è nullo. Quando il nodo 0 abbassa l'abilitazione, out3 si abbassa mentre out1 contiene il messaggio inviato dal nodo 2.

```

1  stim_proc: process
2      begin
3          -- hold reset state for 100 ns.
4          wait for 10 ns;
5
6          d0_en<='1';
7          d2_en<='1';
8
9          d0_des<="11"; --0->3
10         d2_des<="01"; --2->1
11
12        in0<="01";
13        in2<="10";
14
15        wait for 20 ns;
16        d0_en<='0';
17
18        wait for 20 ns;
19        d2_en<='0';

```

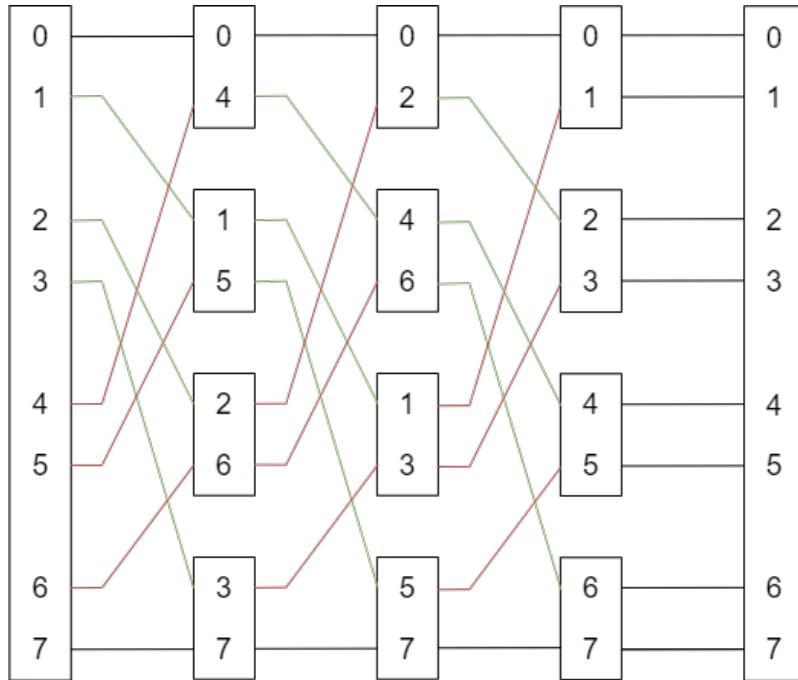


Figura 10.5: Rete multistadio a 3 livelli

10.4 Rete di Interconnessione con controllo distribuito

Il punto (b) della traccia non prevede di lavorare secondo uno schema a priorità, pertanto è stata rimossa l'unità di controllo che implementava la rete a priorità e pilotava gli switch a seconda di una determinata logica.

Da qui vi è la necessità di avere una logica di controllo distribuita tra i vari nodi degli n-stadi che costituiscono la rete. È possibile apprezzare ciò dallo schema in figura 10.5 che rappresenta l'intero sistema di interconnessione, realizzato solamente tramite degli switch 2:2. Quest'ultimi sono diversi da quelli progettati nell'architettura precedente, poiché essi devono instradare il messaggio in base alla destinazione contenuta nel messaggio stesso, compito che prima spettava ad un'unica unità di controllo esterna.

È evidente che non sono solo i due bit dato che devono viaggiare attraverso i nodi, ma anche ulteriori informazioni affinché gli switch possano calcolare l'uscita appropriata.

La struttura del messaggio è riportata in figura 10.6. Il primo bit è il bit di *idle*: se è basso indica che il collegamento non è attivo, ovvero non vi è alcun messaggio in transito su quella linea. I successivi tre bit, ovvero $\lceil \log_2(\#dispositivi) \rceil$ bit, indicano la destinazione del messaggio. Come per il progetto discusso per il punto (a), un generico livello i deve valutare l' i -esimo bit più significativo. Se il bit è 0, allora il messaggio deve essere incanalato verso la porta di uscita in alto, altrimenti verso il basso.

Il rapporto tra i bit utili inviati e il numero di bit totali è $\frac{2}{6} = \frac{1}{3}$. Tale rapporto rappresenta



Figura 10.6: Rete multistadio a 3 livelli

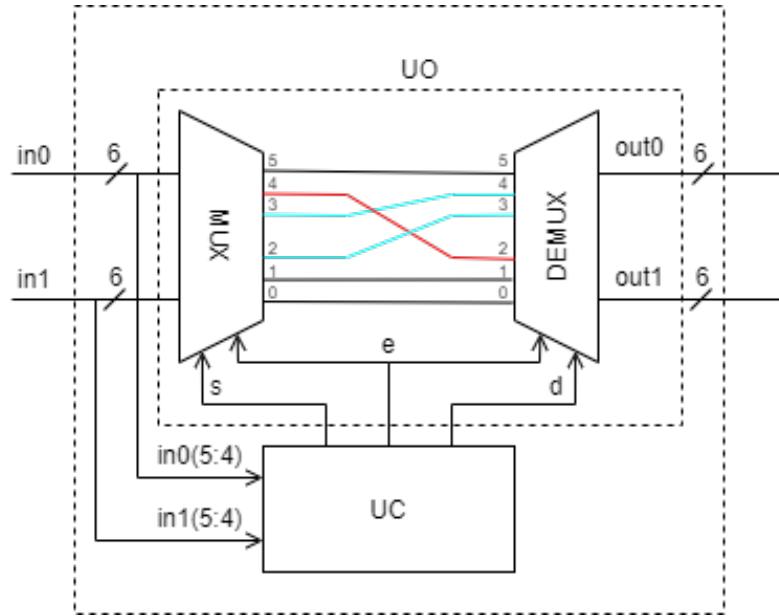


Figura 10.7: Architettura interna di uno switch 2:2 della rete con controllo distribuito

l'efficienza del sistema. Ipoteticamente, all'aumentare del numero di bit dato, aumenta l'efficienza ma anche la complessità degli switch.

Sono consentite le comunicazioni simultanee col rischio che vengano generate collisioni; infatti, si è scelto di realizzare una rete a perdita di pacchetto, dove per ogni collisione un messaggio viene scartato a seconda di una scelta prestabilita.

Non è stato implementato alcun meccanismo che informi la sorgente e/o la destinazione del pacchetto perso.

10.4.1 Switch 2:2

In figura 10.7 è rappresentata l'architettura interna di uno *switch 2:2* in cui sono messe in evidenza l'unità di controllo e l'unità operativa. Quest'ultima è costituita da un multiplexer e da un demultiplexer che, in base ai segnali di controllo (*s*, *d* ed *e*), pilota il messaggio in ingresso.

Per implementare un'architettura che abbia switch dello stesso tipo, non differenti a seconda del livello in cui vengono collocati, è stato implementato un meccanismo di shift che permette di considerare soltanto il bit più significativo nel campo destinazione del messaggio; tale bit è necessario per inoltrare il messaggio verso il corretto nodo del livello successivo.

Pertanto, in ingresso all'UC vi è il primo bit della destinazione di ciascuna linea e il bit di *idle*. Il meccanismo di shift citato in precedenza è stato progettato senza l'uso di uno shift register, ma semplicemente connettendo le interfacce MUX-DEMUX come mostrato in figura.

