



Università degli Studi di Napoli Federico II  
Scuola Politecnica e delle Scienze di Base

---

# Progetto di Architettura dei Sistemi di Elaborazione

Gruppo 45

Reggio Raimondo (M63/1328)  
Pascarella Vincenzo (M63/1337)  
Riccio Emanuele (M63/1339)

---

# Indice

<i>1. Multiplexer</i> .....	5
1.1    Traccia .....	5
1.2    Traccia .....	9
1.3    Sintesi su board FPGA .....	12
<i>2. Encoder BCD</i> .....	14
2.1 Traccia .....	14
2.3 Simulazione .....	15
2.4 Sintesi su board FPGA.....	16
2.5 Timing Analysis.....	18
<i>3.Riconoscitore di Sequenze</i> .....	19
3.1 Traccia .....	19
3.2 Soluzione .....	19
3.2.1 Automa.....	19
3.2.2 Schematici.....	20
3.2.3 Codice .....	20
3.3    Simulazione .....	22
3.4    Sintesi su board FPGA .....	24
<i>4. Shift register</i> .....	27
4.1 Traccia .....	27
4.2 Cenni Teorici .....	27
4.3 Soluzione approccio strutturale .....	27
4.3.1 Schematici.....	28
4.3.2 Codice .....	28
4.4 Simulazione approccio strutturale .....	31
4.5 Soluzione approccio comportamentale.....	33
4.5.1 Schematici.....	33
4.5.2 Codice .....	34
4.6 Simulazione approccio comportamentale.....	35
<i>5. Cronometro</i> .....	38
5.1    Traccia 1 .....	38
5.1.1    Soluzione .....	38
5.1.1.1    Schematici .....	38
5.1.1.2    Codice .....	39
5.2    Traccia 2 .....	44
5.2.1    Soluzione .....	44
5.2.1.1    Schematici .....	44
5.2.2    Codice.....	45
5.3    Traccia 3 .....	50
5.3.1    Soluzione .....	50
5.3.1.1    Schematici .....	50
5.3.2    Codice.....	51
<i>6. Sistema di testing</i> .....	54
6.1    Traccia .....	54

6.2	Soluzione .....	54
6.2.1	Schematici .....	54
6.3	Codice.....	56
6.4	Simulazione .....	68
6.5	Sintesi su Board.....	69
<i>7.</i>	<i>Comunicazione con Handshaking</i> .....	73
7.1	Traccia .....	73
7.2	Cenni teorici .....	73
7.3	Soluzione .....	73
7.3.1	Automa Trasmettitore .....	74
7.3.2	Automa Ricevitore .....	74
7.3.3	Schematici.....	76
7.3.4	Codice .....	79
7.4	Simulazione .....	89
<i>8.</i>	<i>Il processore MIC-1</i> .....	91
8.1	Traccia .....	91
8.2	Introduzione.....	91
8.3	Architettura a stack.....	91
8.4	Datapath.....	91
8.5	Unità di Controllo.....	93
8.6	Flusso di controllo .....	95
<i>9.</i>	<i>Interfaccia Seriale</i> .....	99
9.1	Traccia .....	99
9.2	Traccia .....	104
<i>10.</i>	<i>Switch e Omega Network</i> .....	114
10.1	Traccia 1 .....	114
10.1.1	Soluzione.....	114
10.1.1.1	Schematici .....	114
10.1.2	Codice .....	116
10.1.3	Simulazione.....	119
<i>11.</i>	<i>Moltiplicatore di Robertson</i> .....	121
11.1	Traccia .....	121
11.2	Cenni teorici .....	121
11.3	Soluzione .....	123
11.3.1	Schematici.....	123
11.3.2	Codice .....	124
11.4	Simulazione .....	133
<i>12.</i>	<i>Guess The Sequence</i> .....	135
12.1	Traccia .....	135
12.2	Soluzione .....	135
12.3	Schematici.....	136
12.4	Codice .....	143



# 1. Multiplexer

## 1.1 Traccia

Progettare, implementare in VHDL e testare mediante simulazione un multiplexer indirizzabile 16:1, utilizzando un approccio di progettazione per composizione a partire da multiplexer 4:1.

### 1.1.1 Soluzione

Utilizzando l'approccio modulare, ovvero decomponendo la macchina da implementare in componenti più piccoli, abbiamo proceduto con l'implementazione di un multiplexer 16:1 tramite la composizione di quattro multiplexer 4:1. I multiplexer 4:1 essendo la cella fondamentale di questa macchina è stata implementata mediante una descrizione *Behavioral*, il multiplexer 16:1 invece è stato implementato mediante una descrizione *Structural* delle macchine da essa derivanti.

### 1.1.2 Schematici

Il multiplexer 4:1 è una macchina combinatoria notevole avente quattro ingressi e un'uscita. In particolare, abbiamo realizzato un multiplexer lineare, dunque per 4 ingressi servono 2 linee di selezione, in base alle quali il valore su uno degli ingressi sarà presentato in uscita. Nel complesso tale componente presenta 6 ingressi e un'uscita (*figura 1.1*).

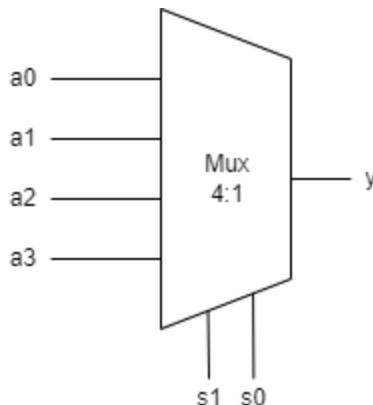


Figura 1.1 Schema multiplexer 4:1

Dalla composizione di cinque multiplexer 4:1 otteniamo un **multiplexer 16:1** (*figura 1.2*). Anche quest'ultimo è un multiplexer lineare: ha 16 ingressi, 4 fili di selezione e ovviamente un'uscita. I 16 ingressi entrano in 4 multiplexer, che essendo 4:1, prendono 4 ingressi e producono un'uscita ciascuno. Tali uscite entrano nel quinto multiplexer che produce l'output finale. Per quanto riguarda la selezione, concettualmente stiamo dividendo l'indirizzo in ingresso al multiplexer finale in due parti:

- 1) I bit meno significativi (c0 e c1) vanno in ingresso ai multiplexer del primo stadio, selezionando per ciascuno il valore di uscita.
- 2) I bit più significativi (c1 e c2) entrano nel multiplexer del secondo stadio e decidono quale degli ingressi provenienti dai multiplexer precedenti selezionare, ovvero quale dei valori in ingresso sarà presentato in uscita.

Ad esempio, se supponiamo di avere come selezione la stringa “0101” allora i bit meno significativi, pari a 01, selezioneranno dai multiplexer al primo stadio l’ingresso a1. I bit più significativi invece selezioneranno per l’ultimo mux l’uscita del secondo multiplexer al primo stadio, di conseguenza l’output finale sarà b1.

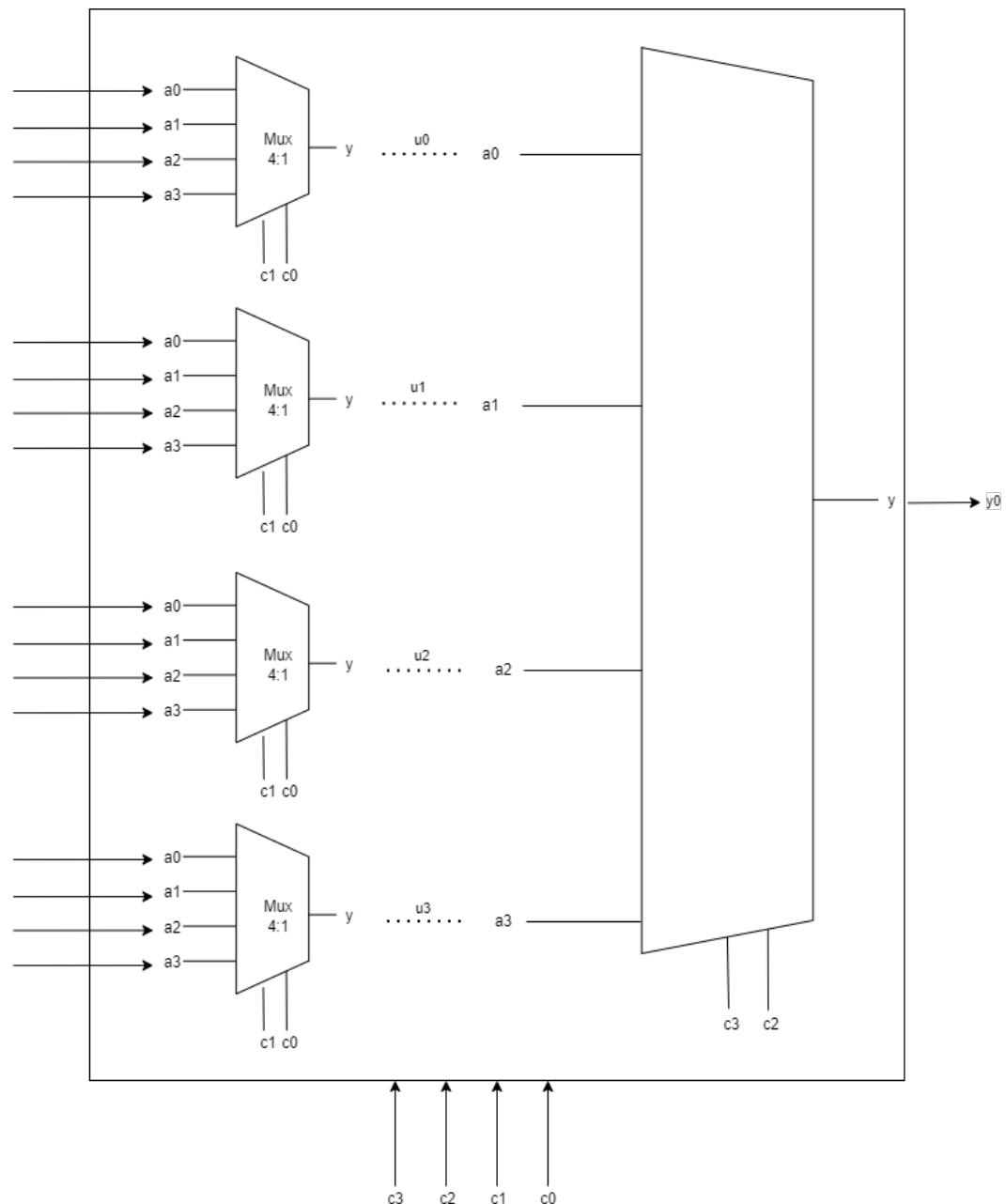


Figura 1.2: Schema multiplexer 16:1

### 1.1.3 Codice

#### Multiplexer 4:1

Partiamo dall'implementazione del multiplexer 4:1. Per prima cosa, bisogna definire l'entity mux\_4\_1, al cui interno dichiariamo le porte.

```
entity mux_4_1 is
    Port (
        a: in std_logic_vector(0 to 3);
        c: in std_logic_vector(1 downto 0);
        y: out std_logic
    );
end mux_4_1;

architecture Behavioral of mux_4_1 is
begin
    with c select
        y <= a(0) when "00",
        a(1) when "01",
        a(2) when "10",
        a(3) when "11",
        '-' when others;
end Behavioral;
```

Figura 1.3 Codice multiplexer 4:1

Dato che tale oggetto è il componente fondamentale della nostra architettura, l'abbiamo descritto utilizzando un approccio comportamentale, descrivendo esplicitamente la trasformazione che i dati subiscono. In particolare, il valore dell'uscita dipende dal valore assunto dall'ingresso di selezione. Per fare ciò abbiamo utilizzato un costrutto with-select, ovvero un'assegnazione condizionata. Notiamo che è presente un caso aggiuntivo di default, in quanto abbiamo definito i segnali come std\_logic, c(0) e c(1) non possono solo assumere valori 0 e 1 ma ciascuno dei 9 valori definiti per questo tipo, in tal caso l'uscita assumerà valore non specificato “ - ”.

#### Multiplexer 16:1

```
40 entity mux_16_1 is
41     Port (
42         b: in std_logic_vector(0 to 15);
43         s: in std_logic_vector(3 downto 0);
44         y0: out std_logic
45     );
46 end mux_16_1;
47
48
49
50 architecture Structural of mux_16_1 is
51
52
53
54     signal u: std_logic_vector(0 to 3);
55
56
57
58 component mux_4_1
59     port(
60         a: in std_logic_vector(0 to 3);
61         c: in std_logic_vector(1 downto 0);
62         y: out std_logic
63     );
64
65 begin
66
67     mux_0_3: for i in 0 to 3 generate m: mux_4_1
68         begin
69             port map(
70                 a(0 to 3) => b(i*4 to i*4+3),
71                 c(1 downto 0) => s(1 downto 0),
72                 y => u(i)
73             );
74         end generate;
75
76
77     mux_4: mux_4_1
78         port map(
79             a(0 to 3) => u(0 to 3),
80             c(1 downto 0) => s(3 downto 2),
81             y => y0
82         );
83 end Structural;
```

Figura 1.4: Codice multiplexer 16:1

Il multiplexer 16:1 è stato descritto in modo strutturale (figura 1.4). L'entity è un oggetto con 20 ingressi di cui 4 di selezione. Gli ingressi sono stati definiti come vettori std\_logic per facilitare l'implementazione; infatti, così facendo abbiamo potuto istanziare più oggetti analoghi tramite un ciclo for.