In questo modo non è stata introdotta una componente aggiuntiva che avrebbe reso lo switch più oneroso sia in termini di spazio che di tempo e soprattutto avrebbe reso la componente sequenziale. La logica di controllo implementata nell'UC consente di gestire anche la situazione in cui i bit *idle* di entrambe le linee sono alti. Infatti, in questo caso viene fatto passare solamente il messaggio trasmesso sulla linea *in0*, ignorando quello sulla linea *in1*.

10.4.2 Codice della rete di Interconnessione con controllo distribuito

Per rendere il codice più snello e leggibile sono stati definiti due tipi che verranno utilizzati nel corso dell'implementazione, ossia:

- subtype msg_type is std_logic_vector(5 downto 0)
- type vect_msg_type is array (0 to 7) of msg_type

Unità di Controllo

L'unità di controllo è stata implementata con un approccio dataflow.

```

1 entity uc is
2   port (
3     state1: in std_logic;
4     state2: in std_logic;
5     des1: in std_logic;
6     des2: in std_logic;
7     en: out std_logic;
8     s: out std_logic;
9     d: out std_logic
10    );
11 end uc;
12
13 architecture Dataflow of uc is
14
15 begin
16   en <= state1 OR state2;
17   d <= des1 when state1='1' else
18     des2 when state2='1' else
19     '0';
20
21   s <= '0' when state1='1' else
22     '1' when state2='1' else
23     '0';
24
25 end Dataflow;
```

Unità Operativa

L'unità operativa è stata implementata con un approccio strutturale.

In particolare, sono stati utilizzati i componenti multiplexer e demultiplexer dei quali omettiamo il codice poiché già descritti nei progetti precedenti.

```

1 entity uo is
2   port (
3     in0: in std_logic_vector(5 downto 0);
4     in1: in std_logic_vector(5 downto 0);
5     en: in std_logic;
```

```

6      s: in std_logic;
7      d: in std_logic;
8      out0: out std_logic_vector(5 downto 0);
9      out1: out std_logic_vector(5 downto 0)
10 );
11
12 end uo;
13
14 architecture Structural of uo is
15   component multiplexer is
16     port(
17       in0: in std_logic_vector(5 downto 0);
18       in1: in std_logic_vector(5 downto 0);
19       en: in std_logic;
20       s: in std_logic;
21       out0: out std_logic_vector(5 downto 0)
22     );
23   end component;
24
25   component demultiplexer is
26     port(
27       in0: in std_logic_vector(5 downto 0);
28       en: in std_logic;
29       d: in std_logic;
30       out0: out std_logic_vector(5 downto 0);
31       out1: out std_logic_vector(5 downto 0)
32     );
33   end component;
34
35   signal s_out0 : std_logic_vector(5 downto 0) := (others => '0');
36 begin
37
38   multi: multiplexer
39     port map(
40       in0 => in0,
41       in1 => in1,
42       en => en,
43       s => s,
44       out0 => s_out0
45     );
46
47   demulti: demultiplexer
48     port map(
49       in0(5) => s_out0(5),
50       in0(4) => s_out0(3),
51       in0(3) => s_out0(2),
52       in0(2) => s_out0(4),
53       in0(1) => s_out0(1),
54       in0(0) => s_out0(0),

```

```

55      en => en,
56      d => d,
57      out0 => out0,
58      out1 => out1
59  );
60
61 end Structural;

```

Switch 2:2

Lo *Switch 2:2* è stato implementato con un approccio strutturale collegando opportunamente unità di controllo e unità operativo come in figura 10.7.

```

1  entity switch2_2 is
2    port(
3      in0: in std_logic_vector(5 downto 0);
4      in1: in std_logic_vector(5 downto 0);
5      out0: out std_logic_vector(5 downto 0);
6      out1: out std_logic_vector(5 downto 0)
7    );
8  end switch2_2;
9
10 architecture Structural of switch2_2 is
11
12   component uo is
13     port(
14       in0: in std_logic_vector(5 downto 0);
15       in1: in std_logic_vector(5 downto 0);
16       en: in std_logic;
17       s: in std_logic;
18       d: in std_logic;
19       out0: out std_logic_vector(5 downto 0);
20       out1: out std_logic_vector(5 downto 0)
21     );
22   end component;
23
24   component uc is
25     port(
26       state1: in std_logic;
27       state2: in std_logic;
28       des1: in std_logic;
29       des2: in std_logic;
30       en: out std_logic;
31       s: out std_logic;
32       d: out std_logic
33     );
34   end component;
35
36   signal s_s : std_logic := '0';

```

```

37 signal s_d : std_logic := '0';
38 signal s_en : std_logic := '0';
39
40 begin
41
42     unitaoperativa: uo
43         port map(
44             in0 => in0,
45             in1 => in1,
46             en => s_en,
47             s => s_s,
48             d => s_d,
49             out0 => out0,
50             out1 => out1
51         );
52
53     unitacontrollo: uc
54         port map(
55             state1 => in0(5),
56             state2 => in1(5),
57             des1 => in0(4),
58             des2 => in1(4),
59             en => s_en,
60             s => s_s,
61             d => s_d
62         );
63
64 end Structural;

```

Rete

La rete è stata implementata con un approccio strutturale. Sono stati istanziati e collegati opportunamente i 12 *Switch 2:2* con il costrutto *generate*.

```

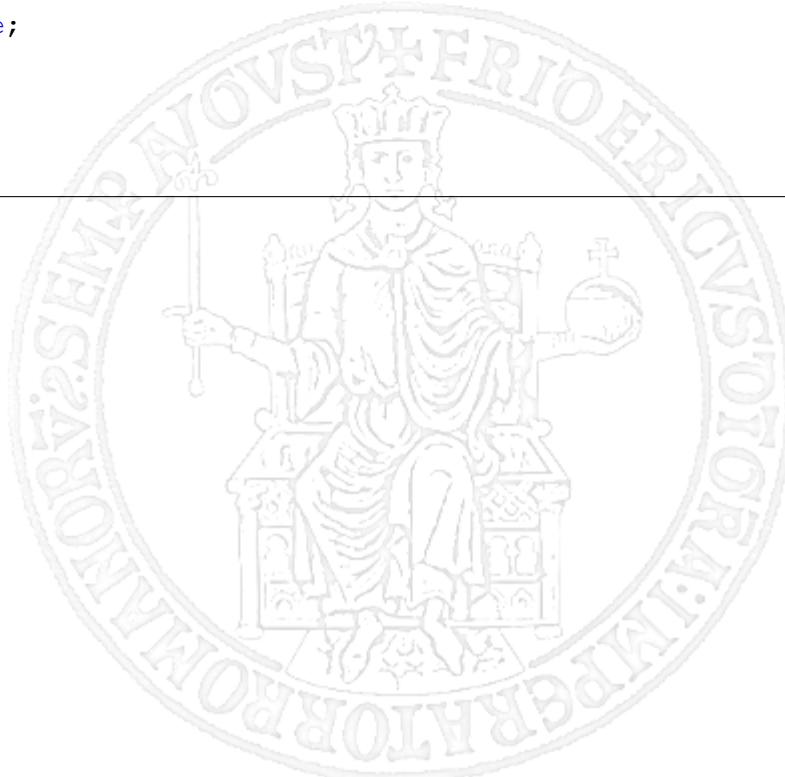
1
2 entity rete is
3     port(
4         in0: in vect_msg_type;
5         out0: out vect_msg_type
6     );
7 end rete;
8
9 architecture Structural of rete is
10
11 component switch2_2 is
12     port(
13         in0: in std_logic_vector(5 downto 0);
14         in1: in std_logic_vector(5 downto 0);
15         out0: out std_logic_vector(5 downto 0);

```

```

16      out1: out std_logic_vector(5 downto 0)
17  );
18 end component;
19
20 type s_vector_type is ARRAY(0 to 3) of vect_msg_type;
21 signal s_vector : s_vector_type := (others => (others => (others => '0')));
22
23 begin
24
25   out0 <= s_vector(3);
26   s_vector(0) <= in0;
27
28
29   switches_c : for j in 1 to 3 generate
30     switches_r : for i in 0 to 3 generate
31
32       switch : switch2_2
33
34         port map(
35           in0 => s_vector(j-1)(i),
36           in1 => s_vector(j-1)(i+4),
37           out0 => s_vector(j)(2*i),
38           out1 => s_vector(j)(2*i+1)
39         );
40
41   end generate;
42 end generate;
43
44
45 end Structural;

```



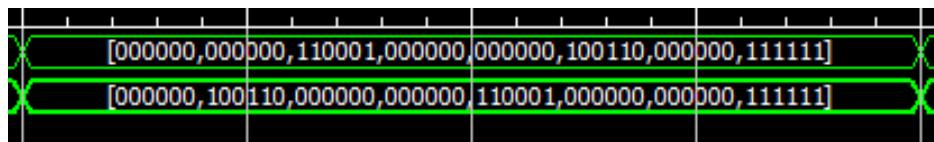


Figura 10.8: Simulazione della prima comunicazione

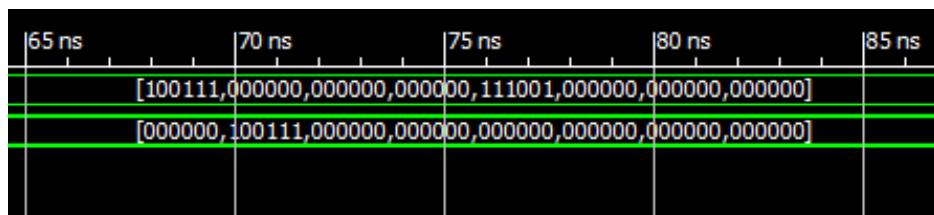


Figura 10.9: Simulazione della seconda comunicazione

10.4.3 Simulazione

Per testare il corretto funzionamento del sistema, sono state simulate le seguenti comunicazioni:

- $2 \rightarrow 4, 5 \rightarrow 1, 7 \rightarrow 7$: immagine in figura 10.10
- $4 \rightarrow 6, 0 \rightarrow 1$: immagine in figura 10.11

Come è possibile notare, le prime non generano collisioni, mentre le seconde collidono nel primo stadio. Infatti, osservando la simulazione in figura 10.8, i messaggi inviati dai nodi 2, 5 e 7 (riga superiore) arrivano alle uscite del dispositivo (riga inferiore). Ciò non accade nella simulazione in figura 10.9 dove è possibile constatare che solo uno dei messaggi inviati è arrivato a destinazione.

```

1 stim_proc: process
2 begin
3     -- hold reset state for 100 ns.
4     wait for 20 ns;
5
6     in0(2)<= "110001";
7     in0(5)<= "100110";
8     in0(7)<= "111111";
9
10    wait for 20 ns;
11
12    in0(2)<= "000000";
13    in0(5)<= "000000";
14    in0(7)<= "000000";
15
16    in0(4) <= "111001";
17    in0(0) <= "100111";
18
19    wait;
20 end process;
```