I componenti di questo oggetto sono i mux\_4\_1 realizzati precedentemente. Li definiamo nel blocco *component* e poi li istanziamo. Prima di fare ciò abbiamo definito un vettore u di 4 segnali per collegare le uscite dei mux al primo stadio con gli ingressi del multiplexer al secondo stadio, per farlo serve definire un segnale di appoggio. I primi quattro mux sono istanziati in un ciclo for perché lavorano allo stesso modo, prendendo però in ingresso un gruppo diverso degli ingressi della macchina esterna. Gli ingressi di selezione sono gli stessi, ovvero i due bit meno significativi. Il mux al secondo stadio, quindi, prende in ingresso il vettore u, ha come linee di selezione i due bit più significativi e collega la sua uscita con quella della macchina complessiva y0.

#### 1.1.4 Simulazione

Per effettuare la simulazione per prima cosa abbiamo creato il testbench (figura 1.5). La prima cosa che possiamo notare è che il corpo della entity è vuoto, questo infatti non è un oggetto che realizziamo ma serve solo per effettuare la simulazione e quindi verificare che il sistema realizzato funzioni correttamente. Il testbench non ha segnali di ingresso né di uscita in quanto non può essere istanziato da nessun blocco, ma istanza al suo interno altri blocchi per effettuarne il test. Nel nostro caso, l'oggetto da testare è il multiplexer 16:1 a cui colleghiamo i segnali definiti prima della sua istanziazione, tramite cui controlleremo che il sistema funzioni correttamente.

Definiamo un process, un algoritmo che il testbench esegue per effettuare il testing. Aspettiamo 10 ns e diamo in ingresso il vettore "1001000001011011" e come vettore di selezione "0011". Utilizzando le assert verifichiamo che l'output presentato corrisponda all'output atteso, in caso contrario segnaliamo l'errore e arrestiamo il processo. Prima di fare ciò aspettiamo però 10 ns, ovvero, aspettiamo che i transistori si stabilizzano e l'uscita vada a regime, in modo tale da evitare di vedere un valore differente da quello corretto e arrestare la simulazione. Come secondo caso di test abbiamo modificato la selezione in "1110", ci si aspetta in uscita il valore a(14) ovvero 1, verifichiamo ciò tramite l'assert dopo 10 ns e poi terminiamo il processo.

Lanciando la simulazione possiamo verificare quanto appena spiegato. In figura 1.6 sono riportati i dati della simulazione, nella quale i segnali assumono inizialmente valore indefinito perché non inizializzati. Dopo 10 ns i valori in input vengono modificati e viene presentata istantaneamente l'uscita, in quanto non abbiamo inseriti ritardi.

```

34  entity TB_mux_16_1 is
35  -- Port ();
36  end TB_mux_16_1;
37
38  architecture Behavioral of TB_mux_16_1 is
39
40  component mux_16_1 is
41    Port (
42      b: in std_logic_vector(0 to 15);
43      s: in std_logic_vector(3 downto 0);
44      y0: out std_logic
45    );
46  end component;
47
48  signal inputs: STD_LOGIC_VECTOR(0 to 15);
49  signal selections1: STD_LOGIC_VECTOR(3 downto 0);
50  signal outputs: STD_LOGIC;
51
52
53  mux: mux_16_1
54    port map (
55      b => inputs,
56      s => selections1,
57      y0 => outputs
58    );
59
60  stim_proc: process
61
62  begin
63
64    --selezione linea
65    wait for 10 ns;
66    inputs <= "1001000001011011";
67    selections1 <= "0011";
68    wait for 10 ns;
69    assert outputs = '1'
70    report "errore0"
71    severity failure;
72    wait for 10 ns;
73    inputs <= "1001000001011011";
74    selections1 <= "1110";
75    wait for 10 ns;
76    assert outputs = '1'
77    report "errore0"
78    severity failure;
79
80    wait;
81
82  end process;

```

Figura 1.5: Codice testbench multiplexer 4:1

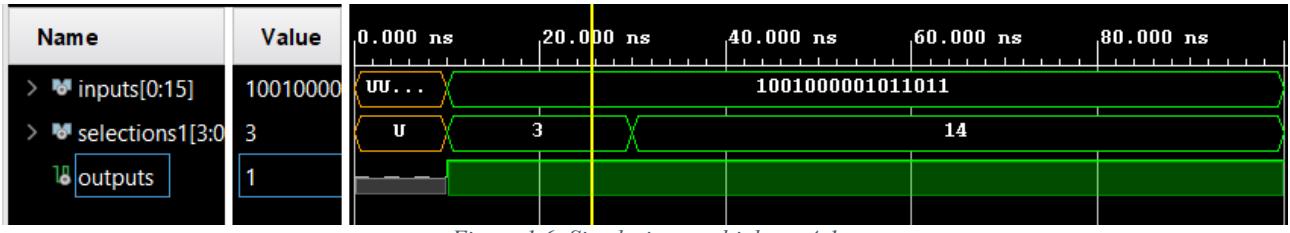


Figura 1.6: Simulazione multiplexer 4:1

## 1.2 Traccia

Utilizzando il componente sviluppato al punto precedente, progettare, implementare in VHDL e testare mediante simulazione una **rete di interconnessione a 16 sorgenti e 4 destinazioni**.

### 1.2.1 Soluzione

Utilizzando la composizione modulare è possibile scomporre la rete 16:4 in componenti più piccole. In particolare, tale macchina è composta da un multiplexer 16:1 e un demultiplexer 1:4. Si è quindi utilizzato il mux precedentemente realizzato per poi procedere con la realizzazione del demux utilizzando una strategia comportamentale in quanto esso rappresenta uno dei moduli fondamentali dell’architettura.

### 1.2.2 Schematici

Il demultiplexer 1:4 è una macchina combinatoria notevole avente un ingresso e quattro uscite. In particolare, abbiamo realizzato un demultiplexer lineare, dunque per 4 uscite servono 2 linee di selezione, in base alle quali il valore dell’ingresso sarà presentato su una delle uscite (figura 1.7).

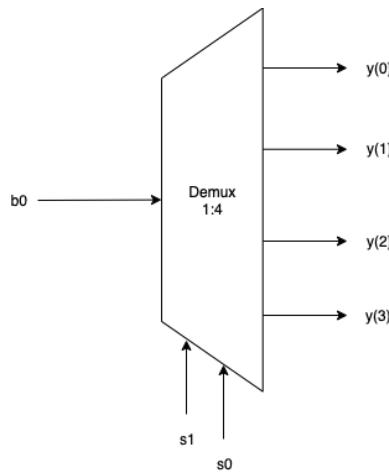


Figura 1.7: Demultiplexer 1:4

Dalla composizione del multiplexer 16:1 e del demultiplexer 1:4 si ottiene una macchina combinatoria con 22 ingressi, di cui 6 selezioni e 4 uscite (figura 1.8). I 16 ingressi entrano nel multiplexer, di questi sedici uno solo viene selezionato tramite i 4 bit di selezione meno significativi, ovvero: sel0, sel1, sel2, sel3. L’uscita del multiplexer viene presa in ingresso dal demux che presenta tale valore su uno dei 4 fili di output in base ai due bit di selezione più significativi (sel4 e sel5). Ad esempio, supponendo di avere in ingresso la stringa di bit “**0100100001000100**” e una selezione pari a “**100100**” vedremmo il valore “” sulla linea y(2).

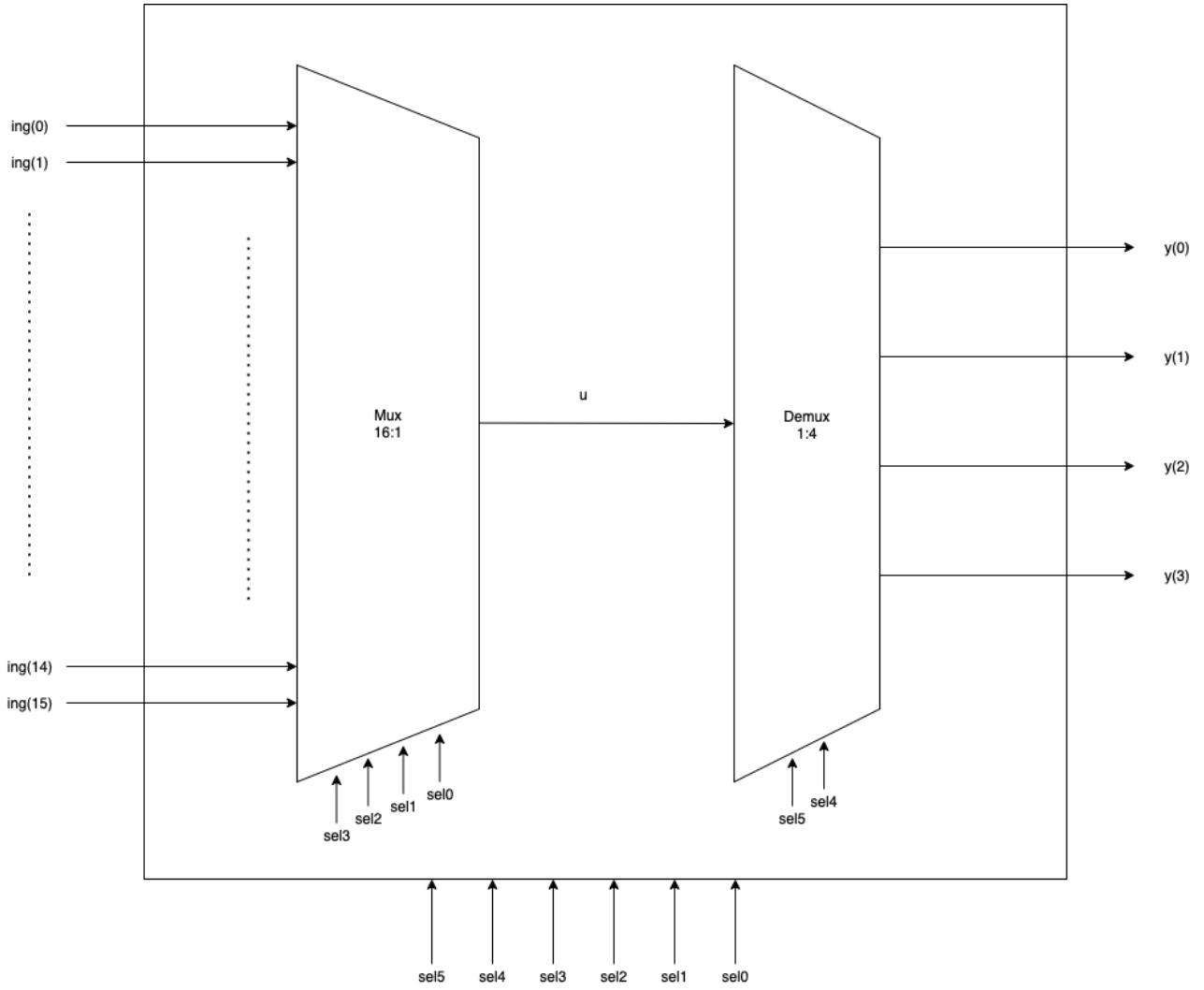


Figura 1.8: struttura rete 16:4

### 1.2.3 Codice

#### Demultiplexer 1:4

Partiamo dall'implementazione del demultiplexer 1:4, per prima cosa definiamo l'entity *demux\_1\_4* al cui interno dichiariamo le porte.

```

34  entity demux_1_4 is
35    Port (
36      b0: in std_logic;
37      s: in std_logic_vector(1 downto 0);
38      y: out std_logic_vector(0 to 3)
39    );
40 end demux_1_4;
41
42 architecture Behavioral of demux_1_4 is
43
44 begin
45 process (s,b0) is
46 begin
47 if (s(0) = '0' and s(1) = '0') then
48   y(0) <= b0; y(1) <= '0'; y(2) <= '0'; y(3) <= '0';
49 elsif (s(0) = '1' and s(1) = '0') then
50   y(1) <= b0; y(0) <= '0'; y(2) <= '0'; y(3) <= '0';
51 elsif (s(0) = '0' and s(1) = '1') then
52   y(2) <= b0; y(0) <= '0'; y(1) <= '0'; y(3) <= '0';
53 else
54   y(3) <= b0; y(0) <= '0'; y(1) <= '0'; y(2) <= '0';
55 end if;
56
57 end process;
```

Figura 1.9: Codice demultiplexer 1:4

Dato che tale oggetto è uno dei componenti fondamentali della nostra architettura, l'abbiamo descritto utilizzando un approccio comportamentale, descrivendo esplicitamente la trasformazione che i dati subiscono. In particolare, il valore dell'uscita dipende dal valore assunto dall'ingresso di selezione. Per fare ciò abbiamo utilizzato un costrutto if-then, ovvero un'assegnazione condizionata (*figura 1.9*).