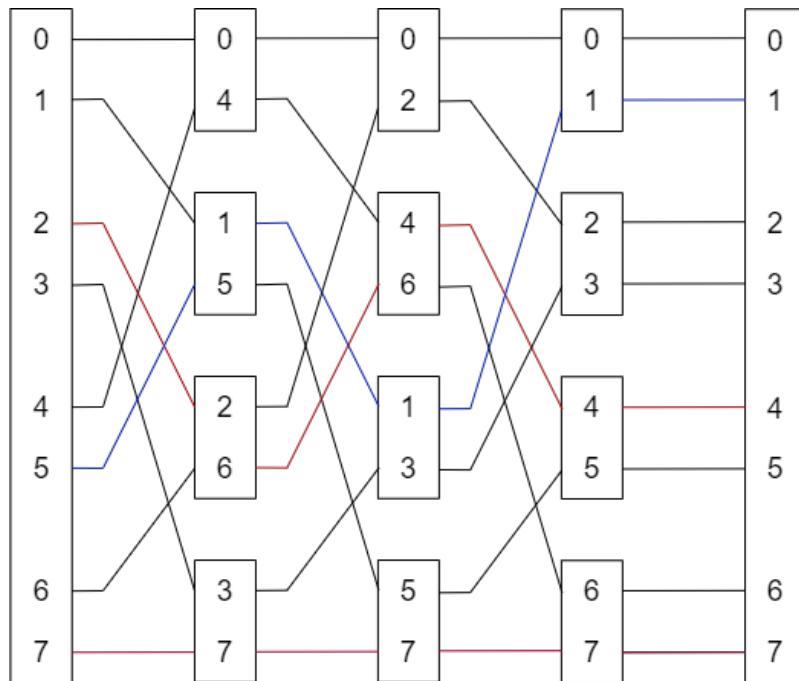


Figura 10.10: Rappresentazione di un utilizzo della rete senza collisione

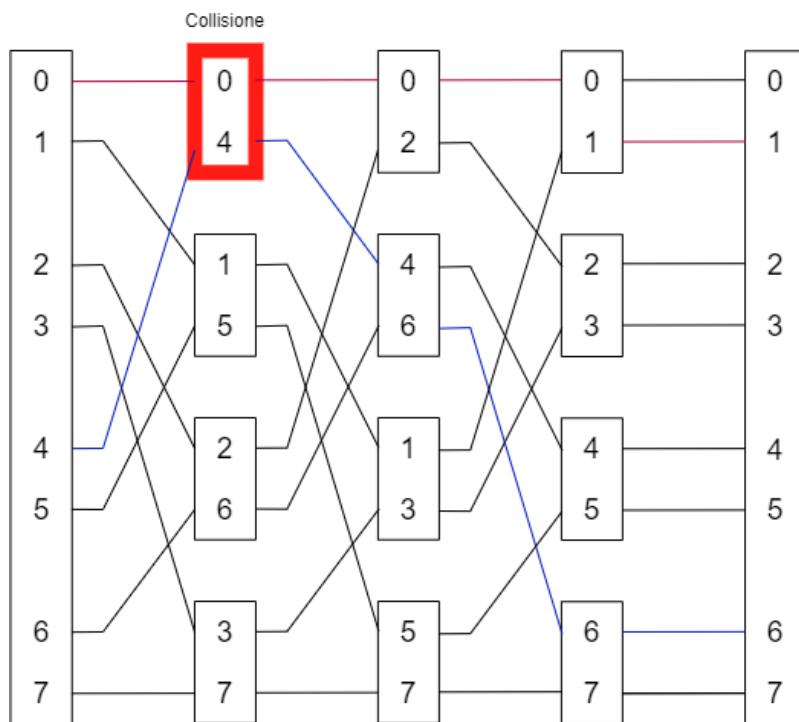


Figura 10.11: Rappresentazione di un utilizzo della rete con collisione

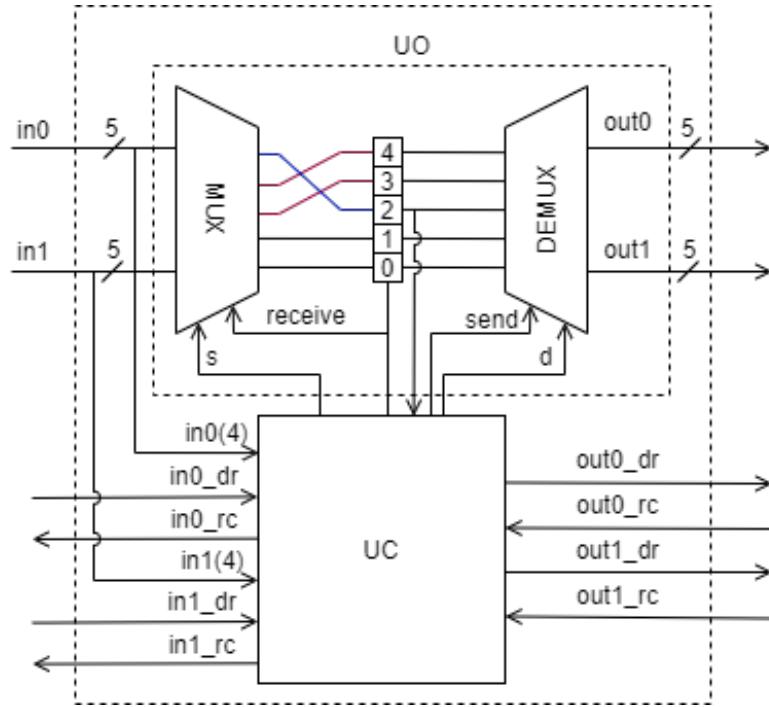


Figura 10.12: Architettura dello switch con protocollo

10.5 Switch con Protocollo

La risoluzione di questo esercizio verrà mostrata dal punto di vista teorico e progettuale senza effettuare l'implementazione in VHDL.

10.5.1 Architettura dello Switch 2:2

La progettazione dello switch deve prevedere che l'unità di controllo implementi un protocollo di comunicazione. Lo stesso protocollo viene utilizzato sia per quanto riguarda la ricezione del messaggio sia per l'invio.

La struttura del messaggio è la stessa di quella esposta nei paragrafi precedenti con l'unica differenza che non è stato utilizzato il bit di idle, non necessario grazie all'utilizzo del protocollo di comunicazione.

Stessa cosa per quanto riguarda il destinatario del messaggio che è selezionato a seconda del bit più significativo del campo destinazione.

La comunicazione adottata è quella *store & forward*. Il mittente instaura la comunicazione con un solo nodo della rete, il quale si occupa di memorizzare il messaggio e contattare il successivo nodo. Uno nodo della rete, per segnalare ad un altro la presenza di dati consistenti che intende inviarli, alza il segnale *Data Ready* (*dr*).

Il ricevente, una volta memorizzato il messaggio, avvisa il mittente dell'avvenuta ricezione tramite il segnale *Data Receive* (*rc*). Il protocollo si chiude col mittente e il ricevente che abbassano rispettivamente i segnali *dr* e *rc*.

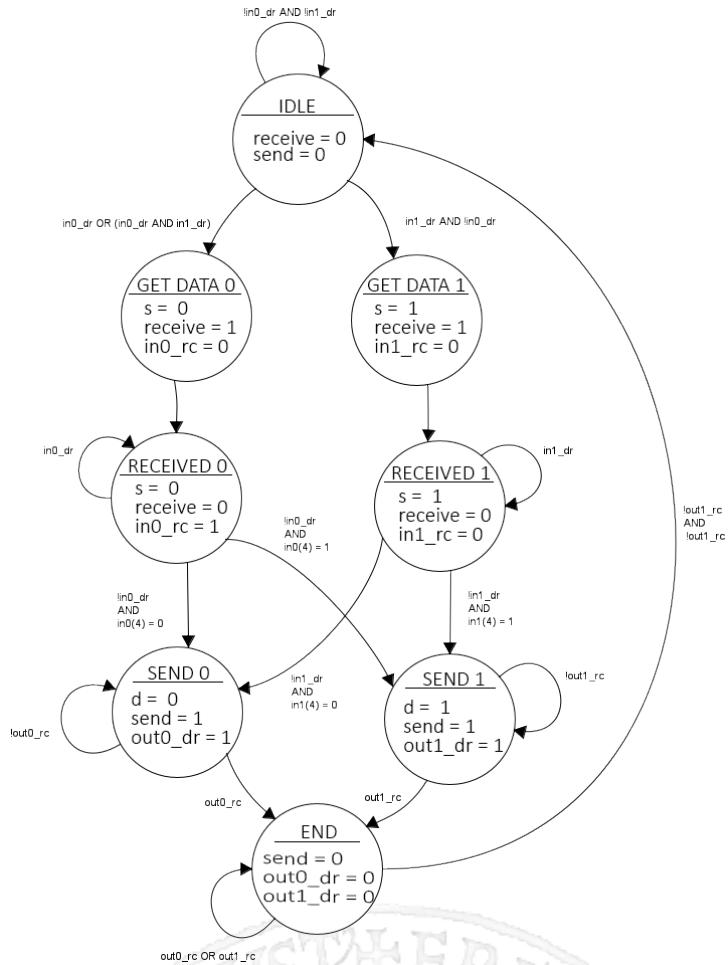


Figura 10.13: Automa che descrive il comportamento dell'unità di controllo

10.5.2 Unità Operativa

L'unità operativa, come evidenziato in figura 10.12, è costituita da un multiplexer e da un demultiplexer come illustrato nei paragrafi precedenti, ma con l'aggiunta di un registro a 5 bit per memorizzare il messaggio in modo da implementare la tecnica dello *store & forward*.

Un'altra caratteristica dell'unità operativa è che riceve, dall'unità di controllo, i segnali di *receive* e *sendD'or* per abilitare rispettivamente multiplexer/registro e demultiplexer.

I segnali di abilitazione *s* e *d* hanno lo stesso comportamento spiegato nelle implementazioni precedenti.

10.5.3 Unità di Controllo

L'unità di controllo presentata si differenzia da quella precedentemente proposta in quanto sono stati aggiunti i segnali necessari per la gestione del protocollo di handshaking. I nuovi segnali sono *data received* e *data ready*, utilizzati per controllare l'invio e la ricezione del messaggio.

La progettazione dell'unità di controllo è stata pensata in logica cablata, dato che la sequenza di operazioni da effettuare è breve e determinata; l'automa sviluppato è rappresentato in figura 10.13.

Capitolo 11

Moltiplicatore di Booth

11.1 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.

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

11.2 Soluzione moltiplicatore di Booth

Tra le macchine aritmetiche proposte dalla traccia, è stato scelto di sviluppare il moltiplicatore di Booth.

In figura 11.1 è mostrata l'architettura implementata. A differenza dello schema mostrato nel materiale didattico sono state effettuate alcune modifiche: il contatore è stato incorporato nell'unità di controllo, sono stati aggiunti i segnali start e finish, e sono stati esplicitati tutti i segnali di controllo. Il segnale di start serve ad avviare il moltiplicatore, mentre quello di finish, quando alto, comunica il completamento dell'operazione richiesta; entrambi sono stati inseriti poiché è stata considerata la possibilità di un sistema esterno che si interfaccia con la macchina oppure per un eventuale implementazione su una *evaluation board*.

11.2.1 Adder

Una prima scelta da effettuare è decidere come implementare l'addizionatore parallelo. È stato scelto di utilizzare un ripple-carry adder.

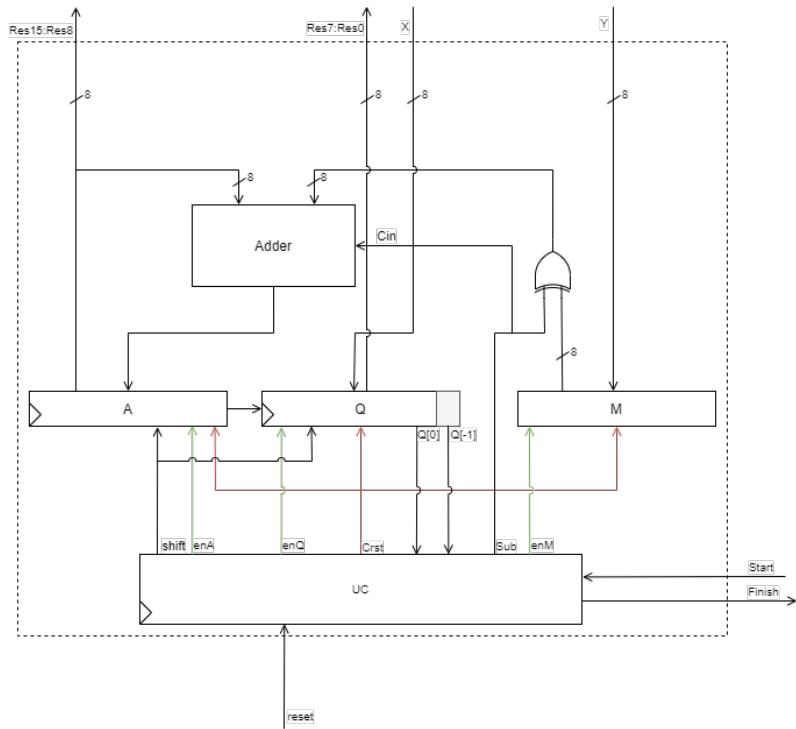


Figura 11.1: Architettura generale del moltiplicatore di Booth

Nonostante il tempo di esecuzione di questa tipologia di addizionatore cresce linearmente con il numero di bit da sommare, il numero ridotto di bit su cui sono rappresentati gli operandi richiesto dalla traccia permette il suo impiego senza che il tempo di esecuzione sia elevato. Qualora il numero di bit fosse maggiore, è possibile adottare una scelta più efficiente in termini di prestazioni (carry look-ahead o carry select) a scapito dell'area occupata.

In figura 11.2 è riportata l'architettura dell'adder. È stato previsto un riporto entrante affinché sia possibile effettuare somme algebriche tra dati codificati in complementi a 2.

La struttura semplice e ripetitiva di questa tipologia di addizionatore ha consentito l'implementazione di un'unica componente, il full adder, istanziata un numero di volte pari al numero di bit (8 in questo caso).

Sia Δ il ritardo di porta, il tempo di esecuzione di un FA è 2Δ . Complessivamente, il tempo di esecuzione dell'adder è $8 \cdot 2\Delta = 16\Delta$

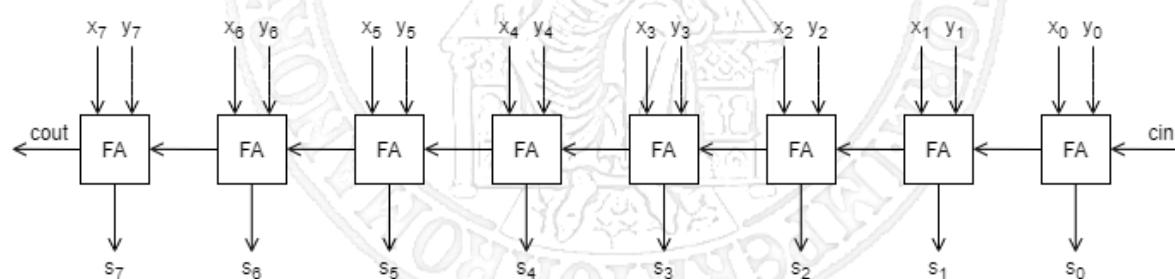


Figura 11.2: Schema adder ripple-carry

```

1 entity adder_sub is
2   port (
3     x: in std_logic_vector(7 downto 0);
4     y: in std_logic_vector(7 downto 0);
5     cin: in std_logic;
6     s: out std_logic_vector(7 downto 0);
7     cout: out std_logic
8   );
9
10 end adder_sub;
11
12 architecture Structural of adder_sub is
13
14   component full_Adder is
15     port (
16       x: in std_logic;
17       y: in std_logic;
18       cin: in std_logic;
19       cout: out std_logic;
20       s: out std_logic
21     );
22   end component;
23
24   signal rip : std_logic_vector(0 to 8) := (others =>'0');
25 begin
26
27   rip(0)<= cin;
28   cout<=rip(8);
29
30   fag: for i in 0 to 7 generate
31     fa: full_Adder
32     port map(
33       x => x(i),
34       y => y(i),
35       cin => rip(i),
36       cout => rip(i+1),
37       s => s(i)
38     );
39
40   end generate;
41
42 end Structural;

```

11.2.2 Shift Register

I registri A, Q e M sono implementati come shift-register con scorrimento verso destra e uscita e caricamento parallelo.

È stata definita una sola componente generale che potesse essere istanziata per tutti e 3 i registri nonostante abbiano scopi d'uso differenti. In particolare, il registro M poteva essere realizzato come un semplice registro a caricamento parallelo; utilizzando uno shift register per la sua implementazione, i segnali di controllo per lo scorrimento sono stati settati a 0.

Il registro A è il registro che contiene la somma parziale del prodotto durante l'esecuzione, mentre ad operazione completata contiene i primi 8 bit più significativi del risultato. Come ingresso seriale prende il bit più significativo dello stesso registro come imposto dell'algoritmo di Booth.

Il registro Q, invece, inizialmente contiene il moltiplicatore dell'operazione; ad operazione conclusa contiene i restanti 8 bit del risultato.