### Rete 16:4

La rete 16:4 è stata descritta in modo strutturale (figura 1.10). L'entity è un oggetto con 22 ingressi di cui 6 di selezione. Gli ingressi sono stati definiti come vettori std\_logic per facilitare l'implementazione. I componenti di questo oggetto sono il mux\_16\_1 realizzato precedentemente e il demux\_1\_4. Li definiamo nel blocco *component* e poi li istanziamo. Prima di fare ciò abbiamo definito un segnale u per collegare l'uscita del mux con l'ingresso del demultiplexer, per farlo serve definire un segnale di appoggio.

#### 1.2.1 Simulazione

```

34 entity rete_16_4 is
35   Port (
36     ing: in std_logic_vector(0 to 15 );
37     sel: in std_logic_vector(5 downto 0);
38     y: out std_logic_vector(0 to 3)
39   );
40 end rete_16_4;
41
42
43 architecture Structural of rete_16_4 is
44
45 signal u: std_logic;
46
47
48 component mux_16_1
49   port(
50     b: in std_logic_vector(0 to 15);
51     s: in std_logic_vector(3 downto 0);
52     y0: out std_logic
53   );
54 end component;
55
56 component demux_1_4
57   port(
58     b0: in std_logic;
59     s: in std_logic_vector(1 downto 0);
60     y: out std_logic_vector(0 to 3)
61   );
62 end component;
63
64 begin
65   comp1: mux_16_1
66   port map(
67     b(0 to 15) => ing(0 to 15),
68     s(3 downto 0) => sel(5 downto 2),
69     y0 => u
70   );
71
72   comp2: demux_1_4
73   port map(
74     b0 => u,
75     s(1 downto 0) => sel(1 downto 0),
76     y(0 to 3) => y(0 to 3)
77   );

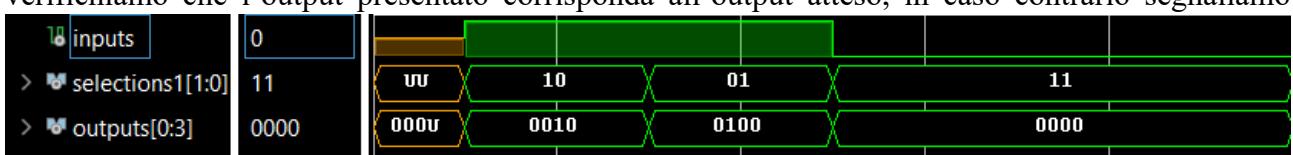
```

*Figura 1.10: Codice rete 16:4*

Per effettuare la simulazione per prima cosa abbiamo creato il testbench. Nel nostro caso, abbiamo testato prima il demultiplexer, in modo tale da evitare che eventuali errori commessi nella sua realizzazione si propaghino anche nell'architettura principale e successivamente la rete 16:4 a cui colleghiamo i segnali definiti prima della sua istanziazione, tramite cui controlleremo che il sistema funzioni correttamente.

I risultati della simulazione del demultiplexer sono riportati in *figura 1.11*, i casi di test effettuati sono: ingresso fisso ad '1' e selezione '10' e '01', un ulteriore caso di test è stato effettuato ponendo l'ingresso a '0' e le linee di selezione '11'.

Il testbench della rete 16:4 è invece mostrato in *figura 1.12*, in cui aspettiamo 10 ns e diamo in ingresso il vettore "0000000001000000" e come vettore di selezione "100100". Utilizzando le assert verifichiamo che l'output presentato corrisponda all'output atteso, in caso contrario segnaliamo



*Figura 1.11: Simulazione demux 1:4*

l'errore e arrestiamo il processo. Prima di fare ciò aspettiamo però 10 ns, ovvero, aspettiamo che i transistori si stabilizzano e l'uscita vada a regime, in modo tale da evitare di vedere un valore differente da quello corretto e arrestare la simulazione. Come secondo caso di test abbiamo modificato il valore di input in “0100010001001100” e la selezione in “000110”, ci si aspetta in uscita il valore a(1) ovvero 1 sulla linea y(2), verifichiamo ciò tramite l'assert dopo 10 ns e poi terminiamo il processo.

```

34      entity TB_rete_16_4 is
35      -- Port ();
36      end TB_rete_16_4;
37
38      architecture Behavioral of TB_rete_16_4 is
39
40          component rete_16_4 is
41              Port (
42                  ing : in STD_LOGIC_VECTOR(0 to 15);
43                  sel : in STD_LOGIC_VECTOR(5 downto 0);
44                  y : out STD_LOGIC_VECTOR(0 to 3)
45              );
46          end component;
47
48          signal inputs: STD_LOGIC_VECTOR(0 to 15);
49          signal selections1: STD_LOGIC_VECTOR(5 downto 0);
50          signal outputs: STD_LOGIC_VECTOR(0 to 3);
51
52      begin
53
54          but: rete_16_4
55              Port map (
56                  ing => inputs,
57                  sel => selections1,
58                  y => outputs
59              );
60
61          stim_proc: process
62          begin
63              wait for 10 ns;
64              inputs <= "0000000001000000";
65              selections1 <= "100100";
66              wait for 10 ns;
67              assert outputs <= "1000"
68              report "errore0"
69              severity failure;
70              wait for 10 ns;
71              inputs <= "0100010001001100";
72              selections1 <= "000110";
73              wait for 10 ns;
74              assert outputs <= "0100"
75              report "errore0"
76              severity failure;
77
78          end process;
79
80      end Behavioral;
81
82
83
84
85

```

Figura 1.12: Codice testbench rete 16:4

Lanciando la simulazione possiamo verificare quanto appena spiegato. In figura 1.13 sono riportati i dati della simulazione della rete 16:4, nella quale i segnali assumono inizialmente valore indefinito perché non inizializzati.

### 1.3 Sintesi su board FPGA

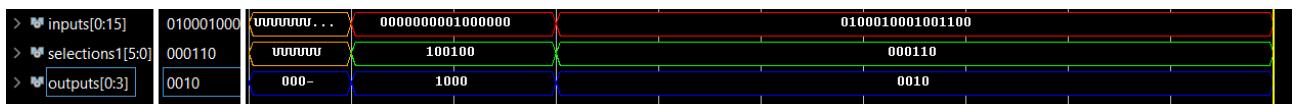


Figura 1.13: Simulazione rete 16:4

Per sintetizzare l'Encoder sull'FPGA è stato modificato il file dei *constraints* datoci in dotazione, in maniera tale da poter ricevere i valori in input, in particolare i valori di selezione dagli switch e mostrare l'output sui led. Per quanto riguarda i 16 bit in input alla rete essi sono stati precaricati, di conseguenza l'entity rete\_16\_4 è stata modificata opportunamente in maniera tale da permettere il pre-caricamento (figura 1.14).

```

33
34      entity rete_16_4 is
35          Port (
36              sel: in std_logic_vector(5 downto 0);
37              y: out std_logic_vector(0 to 3)
38          );
39      end rete_16_4;
40
41
42      architecture Structural of rete_16_4 is
43
44          signal u: std_logic;
45          signal ing: std_logic_vector(0 to 15) := "1010000111001010";
46

```

Figura 1.14: Codice rete 16:4 su board

Possiamo notare che l'ingresso è stato eliminato dall'entity principale ma è stato definito un vettore di 16 segnali in cui è stato già caricata una stringa di bit, tale segnale verrà poi utilizzato come input alla rete e di conseguenza in input al multiplexer.

## 2. Encoder BCD

### 2.1 Traccia

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

#### 2.1.1 Soluzione

Mediante un approccio comportamentale abbiamo proceduto all'implementazione di un Encoder BCD, che preso in ingresso una stringa binaria di 10 bit, nella quale solo uno dei 10 bit può essere alto in quanto essa è la rappresentazione decodificata di una cifra decimale, dà in output la sua rappresentazione in binario.

#### 2.1.2 Schemi

Un codificatore, ovvero un encoder, è un circuito digitale combinatorio dotato n segnali in ingresso e  $\log_2 n$  segnali di uscita. Nel nostro caso abbiamo 10 ingressi e 4 uscite come mostrato in figura 2.1.

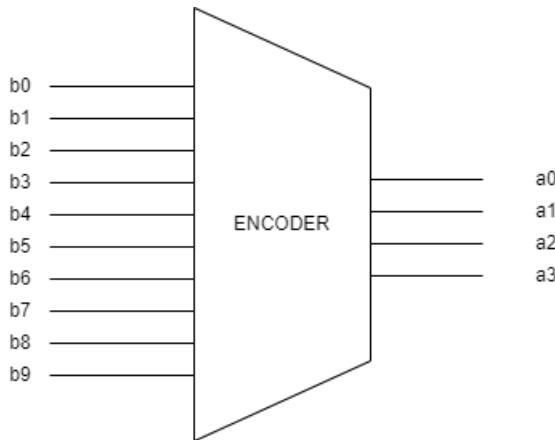


Figura 2.1: Encoder BCD

#### 2.1.3 Codice

Per prima cosa bisogna definire l'entity *encoder\_BCD*, al cui interno dichiariamo le porte. In questo caso abbiamo, come già si può notare nella figura 2.1, in ingresso un vettore composto da 10 bit e in uscita un vettore di 4 bit. In figura 2.2 è mostrata la descrizione behavioral dell'encoder nella quale descriviamo esplicitamente le trasformazioni che i dati subiscono, l'uscita dipende dal valore dell'ingresso. Notiamo che è presente un caso aggiuntivo di default in cui l'uscita vale "1111" ovvero la codifica binaria di 15, valore non previsto in ingresso ma utilizzato per coprire tutti gli ingressi non desiderati in quanto essendo il vettore in input un *std\_logic* e non essendo richiesto nessun controllo sui valori dei bit, più di uno di questi potrebbe essere alto contemporaneamente.

```

entity encoder_BCD is
  Port (
    ing: in std_logic_vector(9 downto 0);
    u: out std_logic_vector(3 downto 0);
  );
end encoder_BCD;

architecture Behavioral of encoder_BCD is

begin
  with ing select
    u <= "00000" when "0000000001",
      "0001" when "0000000010",
      "0010" when "0000000100",
      "0011" when "0000001000",
      "0100" when "0000010000",
      "0101" when "0000100000",
      "0110" when "0001000000",
      "0111" when "0010000000",
      "1000" when "0100000000",
      "1001" when "1000000000",
      "1111" when others;
end Behavioral;

```

Figura 2.2: Entity Encoder BCD

### 2.3 Simulazione

Per effettuare la simulazione per prima cosa abbiamo creato il **testbench**, il cui codice è mostrato in figura 2.3. Abbiamo definito il segnale *input* che viene collegato con l'ingresso dell'encoder e il segnale *output* collegato all'uscita dell'encoder e che useremo per testare la nostra unità. Il test è composto di un process in cui assegniamo i valori di input, nel caso in esame l'input è “0000100000” di conseguenza l'uscita che ci si aspetta è 5. Utilizzando le *assert* verifichiamo che l'output sia corretto e quindi che sia 5, in caso contrario segnaliamo l'errore e arrestiamo il processo. Notiamo che prima di effettuare tale controllo aspettiamo 10 ns, ovvero aspettiamo che i transistori si stabilizzino e l'uscita sia a regime, altrimenti rischiamo di vedere un valore differente e arrestare erroneamente la simulazione. Dopodiché effettuiamo due ulteriori test modificando i valori dell'input e controllando l'output.

```

34  entity TB_encoder_BCD is
35  end TB_encoder_BCD;
36  -
37  architecture Behavioral of TB_encoder_BCD is
38  -
39  component encoder_BCD
40    port(
41      ing: std_logic_vector(9 downto 0);
42      u: out std_logic_vector(3 downto 0);
43    );
44  end component;
45  -
46  signal input: std_logic_vector(9 downto 0);
47  signal output: std_logic_vector(3 downto 0);
48  -
49  begin
50    --Unit Under Test
51    uut: encoder_BCD
52    Port map (
53      ing => input,
54      u => output
55    );
56  -
57  stim_proc: process
58    begin
59      --selezione linea
60      wait for 10 ns;
61      input <= "0000100000";
62      wait for 10 ns;
63      assert output <= "0101"
64        report "errore0"
65        severity failure;
66      wait for 10 ns;
67      input <= "0010000000";
68      wait for 10 ns;
69      assert output <= "0111"
70        report "errore0"
71        severity failure;
72      wait for 10 ns;
73      input <= "0000000001";
74      wait for 10 ns;
75      assert output <= "1010"
76        report "errore0"
77        severity failure;
78      wait;
79    end process;
80  end Behavioral;

```

Figura 2.3: Codice TestBench Encoder BCD

Lanciando la simulazione possiamo verificare quanto appena spiegato. In figura 2.4 vediamo che inizialmente i segnali assumono valore interminato, in quanto non li abbiamo inizializzati, infatti, l'uscita in questo caso copre il caso *others* dando come output f ovvero "1111". Dopo 10 ns modifichiamo i valori in input per effettuare il primo test. Dopo altri 10 ns inseriamo ulteriori valori in ingresso all'encoder il quale ci restituisce l'output atteso. Infine, dopo ulteriori 10 ns eseguiamo l'ultimo test modificando ancora una volta l'input ottenendo così l'output previsto ovvero 0. Osserviamo che ogni volta che modifichiamo gli ingressi l'uscita varia a tempo 0, ovvero istantaneamente, questo perché l'implementazione dei componenti non abbiamo aggiunto ritardi, in caso contrario avremmo potuto notare che l'uscita varierebbe solo in seguito a un certo ritardo  $\Delta T$ .

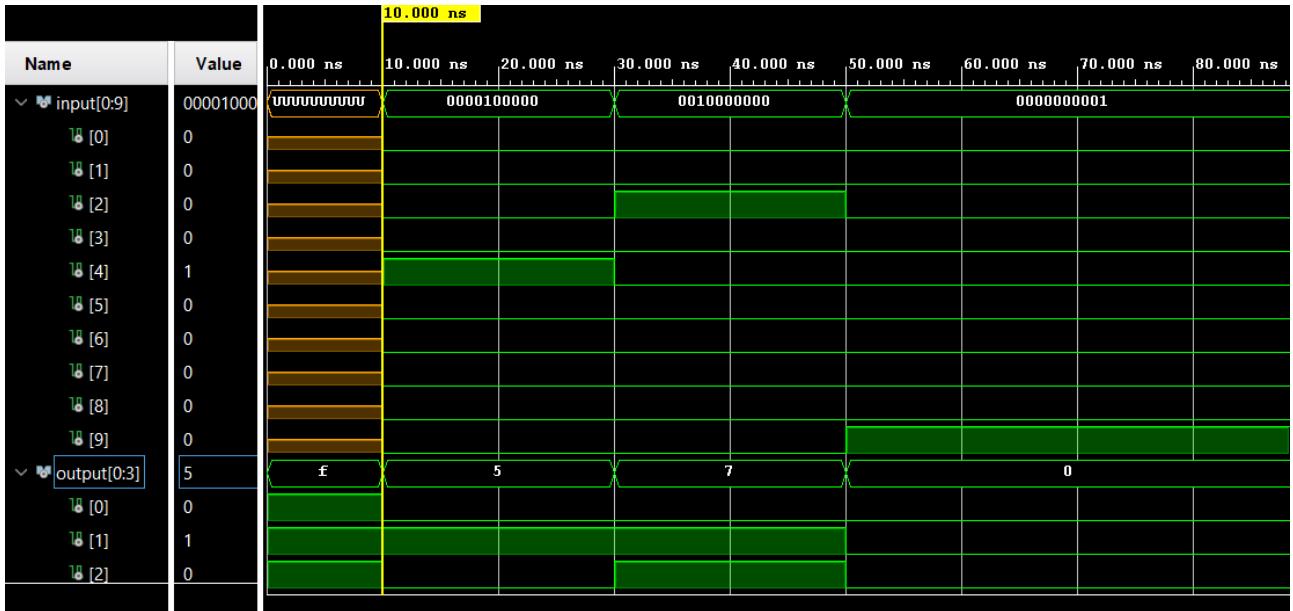


Figura 2.4: Simulazione Encoder BCD

## 2.4 Sintesi su board FPGA

Per sintetizzare l'Encoder sull'FPGA è stato modificato il file dei *constraints* datoci in dotazione, in maniera tale da poter ricevere i valori in input dagli switch e mostrare l'output sui led (figura 2.5 e 2.6).

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

#set_property SEVERITY {Warning} {get_drc_checks NSTD-1};
#set_property SEVERITY {Warning} {get_drc_checks UCIO-1};

##Switches
set_property -dict { PACKAGE_PIN I16      IOSTANDARD LVCMOS33 } [get_ports { ing[1] }]; #IO_L3N_T0_DQS_EMCCCLK_14 Sch=sv[1]
set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 } [get_ports { ing[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sv[2]
set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 } [get_ports { ing[3] }]; #IO_L13N_T2_MRCC_14 Sch=sv[3]
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 } [get_ports { ing[4] }]; #IO_L12N_T1_MRCC_14 Sch=sv[4]
set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMOS33 } [get_ports { ing[5] }]; #IO_L7N_T1_D10_14 Sch=sv[5]
set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMOS33 } [get_ports { ing[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sv[6]
set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMOS33 } [get_ports { ing[7] }]; #IO_L5N_T0_D07_14 Sch=sv[7]
set_property -dict { PACKAGE_PIN T8       IOSTANDARD LVCMOS18 } [get_ports { ing[8] }]; #IO_L24N_T3_34 Sch=sv[8]
set_property -dict { PACKAGE_PIN U8       IOSTANDARD LVCMOS18 } [get_ports { ing[9] }]; #IO_25_34 Sch=sv[9]
```

Figura 2.5: Acquisizione dei dati tramite switch

```

## LEDs
set_property -dict { PACKAGE_PIN H17  IOSTANDARD LVCMS33 } [get_ports { u[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15  IOSTANDARD LVCMS33 } [get_ports { u[1] }]; #IO_L24F_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13  IOSTANDARD LVCMS33 } [get_ports { u[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14  IOSTANDARD LVCMS33 } [get_ports { u[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
#set_property -dict { PACKAGE_PIN R18  IOSTANDARD LVCMS33 } [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17  IOSTANDARD LVCMS33 } [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
#set_property -dict { PACKAGE_PIN U17  IOSTANDARD LVCMS33 } [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
#set_property -dict { PACKAGE_PIN U16  IOSTANDARD LVCMS33 } [get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
#set_property -dict { PACKAGE_PIN V16  IOSTANDARD LVCMS33 } [get_ports { LED[8] }]; #IO_L18N_T2_A15_D31_14 Sch=led[8]
#set_property -dict { PACKAGE_PIN T15  IOSTANDARD LVCMS33 } [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
#set_property -dict { PACKAGE_PIN U14  IOSTANDARD LVCMS33 } [get_ports { LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
#set_property -dict { PACKAGE_PIN T16  IOSTANDARD LVCMS33 } [get_ports { LED[11] }]; #IO_L15N_T2_DQG_DOUT_C50_B_14 Sch=led[11]
#set_property -dict { PACKAGE_PIN V15  IOSTANDARD LVCMS33 } [get_ports { LED[12] }]; #IO_L16P_T2_C5E_B_14 Sch=led[12]
#set_property -dict { PACKAGE_PIN V14  IOSTANDARD LVCMS33 } [get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
#set_property -dict { PACKAGE_PIN V12  IOSTANDARD LVCMS33 } [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
#set_property -dict { PACKAGE_PIN V11  IOSTANDARD LVCMS33 } [get_ports { LED[15] }]; #IO_L21N_T3_DQG_A06_D22_14 Sch=led[15]

```

Figura 2.6: Output sui led

È stata successivamente realizzata un ulteriore implementazione su FPGA utilizzando il display seven segments datoci in dotazione. Abbiamo quindi creato una nuova entity (*figura 2.7*) al cui interno abbiamo dichiarato le porte necessarie al corretto funzionamento del dispositivo, ovvero: clock, reset, input, anodi e catodi. L’entity è stata realizzata utilizzando un approccio strutturale, quindi sono stati istanziati due componenti (*figura 2.7*): il componente encoder BCD presentato precedentemente e il display a sette segmenti, effettuando i collegamenti necessari e settandolo in modo opportuno.

```

entity encoder_bcd_display is
  Port (
    clock_in : IN std_logic;
    reset_in : IN std_logic;
    input : IN std_logic_vector(9 downto 0);
    anodes_out : OUT std_logic_vector(7 downto 0);
    cathodes_out : OUT std_logic_vector(7 downto 0)
  );
end encoder_bcd_display;

architecture Structural of encoder_bcd_display is
component encoder_BCD
  port(
    ing : in std_logic_vector(9 downto 0);
    ui_out : out std_logic_vector(3 downto 0)
  );
end component;
component display_seven_segments
  generic(
    CLKIN_freq : integer := 100000000; --frequenza del clock in input: quello della board è a 100MHz
    CLKOUT_freq : integer := 500 --frequenza dell'impulso in uscita, in corrispondenza del quale
                                --si scandisce ciascuna cifra (deve essere compreso fra 500Hz e 8KHz)
  );
  port(
    CLK : IN std_logic;
    RST : IN std_logic;
    VALUE : IN std_logic_vector(31 downto 0);--valori da mostrare sugli 8 display
    ENABLE : IN std_logic_vector(7 downto 0);--abilitazione di ciascuna cifra (accensione)
    DOTS : IN std_logic_vector(7 downto 0);--abilitazione punti (accensione)
    ANODES : OUT std_logic_vector(7 downto 0);
    CATHODES : OUT std_logic_vector(7 downto 0)
  );
end component;
begin
  signal y_temp : std_logic_vector(3 downto 0);
  signal reset_n : std_logic;
  enc: encoder_BCD
    port map(
      ing => input,
      ui_out => y_temp
    );
  seven_segment_array: display_seven_segments generic map(
    CLKIN_freq => 100000000, --qui inserisco i parametri effettivi (clock della board e clock in uscita desiderato)
    CLKOUT_freq => 500 --inserendo un valore inferiore si vedranno le cifre illuminarsi in sequenza
  )
    port map(
      CLK => clock_in,
      RST => reset_n,
      VALUE(31 downto 4) => "00000000000000000000000000000000",
      VALUE(3 downto 0) => y_temp,
      enable => "00000001", --stabilisco che tutti i display siano accesi
      dots => "00000000", --stabilisco che tutti i punti siano spenti
      anodes => anodes_out,
      cathodes => cathodes_out
    );
end Structural;

```

Figura 2.7: Entity Encoder su display

Possiamo notare come l’ingresso del display sia composto da 32 bit mentre l’uscita dell’encoder è composta da un `std_logic_vector` di 4 bit, il problema è stato risolto inserendo nei primi 28 bit in ingresso al display una stringa di 0 (*figura 2.8*). Inoltre è stato utilizzato un segnale di reset che assume il valore negato del reset dell’FPGA essendo quest’ultimo 0 attivo.

```

reset_n <= not reset_in;

enc: encoder_BCD
  port map(
    ing => input,
    ui_out => y_temp
  );

seven_segment_array: display_seven_segments generic map(
  CLKIN_freq => 100000000, --qui inserisco i parametri effettivi (clock della board e clock in uscita desiderato)
  CLKOUT_freq => 500 --inserendo un valore inferiore si vedranno le cifre illuminarsi in sequenza
)
  port map(
    CLK => clock_in,
    RST => reset_n,
    VALUE(31 downto 4) => "00000000000000000000000000000000",
    VALUE(3 downto 0) => y_temp,
    enable => "00000001", --stabilisco che tutti i display siano accesi
    dots => "00000000", --stabilisco che tutti i punti siano spenti
    anodes => anodes_out,
    cathodes => cathodes_out
  );
end Structural;

```

Figura 2.8: Collegamento dei componenti

## 2.5 Timing Analysis

Abbiamo effettuato la timing analysis dell'encoder BCD. Essendo essa una macchina combinatoria non è possibile effettuare tale analisi in quanto non sarebbe presente un'unità temporale di riferimento. Per risolvere il problema sono stati implementati due flip flop: uno utilizzato per ricevere il dato dall'esterno e presentarlo all'encoder e l'altro utilizzato per prelevar l'uscita della macchina combinatoria e mostrala in uscita. Entrambi i flip flop ricevono lo stesso clock in parallelo, quest'ultimo viene utilizzato come riferimento temporale.

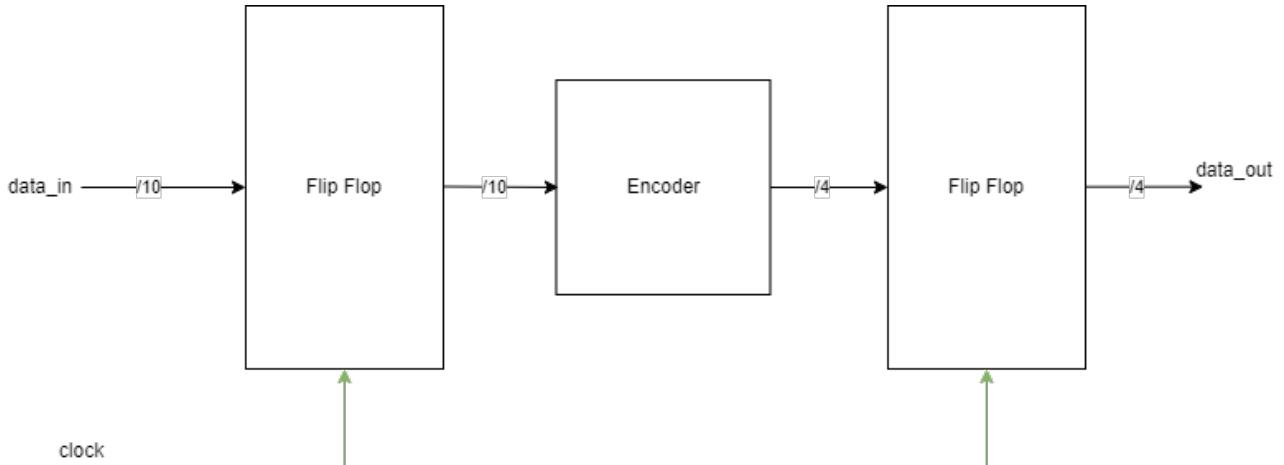


Figura 2. 9: Schema Time Analysis

Riportiamo di seguito un esempio dei risultati ottenuti.