Il registro M contiene gli 8 bit del moltiplicando inserito nell'adder, il segnale di uscita è posto in ingresso ad una xor, così che possa essere complementato nel caso di una sottrazione.

```

1  entity shift_register is
2      port(
3          x: in std_logic_vector(7 downto 0);
4          serial_in : in std_logic;
5          clk: in std_logic;
6          reset: in std_logic;
7          en: in std_logic;
8          shift: in std_logic;
9          serial_out: out std_logic;
10         y: out std_logic_vector(7 downto 0)
11     );
12 end shift_register;
13
14 architecture Behavioral of shift_register is
15
16 signal memoria: std_logic_vector(7 downto 0) := (others => '0');
17 signal s_serial_out: std_logic := '0';
18 begin
19
20     serial_out <= s_serial_out;
21     y <= memoria;
22
23     p: process(clk, reset)
24         begin
25             if(reset='1') then
26                 memoria <= (others =>'0');
27             elsif(clk'event and clk='0') then
28                 if (en='1') then
29                     if(shift='0') then
30                         memoria <= x;
31                     elsif(shift='1') then
32                         memoria(6 downto 0) <= memoria(7 downto 1);
33                         memoria(7) <= serial_in;

```

```
34      s_serial_out <= memoria(0);  
35  end if;  
36  end if;  
37  end if;  
38  
39 end process;  
40  
41 end Behavioral;
```



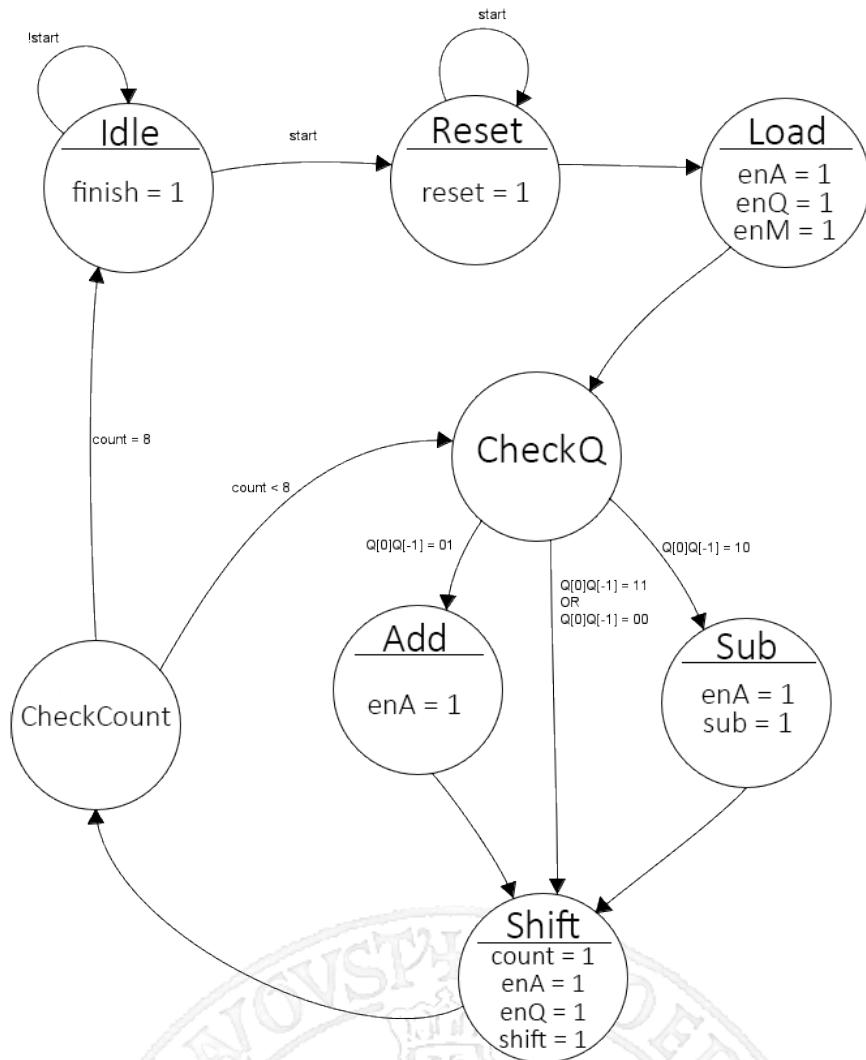


Figura 11.3: Automa dell'unità di controllo

11.2.3 Unità di Controllo

L'unità di controllo è stata implementata in logica microprogrammata. L'automa in figura 11.3 mostrato l'algoritmo di Booth per svolgere la moltiplicazione. Qui di seguito vi è riportata la descrizione degli stati.

Idle Si attende il segnale di start.

Reset Tutti i registri e il contatore nell'UC sono resettati. Si permane in questo stato fin quando start non si abbassa.

Load Vengono caricati nel registro M e Q il moltiplicando e il moltiplicatore.

CheckQ Viene controllato il bit meno significativo del registro Q e il bit uscente. Dalla codifica di Booth, se $Q = 01$, si effettua una somma; se $Q = 10$ si effettua una differenza; negli altri casi, viene traslato il registro A e Q.

Add È effettuata la somma tra il valore del registro accumulatore e il prodotto tra il moltiplicando e il bit Q_0 del moltiplicatore.

Sub Viene effettuata la sottrazione tra il valore del registro accumulatore e il prodotto tra il moltiplicando e il bit Q_0 del moltiplicatore.

Shift Vengono traslati i bit dei registri A e Q e viene incrementato un contatore interno.

CheckCount Se count è minore di 8, si effettuano le operazioni viste in precedenza. Quando count è uguale a 8 significa che l'operazione di moltiplicazione è terminata e si ritorna allo stato iniziale.

Come è possibile osservare in figura 11.4 e come detto precedentemente, il contatore è stato inserito all'interno dell'unità di controllo

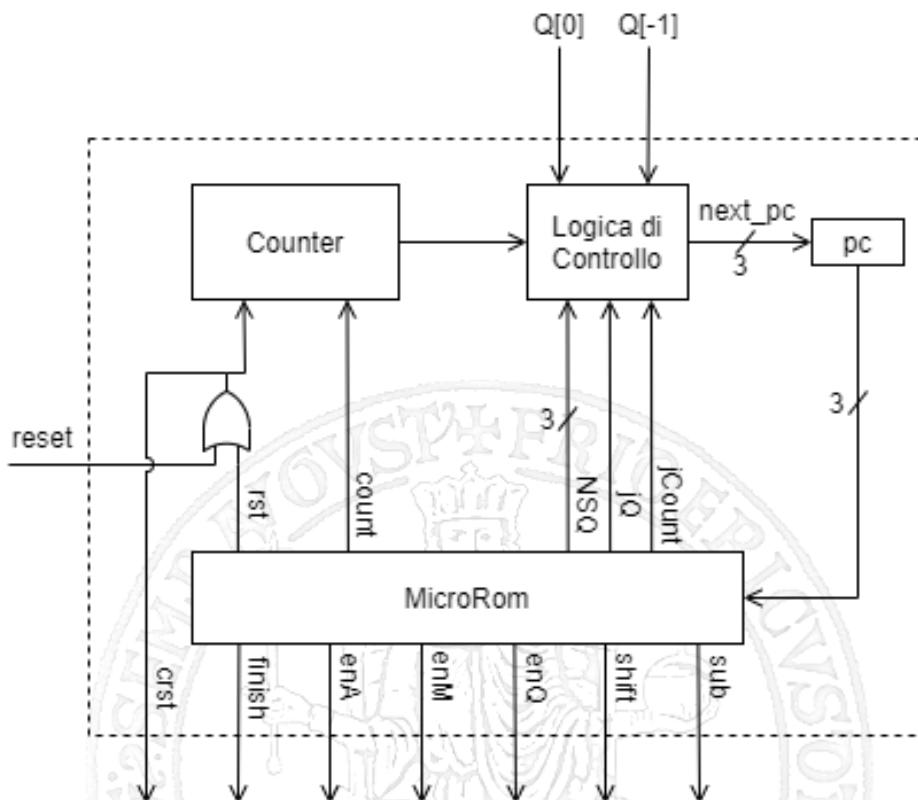


Figura 11.4: Architettura dell'unità di controllo

```

1 entity uc is
2 port (
3     start: in std_logic;
4     reset: in std_logic;
5     clk: in std_logic;
6     Q: in std_logic_vector(1 downto 0);
7     c_reset: out std_logic;
8     finish: out std_logic;