```

211 Max Delay Paths
212 -----
213 Slack (MET) : 7.501ns (required time - arrival time)
214 Source: f1/data_out_reg[4]/C
215 (rising edge-triggered cell FDRE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
216 Destination: f2/data_out_reg[0]/D
217 (rising edge-triggered cell FDRE clocked by sys_clk_pin {rise@0.000ns fall@5.000ns period=10.000ns})
218 Path Group: sys_clk_pin
219 Path Type: Setup (Max at Slow Process Corner)
220 Requirement: 10.000ns (sys_clk_pin rise@10.000ns - sys_clk_pin rise@0.000ns)
221 Data Path Delay: 2.363ns (logic 0.897ns (37.960%) route 1.466ns (62.040%))
222 Logic Levels: 2 (LUT6=2)
223 Clock Path Skew: -0.145ns (DCD - SCD + CPR)
224 Destination Clock Delay (DCD): 2.704ns = ( 12.704 - 10.000 )
225 Source Clock Delay (SCD): 2.965ns
226 Clock Pessimism Removal (CPR): 0.116ns
227 Clock Uncertainty: 0.035ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
228 Total System Jitter (TSJ): 0.071ns
229 Total Input Jitter (TIJ): 0.000ns
230 Discrete Jitter (DJ): 0.000ns
231 Phase Error (PE): 0.000ns
232
233 Location Delay type Incr(ns) Path(ns) Netlist Resource(s)
---
```

Figura 2. 10: Esempio di risultato

# 3.Riconoscitore di Sequenze

## 3.1 Traccia

Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza **1001**. La macchina prende in ingresso un segnale binario  $i$  che rappresenta il dato, un segnale  $A$  di tempificazione e un segnale  $M$  di modo, che ne disciplina il funzionamento, e fornisce un'uscita  $Y$  alta quando la sequenza viene riconosciuta. In particolare,

- se  $M=0$ , la macchina valuta i bit seriali in ingresso a gruppi di 4,
- se  $M=1$ , la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta.

## 3.2 Soluzione

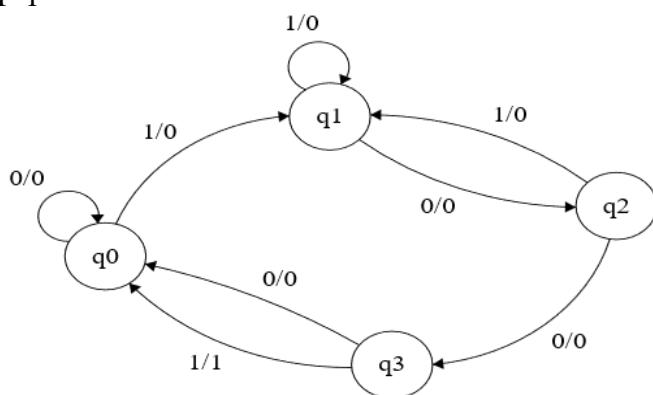
La macchina che vogliamo realizzare è un riconoscitore di sequenza, ovvero un sistema per cui dato un ingresso seriale, produce in uscita un valore binario che risulta essere alto nel caso in cui la sequenza in input sia “1001”. Dunque, la macchina è caratterizzata da tre ingressi: un segnale di tempificazione ( $clk$ ), un segnale di modo ( $m$ ) ed il dato ( $x$ ).

Innanzitutto, abbiamo realizzato l'automa che permette di descrivere con precisione e in maniera formale il comportamento del sistema. A partire dall'automa abbiamo sviluppato il sistema in VHDL.

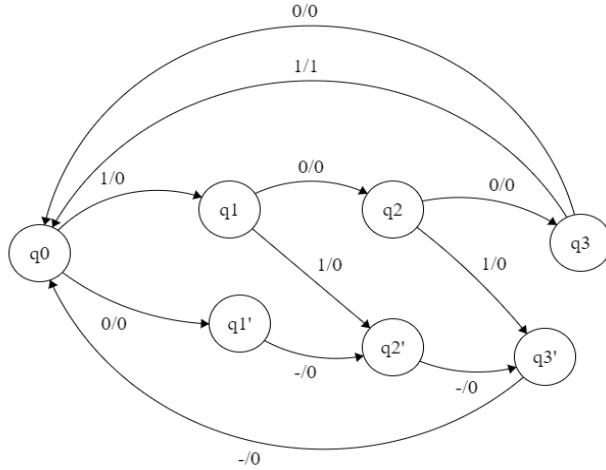
### 3.2.1 Automa

In primo luogo, l'automa differisce per il modo di funzionamento del sistema. Infatti, se  $M=0$ , la macchina valuta i bit seriali in ingresso a gruppi di 4, a differenza della modalità con  $M=1$  per la quale la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta. Quindi se ad esempio in ingresso ricevessimo due volte 1, in entrambi i modi di funzionamento passeremmo dapprima allo stato  $q_1$ , poi nel caso di  $M=1$  rimarremmo nello stato  $q_1$ , invece con  $M=0$ , andremmo in un ulteriore stato per la quale la sequenza risulterà scorretta.

- $M=1$



- $M=0$



Entrambi risultano essere automi a stati finiti di Mealy, infatti i valori di uscita sono determinati dallo stato attuale e dall'ingresso corrente, a differenza della macchina di Moore, che invece lavora solo in funzione dello stato corrente.

### 3.2.2 Schematici

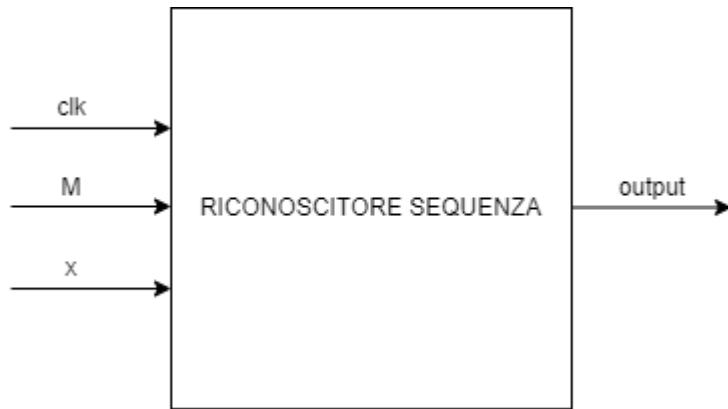


Fig. 3.1 Schema riconoscitore di sequenza

### 3.2.3 Codice

Partiamo dall'entity, abbiamo definito il riconoscitore come un oggetto caratterizzato da 3 input e un solo output, tutto di tipo std\_logic, come mostra la figura 3.2.

Il comportamento della macchina è stato descritto a *livello comportamentale*, infatti, l'architecture è realizzata da un process caratterizzato dalla presenza del clock nella sensitivity lists (figura 3.3). Essa presenta tutti i segnali a cui il processo è sensibile. Se uno dei segnali cambia, il processo si sveglia e il codice al suo interno viene eseguito.

In primis sono stati definiti gli stati che caratterizzano la nostra macchina. Successivamente, tramite un costrutto “if”, abbiamo differenziato il caso in cui  $m=1$  dal caso in cui  $m=0$  ed implementato gli automi corrispondenti tramite il costrutto “when...case”.

```

entity riconoscitore_seq is
Port (
    clk: in std_logic;
    m: in std_logic;
    x: in std_logic;
    y: out std_logic
);
end riconoscitore_seq;

architecture Behavioral of riconoscitore_seq is

type stato is (q0,q1,q2,q3,q11,q22,q33);
signal stato_corrente : stato := q0;

begin
process(clk)
begin
    if rising_edge (clk) then
        if m' event then
            stato_corrente <= q0;
            y <= '0';
        end if;
        if(m= '1') then      --riconoscitore un bit alla volta
            case stato_corrente is
                when q0 =>
                    if(x = '1') then stato_corrente <= q1; y <= '0';
                    else stato_corrente <= q0; y <= '0';
                    end if;
                when q1 =>
                    if(x = '1') then stato_corrente <= q1; y <= '0';
                    else stato_corrente <= q2; y <= '0';
                    end if;
                when q2 =>
                    if(x = '1') then stato_corrente <= q1; y <= '0';
                    else stato_corrente <= q3; y <= '0';
                    end if;
                when q3 =>
                    if(x = '1') then stato_corrente <= q0; y <= '1';
                    else stato_corrente <= q0; y <= '0';
                    end if;
                when others =>
                    stato_corrente <= q0;
                    y <= '0';
            end case;
        else case stato_corrente is
                when q0 =>
                    if(x = '1') then stato_corrente <= q1; y <= '0';
                    else stato_corrente <= q11; y <= '0';
                    end if;
                when q1 =>
                    if(x = '1') then stato_corrente <= q22; y <= '0';
                    else stato_corrente <= q2; y <= '0';
                    end if;
            end case;
        end if;
    end if;
end process;
end Behavioral;

```

Fig. 3.2 Entity riconoscitore di sequenza

```

when q2 =>
    if(x = '1') then stato_corrente <= q33; y <= '0';
    else stato_corrente <= q3; y <= '0';
    end if;
when q3 =>
    if(x = '1') then stato_corrente <= q0; y <= '1';
    else stato_corrente <= q0; y <= '0';
    end if;
when q11 =>
    stato_corrente <= q22; y <= '0';
when q22 =>
    stato_corrente <= q33; y <= '0';
when q33 =>
    stato_corrente <= q0; y <= '0';
when others =>
    stato_corrente <= q0;
    y <= '0';
end case;
end if;
end if;
end process;
end Behavioral;

```

Fig. 3.3 Codice VHDL riconoscitore di sequenza

### 3.3 Simulazione

Per effettuare la simulazione è stato realizzato il testbench in figura 3.4. Abbiamo fissato il periodo del clock con rispettivo process che definisce il comportamento del segnale stesso. Inoltre, sono stati determinati tutti i segnali responsabili della simulazione con m inizializzato a 1. Dopo 210 ns, ogni 10 ns, settiamo i valori del dato in ingresso in modo tale che la sequenza venga riconosciuta. Come è possibile vedere dalla figura 3.5, la macchina ha riconosciuto la sequenza, di conseguenza ha fornito un'uscita alta che poi è ritornata bassa in corrispondenza dell'input successivo.

```

entity TB_riconoscitore_seq is
end TB_riconoscitore_seq;

architecture Behavioral of TB_riconoscitore_seq is

component riconoscitore_seq
port(
    clk: in std_logic;
    m: in std_logic;
    x: in std_logic;
    y: out std_logic
);
end component;

--Inputs
signal x : std_logic := '0';
signal clk: std_logic := '0';
signal m: std_logic := '1';

--Outputs
signal y : std_logic;

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

BEGIN

```

```

uut: riconoscitore_seq port map(
    x => x,
    clk => clk,
    m => m,
    y => y
);

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

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

    wait for CLK_period*10;

    -- insert stimulus here
    x<='0';
    wait for 10 ns;
    x<='1';
    wait for 10 ns;
    x<='0';
    wait for 10 ns;
    x<='1';
    wait for 10 ns;
    x<='0';
    wait for 10 ns;
    x<='1';
    wait for 10 ns;
    x<='0';
    wait for 10 ns;
    x<='1';

    wait;
end process;

```

Fig. 3.4 Testbench riconoscitore di sequenza

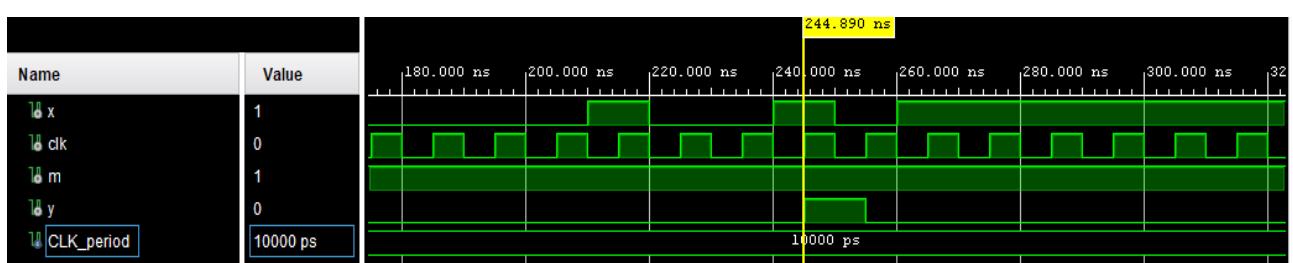


Fig. 3.5 Simulazione riconoscitore di sequenza

## 3.4 Sintesi su board FPGA

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

Per effettuare la sintesi sulla board Nexys A7 100T è stato utilizzato il componente sviluppato nei punti precedenti, con opportune modifiche al fine di acquisire l'input in combinazione con i bottoni btn1 e btn2 utilizzati rispettivamente per il dato in ingresso e il modo (figura 3.6).

```
if(read2 = '1' and last_read2 = '0') then
    if mode /= m then
        m <= mode;
        stato_corrente <= q0;
        y <= '0';
    end if;
    last_read2 <= '1';
else
    last_read2 <= '0';
end if;
```

Fig. 3.6 Codice VHDL per btn2

Per brevità è stata inserita solo la gestione del bottone utilizzato per il modo. Abbiamo definito un segnale last\_read2 per individuare il fronte di salita del segnale, dato che non è possibile utilizzare le funzioni rising\_edge o falling\_edge per segnali a cui è stato effettuato il debounce.

Quindi, se il bottone ora è premuto e precedentemente non lo era (last\_read2 = '0'), viene eseguito il codice all'interno del primo costrutto if nel quale controlliamo che l'input del modo di funzionamento (mode) sia diverso dal precedente e lo assegniamo al segnale m che poi utilizziamo per differenziare il comportamento della macchina. Infine, poniamo last\_read2 a 1 dato che all'ultima lettura il bottone è stato premuto.

In caso contrario last\_read2 è 0 visto che quest'istruzione viene eseguita solo se il bottone all'ultima lettura non è stato premuto.

Considerando che quando si premono i pulsanti su board ci sono rimbalzi imprevedibili che sono indesiderati, abbiamo effettuato il debounce dei pulsanti generando solo un singolo impulso con un periodo del clock di ingresso quando il pulsante su board viene premuto e rilasciato.

Infine, il top level module è stato descritto a livello Strutturale. Esso è caratterizzato da tre componenti:

- ButtonDebouncer (btn1)
- ButtonDebouncer (btn2)
- riconoscitore\_seq

Come possiamo notare dal codice sottostante, vi è la presenza di due segnali intermedi per collegare i fili in uscita ai debouncer a quelli in ingresso al componente riconoscitore. Inoltre, dato che il bottone CPU\_RESET della board è “zero attivo” è stato definito il segnale reset\_n che risulta essere il negato del segnale di reset in ingresso collegato al bottone stesso.

```

entity riconoscitore_seq_button is
  Port (
    clock_in: in std_logic;
    reset_in: in std_logic;
    M: in std_logic;
    input: in std_logic;
    btn1: in std_logic;
    btn2: in std_logic;
    LED: out std_logic;
    output: out std_logic
  );
end riconoscitore_seq_button;

architecture Structural of riconoscitore_seq_button is

component ButtonDebouncer
  generic (
    CLK_period: integer := 10; -- periodo del clock della board 10 nanosecondi
    btn_noise_time: integer := 650000000 --intervallo di tempo in cui si ha l'oscillazione del bottone
  );
  Port ( RST : in STD_LOGIC;
         CLK : in STD_LOGIC;
         BTN : in STD_LOGIC;
         CLEARED_BTN : out STD_LOGIC);
end component;

component riconoscitore_seq
  port (
    clk: in std_logic;
    mode: in std_logic;
    x: in std_logic;
    read1: in std_logic;
    read2: in std_logic;
    led: out std_logic;
    y: out std_logic
  );
end component;

signal reset_n: std_logic;
signal read_temp1: std_logic;
signal read_temp2: std_logic;
begin

  reset_n <= not reset_in;

  debouncer1: ButtonDebouncer GENERIC MAP(
    CLK_period => 10, -- periodo del clock della board pari a 10ns
    btn_noise_time => 650000000 --intervallo di tempo in cui si ha l'oscillazione del bottone
  )
  port map(
    RST => reset_n,
    CLK => clock_in,
    BTN => btn1,
    CLEARED_BTN => read_temp1
  );

  debouncer2: ButtonDebouncer GENERIC MAP(
    CLK_period => 10, -- periodo del clock della board pari a 10ns
    btn_noise_time => 650000000 --intervallo di tempo in cui si ha l'oscillazione del bottone
  )
  port map(
    RST => reset_n,
    CLK => clock_in,
    BTN => btn2,
    CLEARED_BTN => read_temp2
  );

  riconoscitore: riconoscitore_seq
  port map(
    clk => clock_in,
    mode => M,
    x => input,
    read1 => read_temp1,
    read2 => read_temp2,
    LED => led,
    y => output
  );
end Structural;

```

Oltre ai vari segnali di ingresso, l'entity è caratterizzata da due segnali di uscita:

- LED (per visualizzare M su di un led)
- output (per visualizzare l'uscita si di un led)

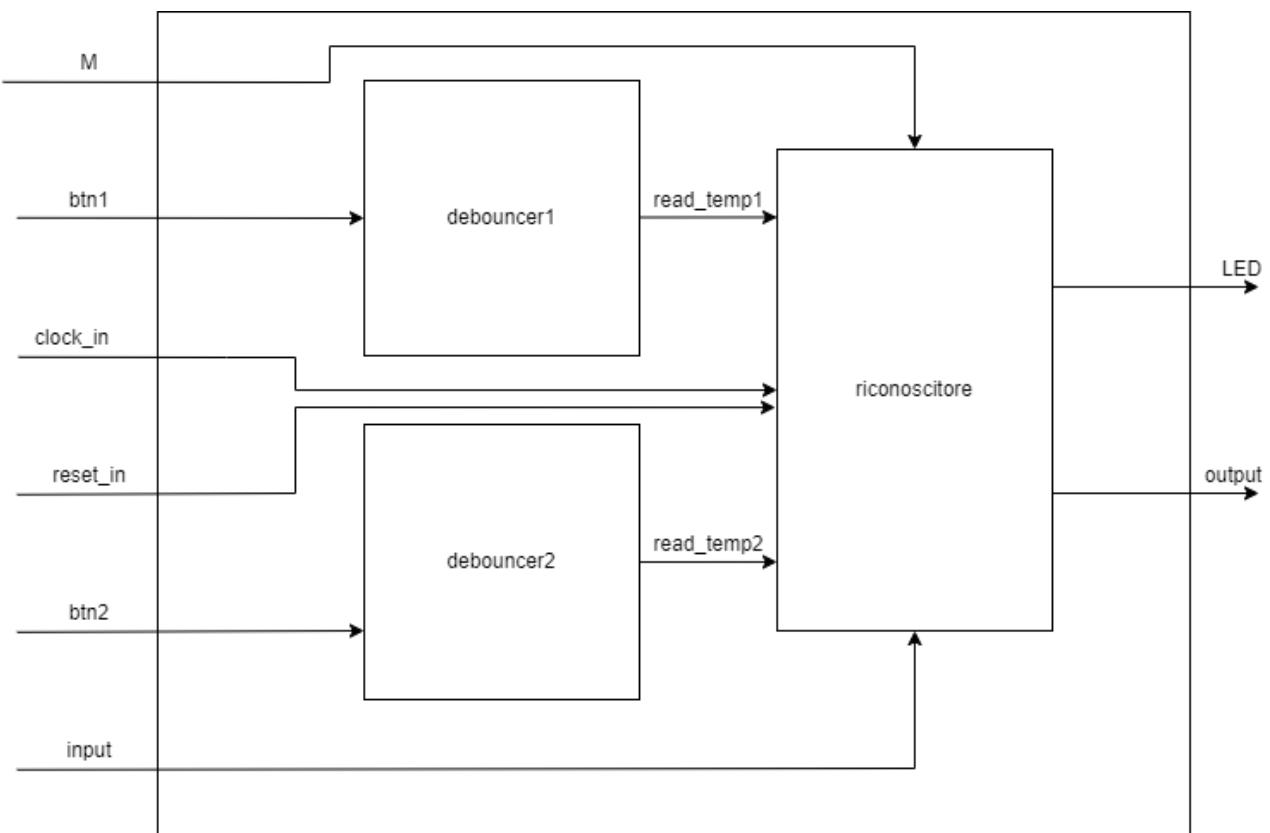


Fig. 3.7 Schema riconoscitore di sequenza on board

# 4. Shift register

## 4.1 Traccia

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

## 4.2 Cenni Teorici

Un registro a scorrimento, nella sua definizione classica, è fatto da un insieme di registri. È una macchina utilizzata per molte periferiche, serve per gli algoritmi che fanno crittografia, poiché shiftano bit e fanno poi operazioni di matematica; è utile perché shiftare a destra o sinistra significa dividere e moltiplicare in binario. Se volessimo realizzare questi registri ad un basso livello di astrazione, vorrebbe dire scegliere il singolo elemento bistabile che li compone, ovvero un flip-flop di tipo T, RS e così via. Possiamo avere vari tipi di shift register:

- serie-serie
- parallelo-parallelo
- serie-serie circolare

È possibile realizzare tale componente utilizzando sia un approccio strutturale che comportamentale.

## 4.3 Soluzione approccio strutturale

Abbiamo implementato uno shift register caratterizzato da 4 bit servendoci di 4 flipflopD e di altrettanti Multiplexer 4:1. In generale il sistema è caratterizzato da 6 ingressi e da 1 uscita: un segnale di temporizzazione (clock\_in), un segnale di reset (reset\_in), un segnale di enable dei flip-flop (en), un segnale che determina il verso dello shift (v), un segnale che definisce il numero di posizioni dello shift (Y) ed i dati seriale di ingresso (input) e di uscita (output).

Oltre che il clock\_in ed il reset\_in che servono rispettivamente per il sincronismo ed il reset, ogni flip-flop è caratterizzato da un ulteriore ingresso che corrisponde all'uscita di uno dei quattro Mux (mux\_out(i)), la quale definisce il dato che entra all'interno del flip-flop stesso. Inoltre, presentano un'uscita che poi risulterà collegata in ingresso a due o più Mux.

In particolare, i multiplexer permettono di scegliere se far shiftare a destra o a sinistra di una o due posizioni a seconda delle opportune selezioni date dai segnali v ed Y. Infatti, ogni Mux presenta 4 ingressi che variano a seconda della posizione in cui si trovano ma che in generale, in base a come è stata codificata il funzionamento rispetto ai valori di selezione sono:

1. Uscita del flip-flop in posizione precedente (Shift a destra di 1)
2. Uscita del flip-flop in posizione successiva (Shift a sinistra di 1)
3. Uscita del flip-flop due posizioni precedenti (Shift a destra di 2)
4. Uscita del flip-flop due posizioni successive (Shift a sinistra di 2)

Abbiamo dei casi particolari:

- Nel caso del Mux più a sinistra, uno degli ingressi è dato dall'input stesso della macchina (shift a destra di 1). Inoltre, dato che non è stato implementato un registro a scorrimento circolare, nel caso in cui si voglia fare uno shift a destra di due posizioni è stato scelto come valore di uscita e quindi di ingresso nel flip-flop lo 0.
- Per quanto riguarda il secondo Mux, anche in questo caso uno degli ingressi è dato dall'input stesso della macchina (shift a destra di 2).

- Anche per il terzo Mux uno degli ingressi è dato dall'input della macchina (shift a sinistra di 2). Questo perché abbiamo scelto che l'input non venga dato solo al flip-flop più a sinistra ma anche a destra.
- Nel caso del Mux più a destra vale il discorso inverso del primo Mux.

I flip-flop si occupano semplicemente di salvare il dato in ingresso ad ogni fronte di clock e di mandarlo in uscita. Un'osservazione che bisogna fare è che tutti i flip-flop ricevono lo stesso clock e devono lavorare in maniera sincrona perché tutto funzioni come previsto.

### 4.3.1 Schematici

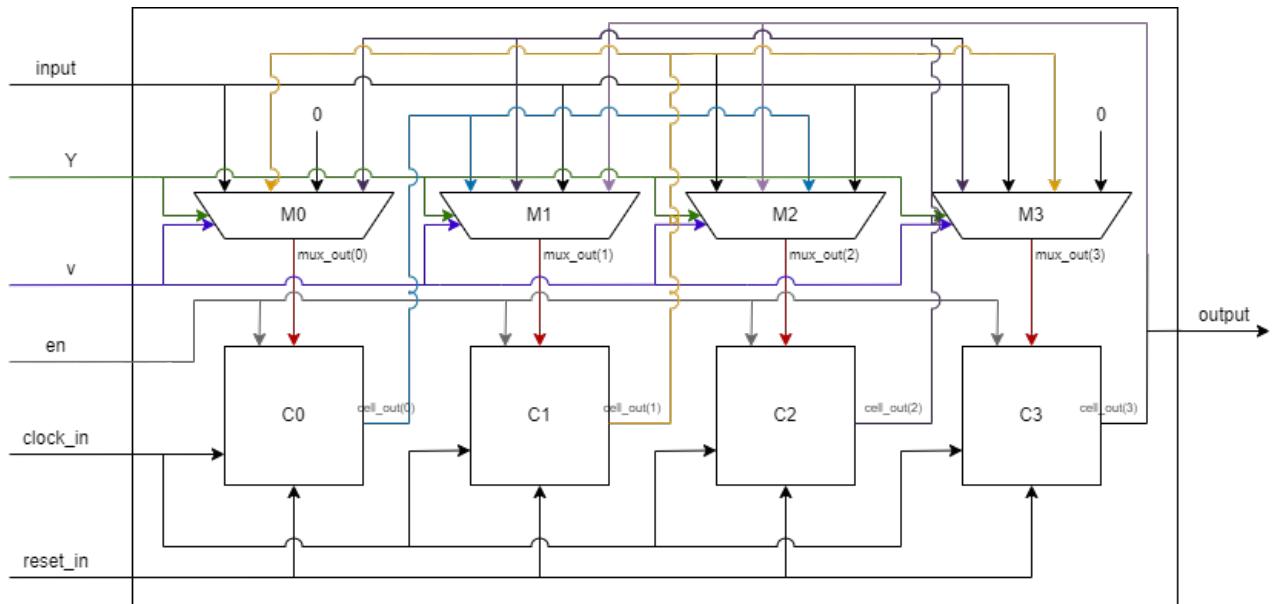


Fig. 4.1 Schema shift register strutturale

### 4.3.2 Codice

Innanzitutto, abbiamo implementato il Multiplexer 4:1 a livello comportamentale. Esso presenta due ingressi di tipo `std_logic_vector` e si occupa di dare in uscita uno dei quattro fili in ingresso in base ad opportuni bit di selezione (Figura 4.2).

Successivamente abbiamo implementato i flip-flop a livello comportamentale. Essi funzionano sul fronte di salita del clock con un reset sincrono. Inoltre, è stato utilizzato un segnale `last_enable` al fine di risolvere i problemi di sincronizzazione tra il clock e l'enable e quindi evitare che ci siano più shift nel caso in cui quest'ultimo resti alzato per più colpi di clock. Quindi sul fronte di salita di quest'ultimo e del clock, i flip-flop salvano il dato in ingresso (Figura 4.3).