```

```

9      enA: out std_logic;
10     enM: out std_logic;
11     enQ: out std_logic;
12     shift: out std_logic;
13     set_sub: out std_logic
14   );
15
16 end uc;
17
18 architecture mixed of uc is
19
20   component microRom
21     port(
22     pc: in std_logic_vector(2 downto 0);
23     nsq: out std_logic_vector(2 downto 0);
24     finish: out std_logic;
25     jcount: out std_logic;
26     jq: out std_logic;
27     reset: out std_logic;
28     count: out std_logic;
29     enA: out std_logic;
30     enM: out std_logic;
31     enQ: out std_logic;
32     shift: out std_logic;
33     sub: out std_logic
34   );
35 end component;
36
37 component counter
38   port(
39     reset: in std_logic;
40     clk: in std_logic;
41     count_in: in std_logic;
42     count: out std_logic_vector(3 downto 0)
43   );
44 end component;
45
46 signal s_reset: std_logic := '0';
47 signal s_count_in: std_logic := '0';
48 signal s_count: std_logic_vector(3 downto 0) := (others=>'0');
49
50 signal pc: std_logic_vector(2 downto 0) := (others=>'0');
51 signal pc_next: std_logic_vector(2 downto 0) := (others=>'0');
52 signal s_nsq: std_logic_vector(2 downto 0) := (others=>'0');
53 signal s_jCount: std_logic := '0';
54 signal s_jQ: std_logic := '0';
55
56 signal var_r: std_logic := '0';
57

```

```

58 begin
59   c_reset<= s_reset or reset;
60   pc <= pc_next;
61   var_r <=reset or s_reset;
62
63   c: counter
64     port map(
65       reset=> var_r,
66       clk => clk,
67       count_in => s_count_in,
68       count => s_count
69     );
70
71
72 Rom: microRom
73   port map(
74     pc=>pc,
75     nsq=>s_nsq,
76     finish=>finish,
77     jcount=>s_jCount,
78     jq=>s_jQ,
79     reset=>s_reset,
80     count=>s_count_in,
81     enA=>enA,
82     enM=>enM,
83     enQ=>enQ,
84     shift=>shift,
85     sub=>set_sub
86   );
87
88
89 logical_control: process(clk,reset)
90 begin
91   if(reset='1')then
92     pc_next<= (others => '0');
93   elsif(clk'event and clk='1') then
94     if(start='1') then
95       pc_next<="001";
96     elsif(s_jCount='0' and s_jQ='0') then
97       pc_next <= s_nsq;
98
99     elsif(s_jQ='1')then
100      if(Q="01") then
101        pc_next<="100"; --ADD
102
103      elsif(Q="10") then
104        pc_next<="101"; --SUB
105      else
106        pc_next<=s_nsq;

```

```
107      end if;
108  elsif(s_jCount='1') then
109    if(unsigned(s_count)<8) then
110      pc_next<= s_nsq;
111    else
112      pc_next<="000"; -- IDLE
113
114      end if;
115
116    end if;
117  end if;
118
119
120
121
122  end process;
123
124 end mixed;
```



11.2.3.1 Control Store

NSQ	3	jCount	jQ	finish	reset	count	enA	enQ	enM	shift	sub
-----	---	--------	----	--------	-------	-------	-----	-----	-----	-------	-----

Figura 11.5: Formato della control word

La control word è composta da 13 bit:

- NSQ: il campo NSQ è composto da 3 bit ed indica l'indirizzo della prossima istruzione da eseguire.
- jCount: quando questo bit è alto, la logica di controllo interna deve controllare il valore di conteggio del contatore per determinare l'istruzione successiva da eseguire.
- jQ: quando questo bit è alto, la logica di controllo interna deve controllare i bit Q_0 e Q_{-1} per determinare l'istruzione successiva da eseguire.
- finish: segnale che indica la fine dell'operazione di moltiplicazione.
- reset: segnale per effettuare il reset di tutti i registri del sistema.
- count: bit per incrementare il valore di conteggio del contatore.
- enA: abilita la scrittura nel registro A.
- enQ: abilita la scrittura nel registro Q.
- enM: abilita la scrittura nel registro M.
- shift: quando questo bit è alto, i registri A e Q vengono shiftati.
- sub: il segnale è alto quando la codifica di Booth indica di effettuare una sottrazione.

Nella tabella in figura 11.6 è rappresentata la *control store* memorizzata nella microRom.

idle Il bit di finish è alto per indicare sia che il moltiplicatore non sta lavorando sia che è terminata l'operazione precedente.

reset In questa micro-istruzione il bit di reset è alto così da resettare il moltiplicatore

load In questa micro-istruzione i bit di enable sono alti in modo che i registri A,M e Q possano caricare i dati.

checkQ Questa micro-istruzione è necessaria per effettuare il controllo sui bit Q_0Q_{-1} . La prossima micro-istruzione puntata da *NSQ* è **shift** che viene eseguita quando i bit valgono 00/11. Negli altri casi, si effettua un salto all'istruzione add (01) o sub (10).

add in questa fase il bit di enA è alto perché viene salvato il risultato della somma nel registro A.

sub Il bit di sub è alto per eseguire l'operazione di sottrazione, insieme al bit enA per salvare il risultato nel registro A.

	Codifica	NSQ	Jcount	Jq	Finish	Reset	Count	EnA	EnQ	EnM	Shift	Sub
idle	000	000	0	0	1	0	0	0	0	0	0	0
reset	001	010	0	0	0	1	0	0	0	0	0	0
load	010	011	0	0	0	0	0	1	1	1	0	0
checkQ	011	110	0	1	0	0	0	0	0	0	0	0
add	100	110	0	0	0	0	0	1	0	0	0	0
sub	101	110	0	0	0	0	0	1	0	0	0	1
shift	110	111	0	0	0	0	1	1	1	0	1	0
checkCount	111	011	1	0	0	0	0	0	0	0	0	0

Figura 11.6: Control Store memorizzata nella microRom

shift I bit di shift e di enable per i registri A e Q sono alti per consentire lo shift di questi registri.

checkCount Il bit jCount è alto per controllare il conteggio del contatore.