```

entity mux4_1 is
Port (
  a: in std_logic_vector(0 to 3);
  c: in std_logic_vector(0 to 1);
  y: out std_logic
);
end mux4_1;

architecture Behavioral of mux4_1 is
begin
  with c select
    y <= a(0) when "00",
                a(1) when "01",
                a(2) when "10",
                a(3) when "11",
                '-' when others;
end Behavioral;

```

Fig. 4.2 Codice Multiplexer 4:1

```

entity cell is
  Port(
    ing: in std_logic;
    usc: out std_logic;
    clk: in std_logic;
    enable: in std_logic;
    rst: in std_logic
  );
end cell;

architecture Behavioral of cell is
signal lastEnable : std_logic := '0';
begin
process(clk)
begin
  if rising_edge(clk) then
    if (rst = '1') then
      usc <= '0';
    end if;
    if (enable = '1' and lastEnable = '0') then
      usc <= ing;
    end if;
    lastEnable <= enable;
  end if;
end process;

end Behavioral;

```

Fig. 4.3 Codice Flip-Flop

Compreso il funzionamento delle macchine che compongono il registro a scorrimento, il codice per implementarlo diventa semplice. Si dovranno istanziare n flip-flop dove n è il numero di bit e n Multiplexer 4:1 che permettono di scegliere il dato i ingresso. Verranno inoltre usati due segnali interni che modellano i cavi di collegamento tra i multiplexer e i flip-flop.

```

entity shift_register is
  Port (
    input: in std_logic;
    output: out std_logic_vector(0 to 3);
    clock_in: in std_logic;
    reset_in: in std_logic;
    Y: in std_logic;      --(1,2)
    v: in std_logic;      --(S,D)
    en: in std_logic
  );
end shift_register;

architecture Structural of shift_register is

component mux4_1
  Port (
    a: in std_logic_vector(0 to 3);
    c: in std_logic_vector(0 to 1);
    y: out std_logic
  );
end component;

component cell
  Port(
    ing: in std_logic;
    usc: out std_logic;
    clk: in std_logic;
    enable: in std_logic;
    rst: in std_logic
  );
end component;

signal mux_out: std_logic_vector(0 to 3);
signal cell_out: std_logic_vector(0 to 3);
begin
  output(0 to 3) <= cell_out(0 to 3);
  m0: mux4_1
  port map(
    a(0) => input, --00 SHIFT 1 ->
    a(1) => cell_out(1), --01 SHIFT 1 <-
    a(2) => '0', -- 10 SHIFT ->> (non ci sta cella)
    a(3) => cell_out(2), --11 SHIFT <<-
    c(0) => Y,
    c(1) => v,
    y => mux_out(0)
  );
  m1: mux4_1
  port map(
    a(0) => cell_out(0), --00 SHIFT 1 ->
    a(1) => cell_out(2), --01 SHIFT 1 <-
    a(2) => input, -- 10 SHIFT ->>
    a(3) => cell_out(3), --11 SHIFT <<-
    c(0) => Y,
    c(1) => v,
    y => mux_out(1)
  );
  m2: mux4_1
  port map(
    a(0) => cell_out(1), --00 SHIFT 1 ->
    a(1) => cell_out(3), --01 SHIFT 1 <-
    a(2) => cell_out(0), -- 10 SHIFT ->>
    a(3) => input, --11 SHIFT <<-
    c(0) => Y,
    c(1) => v,
    y => mux_out(2)
  );
  m3: mux4_1
  port map(
    a(0) => cell_out(2), --00 SHIFT 1 ->
    a(1) => input, --01 SHIFT 1 <-
    a(2) => cell_out(1), -- 10 SHIFT ->>
    a(3) => '0', --11 SHIFT <<-
    c(0) => Y,
    c(1) => v,
    y => mux_out(3)
  );

```

```

c0: cell
port map(
    ing => mux_out(0),
    usc =>cell_out(0),
    clk => clock_in,
    enable => en,
    rst => reset_in
);

c1: cell
port map(
    ing => mux_out(1),
    usc =>cell_out(1),
    clk => clock_in,
    enable => en,
    rst => reset_in
);
c2: cell
port map(
    ing => mux_out(2),
    usc => cell_out(2),
    clk => clock_in,
    enable => en,
    rst => reset_in
);
c3: cell
port map(
    ing => mux_out(3),
    usc => cell_out(3),
    clk => clock_in,
    enable => en,
    rst => reset_in
);
end Structural;

```

## 4.4 Simulazione approccio strutturale

Per effettuare la simulazione è stato realizzato il testbench sottostante. Abbiamo fissato il periodo del clock con rispettivo process che definisce il comportamento del segnale stesso. Inoltre, sono stati determinati tutti i segnali responsabili della simulazione. Dopo 200 ns, fissiamo il verso e il numero di posizioni dello shift. Successivamente, ogni 10 ns, settiamo i valori del dato in ingresso e alziamo ed abbassiamo il segnale di enable per abilitare i flip-flop.

```

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

architecture Behavioral of TB_SHIFT is
component shift_register is
    Port (
        input: in std_logic;
        output: out std_logic_vector(0 to 3);
        clock_in: in std_logic;
        reset_in: in std_logic;
        Y: in std_logic;      --{1,2}
        v: in std_logic;      --{S,D}
        en: in std_logic
    );
end component;

signal INPUT : std_logic := '0';
signal CLK: std_logic;
signal K : std_logic := '0'; --Y
signal V: std_logic := '0';
signal E : std_logic := '0';
signal R: std_logic := '0';
signal O: std_logic_vector(0 to 3) := "0000";
constant CLK_period : time := 10 ns;

```

```

begin
  uut: shift_register
  port map(
    input=>INPUT,
    output(0 to 3) => O,
    clock_in => CLK,
    reset_in => R,
    Y => K,
    v => V,
    en => E
  );
  CLK_process :process
  begin
    CLK <= '0';
    wait for CLK_period/2;
    CLK <= '1';
    wait for CLK_period/2;
  end process;

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

  wait for CLK_period*10;

  -- insert stimulus here
  K <= '0'; --Y
  V <= '0';
  E <= '0';

  wait for 10 ns;
  INPUT <= '1';

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

  wait for 10 ns;
  INPUT <= '1';
  wait for 10 ns;
  E <= '1';
  wait for 10 ns;
  E <= '0';

  wait for 10 ns;
  INPUT <= '1';
  wait for 10 ns;
  E <= '1';
  wait for 10 ns;
  E <= '0';

  wait for 10 ns;
  INPUT <= '1';
  wait for 10 ns;
  E <= '1';
  wait for 10 ns;
  E <= '0';

  wait for 10 ns;
  INPUT <= '0';
  wait for 10 ns;
  E <= '1';
  wait for 10 ns;
  E <= '0';

  wait;
end process;

```

end Behavioral;

Come è possibile vedere dalla figura 4.4, la macchina, dati gli ingressi (1-1-1-1-0), e in corrispondenza del fronte di salita del clock e del segnale di enable, ha effettuato correttamente cinque volte lo shift di una posizione a destra.

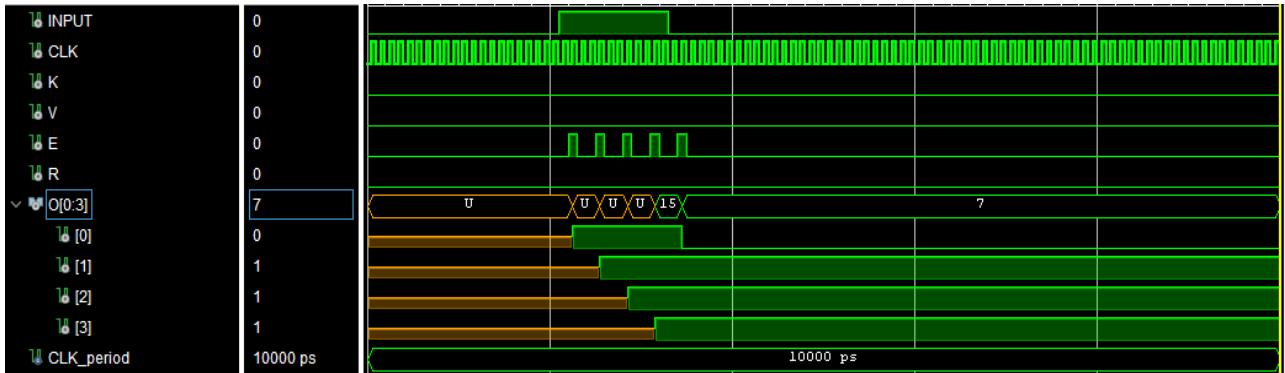


Fig. 4.4 Simulazione shift register (shift 1 posizione a destra)

Di seguito mostriamo la simulazione dello shift a sinistra di due posizioni della stessa sequenza di ingressi. Per far ciò dobbiamo porre a 1 entrambi i bit di selezione dei Multiplexer. Risulta corretto lo shift anche in questo caso.

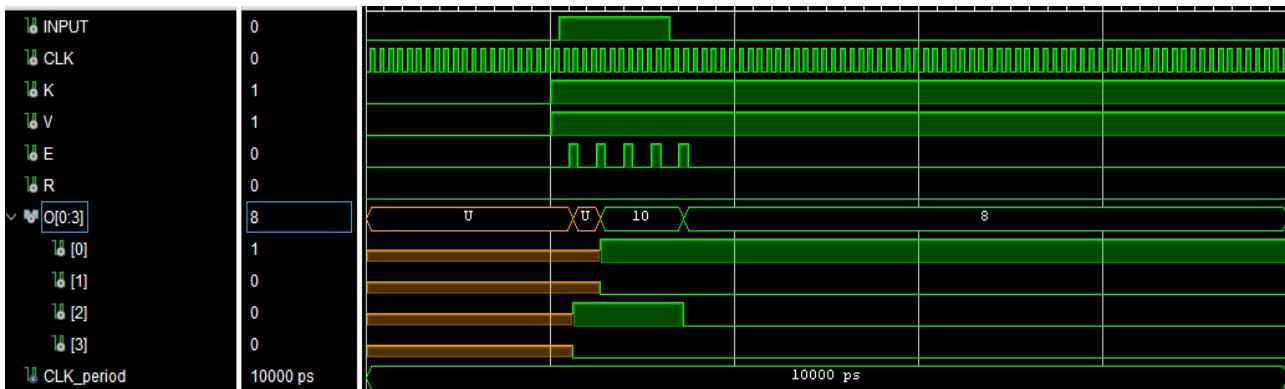


Fig. 4.5 Simulazione shift register (shift 2 posizioni a sinistra)

## 4.5 Soluzione approccio comportamentale

A differenza dell'approccio strutturale, ora non abbiamo bisogno né di Multiplexer, né di flip-flop, in quanto implementiamo lo stesso shift register, con lo stesso comportamento di quello precedente, a livello comportamentale, tramite l'utilizzo di un process con annessi costrutti *if* e di un *vettore temporaneo*. Il sistema presenta, allo stesso modo di quello precedentemente sviluppato: un segnale di temporizzazione (clock\_in), un segnale di reset (reset\_in), un segnale di enable dei flip-flop (en), un segnale che determina il verso dello shift (v), un segnale che definisce il numero di posizioni dello shift (Y) ed i dati seriali di ingresso (input) e di uscita (output).

### 4.5.1 Schematici



Fig. 4.6 Schema shift register

### 4.5.2 Codice

Come già accennato in precedenza, il sistema è stato sviluppato utilizzando un vettore temporaneo che mantenesse i valori degli ipotetici flip-flop istante per istante. Lo scorrimento, infatti, può essere espresso dicendo che l'elemento  $i$ -esimo del vettore assume il valore dell'elemento  $(i-1)$ -esimo, e ciò per tutti gli elementi ad eccezione del primo, cui deve essere assegnato esplicitamente il valore del segnale *input*, e dell'ultimo che deve essere assegnato esplicitamente al segnale *output*. Questo nel caso in cui sia  $Y$  che  $v$  siano uguali a 0. In tutti gli altri casi, discriminati dai vari costrutti if, si presentano gli stessi problemi e gli stessi casi particolari descritti in precedenza. (( $Y=0, v=1$  shift di 1 a  $S_x$ ), ( $Y=1, v=0$  shift di 2 a  $D_x$ ), ( $Y=1, v=1$  shift di 2 a  $S_x$ )).