11.3 Codice Moltiplicatore

Il moltiplicatore è stato implementato con un approccio strutturale, sono state collegate opportunamente le entità precedentemente descritte.

```

1 entity moltiplicatore is
2   port(
3     reset: in std_logic;
4     start: in std_logic;
5     x: in std_logic_vector(7 downto 0);
6     y: in std_logic_vector(7 downto 0);
7     clk: in std_logic;
8     finish: out std_logic;
9     res: out std_logic_vector(15 downto 0)
10    );
11 end moltiplicatore;
12
13 architecture Behavioral of moltiplicatore is
14
15 component adder_sub is
16   port(
17     x: in std_logic_vector(7 downto 0);
18     y: in std_logic_vector(7 downto 0);
19     cin: in std_logic;
20     s: out std_logic_vector(7 downto 0);
21     cout: out std_logic
22    );
23 end component;
24
25 component uc is
26   port(
27     start: in std_logic;
28     reset: in std_logic;
29     clk: in std_logic;
30     Q: in std_logic_vector(1 downto 0);

```

```

31      c_reset: out std_logic;
32      finish: out std_logic;
33      enA: out std_logic;
34      enM: out std_logic;
35      enQ: out std_logic;
36      shift: out std_logic;
37      set_sub: out std_logic
38  );
39 end component;

40
41 component shift_register is
42 port(
43     x: in std_logic_vector(7 downto 0);
44     serial_in : in std_logic := '0';
45     clk: in std_logic;
46     reset: in std_logic;
47     en: in std_logic;
48     shift: in std_logic;
49     serial_out: out std_logic := '0';
50     y: out std_logic_vector(7 downto 0)
51 );
52 end component;

53
54 signal s_sum: std_logic_vector(7 downto 0) := (others => '0');
55 signal s_sub: std_logic := '0';
56 signal s_notM: std_logic_vector(7 downto 0) := (others => '0');
57 signal s_M: std_logic_vector(7 downto 0) := (others => '0');
58 signal s_A: std_logic_vector(7 downto 0) := (others => '0');
59 signal s_c_reset: std_logic := '0';
60 signal s_enA: std_logic := '0';
61 signal s_shift: std_logic := '0';
62 signal s_a_q: std_logic := '0';
63 signal s_enQ: std_logic := '0';
64 signal s_enM: std_logic := '0';
65 signal s_q_out: std_logic := '0';

66
67
68 signal s_Q: std_logic_vector(7 downto 0) := (others => '0');
69
70 signal var_q: std_logic_vector(1 downto 0) := (others => '0');

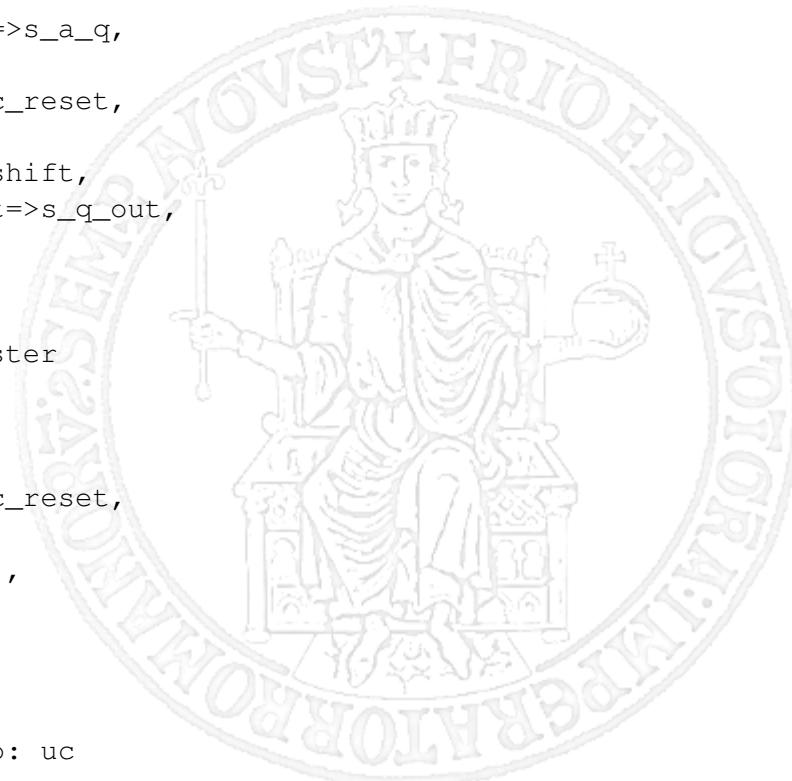
71 begin
72
73     o: for i in 0 to 7 generate
74         s_notM(i) <= s_M(i) xor s_sub;
75     end generate;
76
77
78     res <= s_A & s_Q;
79

```

```

80  var_q <=s_Q(0) & s_q_out;
81
82  ads: adder_sub
83    port map(
84      x=> s_A,
85      y=> s_notM,
86      cin=>s_sub,
87      s=>s_sum
88    );
89
90  A: shift_register
91    port map(
92      x=>s_sum,
93      serial_in=>s_A(7),
94      clk=>clk,
95      reset=>s_c_reset,
96      en=>s_enA,
97      shift=>s_shift,
98      serial_out=>s_a_q,
99      y=>s_A
100   );
101
102 Q: shift_register
103   port map(
104     x=>x,
105     serial_in=>s_a_q,
106     clk=>clk,
107     reset=>s_c_reset,
108     en=>s_enQ,
109     shift=>s_shift,
110     serial_out=>s_q_out,
111     y=>s_Q
112   );
113
114 M: shift_register
115   port map(
116     x=>y,
117     clk=>clk,
118     reset=>s_c_reset,
119     en=>s_enM,
120     shift=>'0',
121     y=>s_M
122   );
123
124
125 unitacontrollo: uc
126   port map(
127     start=>start,
128     reset=>reset,

```



```

129         clk=>clk,
130         Q => var_q,
131         c_reset => s_c_reset,
132         finish=> finish,
133         enA => s_enA,
134         enM => s_enM,
135         enQ => s_enQ,
136         shift => s_shift,
137         set_sub => s_sub
138     );
139
140 end Behavioral;

```

Figura 11.7: Simulazione dell'operazione di prodotto $4 \cdot 5$

11.4 Simulazione

Per il test del moltiplicatore, è stato eseguito il prodotto $4 \cdot 5$. In figura 11.7 è riportato l'andamento dei segnali durante la simulazione. Come è possibile notare, il valore del risultato varia ad ogni operazione dell'algoritmo di Booth, fin quando non si alza il segnale di *finish* che notifica il completamento del calcolo e l'assestamento del risultato.

```

1  stim_proc: process
2    begin
3
4      wait for 20 ns;
5
6      reset<='1';
7      wait for 10 ns;
8      reset<='0';
9
10
11     wait for 10 ns;
12     y<="00000101";
13     x<="00000100";
14
15     wait for 10 ns;
16     start<='1';
17
18     wait for 30 ns;
19     start<='0';
20
21     wait;
22   end process;
23
24 END;
```

Bibliografia

- [1] Conte G., Mazzeo A., Mazzocca N., Prinetto P., Architettura dei calcolatori, 1 ed., Città Studi Edizioni, 2015
- [2] Cozzolino G., Comunicazione tra dispositivi tramite interfaccia seriale, lucidi del corso
- [3] De Benedictis A., Mazzocca N, Macchine aritmetiche, lucidi del corso
- [4] Digilent RO, RS232 Reference Component, 2008
- [5] Mazzocca N., Moriconi A., Il Processore Mic-1, dispensa didattica del corso