```

entity shift_register_beha is
  Port (
    input: in std_logic;
    output: out std_logic_vector(0 to 3);
    clock_in: in std_logic;
    reset_in: in std_logic;
    Y: in std_logic;      --{1,2}
    v: in std_logic;      --{S,D}
    en: in std_logic
  );
end shift_register_beha;

architecture Behavioral of shift_register_beha is
signal lastEn : std_logic := '0';
signal cell_out : std_logic_vector(0 to 3) := "0000";

begin
process(clock_in)
begin

  if rising_edge(clock_in) then
    if (reset_in = '1') then
      output(0 to 3) <= "0000";
    elsif(en='1' and lastEn='0') then
      if(Y='0' and v='0') then
        cell_out(0) <= input;
        cell_out(1) <= cell_out(0);
        cell_out(2) <= cell_out(1);
        cell_out(3) <= cell_out(2);
      end if;
    else
      cell_out(0) <= cell_out(1);
      cell_out(1) <= cell_out(2);
      cell_out(2) <= cell_out(3);
      cell_out(3) <= cell_out(0);
    end if;
  end if;
  lastEn <= en;
end process;

```

```

        elsif(Y='0' and v='1') then
            cell_out(3) <= input;
            cell_out(2) <= cell_out(3);
            cell_out(1) <= cell_out(2);
            cell_out(0) <= cell_out(1);
        elsif(Y='1' and v='0') then
            cell_out(0) <= '0';
            cell_out(1) <= input;
            cell_out(2) <= cell_out(0);
            cell_out(3) <= cell_out(1);
        else --Y='1' and v='1'
            cell_out(3) <= '0';
            cell_out(2) <= input;
            cell_out(1) <= cell_out(3);
            cell_out(0) <= cell_out(2);
        end if;
    end if;
    lastEn <= en;
    output(0 to 3) <= cell_out(0 to 3);
) end if;

) end process;
) end Behavioral;

```

## 4.6 Simulazione approccio comportamentale

Per effettuare la simulazione è stato realizzato il testbench allo stesso modo di quello precedente dato che sia il comportamento della macchina che i segnali in ingresso e uscita sono gli stessi.. Abbiamo fissato il periodo del clock con rispettivo process che definisce il comportamento del segnale stesso. Inoltre, sono stati determinati tutti i segnali responsabili della simulazione. Dopo 200 ns, fissiamo il verso e il numero di posizioni dello shift. Successivamente, ogni 10 ns, settiamo i valori del dato in ingresso e alziamo ed abbassiamo il segnale di enable per abilitare i flip-flop.

```

entity TB_shift beha is
-- Port ();
end TB_shift beha;

architecture Behavioral of TB_shift beha is

component shift_register beha is
    Port (
        input: in std_logic;
        output: out std_logic_vector(0 to 3);
        clock_in: in std_logic;
        reset_in: in std_logic;
        Y: in std_logic;      --(1,2)
        v: in std_logic;      --(S,D)
        en: in std_logic
    );
end component;

signal INPUT : std_logic := '0';
signal CLK: std_logic;
signal K : std_logic := '0'; --Y
signal V: std_logic := '0';
signal E : std_logic := '0';
signal R: std_logic := '0';
signal O: std_logic_vector(0 to 3) := "0000";

constant CLK_period : time := 10 ns;

begin
uut: shift_register beha
port map(
    input=>INPUT,
    output(0 to 3) => O,
    clock_in => CLK,
    reset_in => R,
    Y => K,
    v => V,
    en => E
);

```

```

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

stim_proc: process
begin
    wait for 100 ns;
    wait for CLK_period*10;

    K <= '1'; --Y
    V <= '1';
    E <= '0';
    wait for 10 ns;
    INPUT <= '0';
    wait for 10 ns;
    E <= '1';
    wait for 10 ns;

    E <= '0';
    wait for 10 ns;
    INPUT <= '0';
    wait for 10 ns;
    E <= '1';
    wait for 10 ns;

    E <= '0';
    wait for 10 ns;
    INPUT <= '1';
    wait for 10 ns;
    E <= '1';
    wait for 10 ns;

    E <= '0';
    wait for 10 ns;
    INPUT <= '1';
    wait for 10 ns;
    E <= '1';

    wait;
end process;
end Behavioral;

```

Come è possibile vedere dalla figura 4.7, la macchina, dati gli ingressi (0-0-1-1), e in corrispondenza del fronte di salita del clock e del segnale di enable, ha effettuato correttamente quattro volte lo shift di due posizioni a destra. Per far ciò abbiamo dovuto porre, seguendo come abbiamo codificato le varie modalità di funzionamento all'interno delle condizioni dei costrutti if, il segnale Y a 1 ed il segnale v a 0.

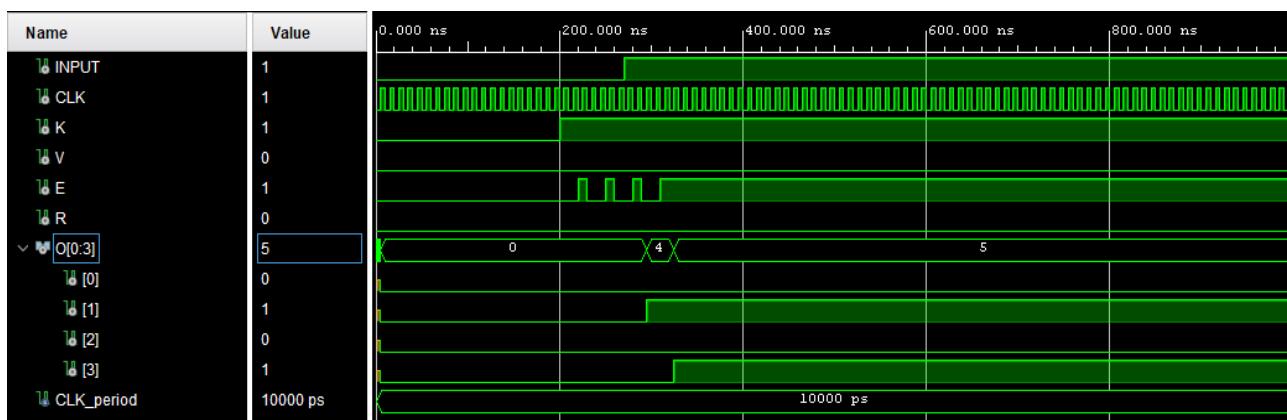


Fig. 4.7 Simulazione shift register (shift 2 posizioni a destra)

Di seguito mostriamo la simulazione dello shift a sinistra di una posizione della stessa sequenza di ingressi. Per far ciò dobbiamo porre a 0 il segnale Y e a 1 il segnale v. Risulta corretto lo shift anche in questo caso.

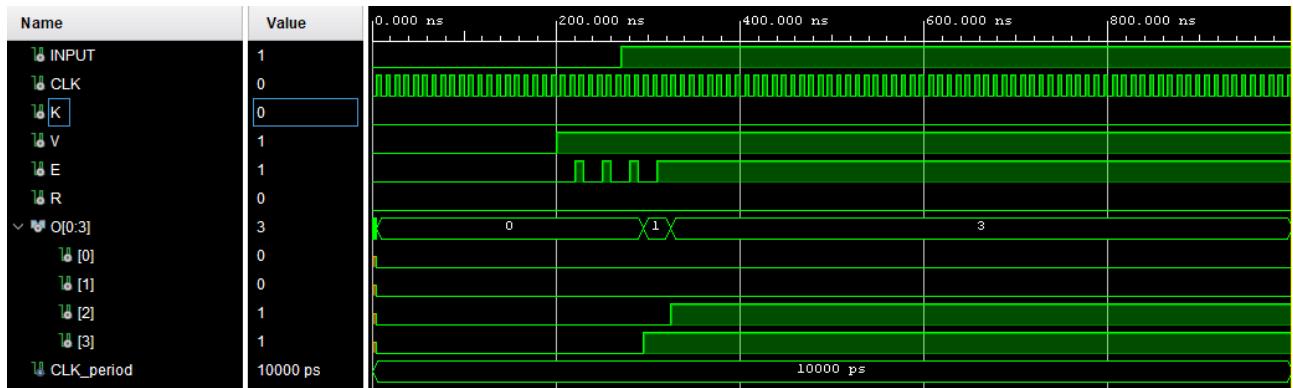


Fig. 4.8 Simulazione shift register (shift 1 posizione a sinistra)

# Capitolo 5

## 5. Cronometro

### 5.1 Traccia 1

Progettare, implementare in VHDL e testare mediante simulazione un cronometro, in grado di scandire secondi, minuti e ore a partire da una base dei tempi prefissata (es. si consideri il clock a disposizione sulla board). Il progetto deve prevedere la possibilità di inizializzare il cronometro con un valore iniziale, sempre espresso in termini di ore, minuti e secondi, mediante un opportuno ingresso di set, e deve prevedere un ingresso di reset per azzerare il tempo.

Il componente deve essere realizzato utilizzando un approccio strutturale, collegando opportunamente dei contatori secondo uno schema a scelta.

#### 5.1.1 Soluzione

Per realizzare il cronometro abbiamo utilizzato 3 contatori collegati in cascata, il primo conta i secondi, da 0 a 59, il secondo i minuti da 0 a 59, infine il terzo le ore, da 0 a 23. Ciascun contatore è stato realizzato tramite un approccio behavioral, utilizzando la clausola generic in modo da poter istanziare diverse tipologie di contatori sfruttando lo stesso schema, inoltre è anche possibile precaricare l'orario di inizio. Per realizzare il clock di riferimento adeguato abbiamo utilizzato una base dei tempi, abbiamo diviso la frequenza del clock della scheda, pari a 100 MHz nel caso della Nexys-A7-100T, così che diventasse pari a 1 Hz, in modo da contare ogni secondo, i contatori successivi ricevono il clock dall'uscita div del contatore precedente e non dalla base dei tempi. Infine per visualizzare l'ora abbiamo adoperato il display a 7 segmenti.

##### 5.1.1.1 Schematici

Il **contatore** (figura 5.1.1) è stato realizzato secondo l'approccio behavioral, utilizzando la clausola generic così da poter istanziare contatori di diverso tipo a partire dallo stesso schema. I due parametri generic, sono N ed L, il primo indica il modulo di conteggio, il secondo il numero di bit su cui è codificato N. In ingresso il blocco riceve il clock, che funge da ingresso di conteggio, il segnale di reset, il segnale di enSet che serve per impostare il conteggio al valore contenuto nella stringa set di L bit in ingresso al contatore. In uscita abbiamo il segnale y, che si alza quando il contatore arriva a contare N-1, poiché i contatori lavorano sul fronte di discesa, e servirà come ingresso di conteggio al contatore successivo, infine in uscita è presente anche la stringa number di L bit, contenente il valore di conteggio.

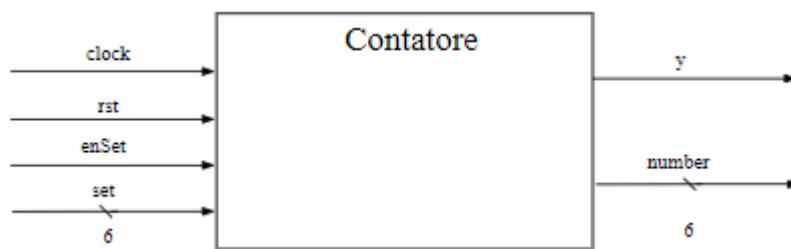


Figura 5.1.1: Contatore

Il **cronometro** è stato realizzato tramite un approccio strutturale, ovvero componendo i contatori in parallelo, come in mostrato in figura 5.1.2. In ingressoabbiamo il segnale di clock con frequenza 1 Hz, che viene mandato al contatore dei secondi, il segnale di reset che viene mandato a tutti i contatori per il reset. I segnali di ingresso SET(6 bit) sel\_demux(2 bit) e enable\_set sono utilizzati per settare

il valore del contatore dei secondi, minuti oppure ore, in particolare SET contiene la stringa di bit da impostare come valore di conteggio(0-59 oppure 0-23) e viene mandato a tutti e tre i contatori, il segnale enable\_set quando vale 1 indica al contatore a cui è collegato di resettare il suo valore ed impostare il valore di conteggio al contenuto di SET, per poter impostare separatamente il valore dei secondi, minuti oppure delle ore il segnale enable\_set è collegato in ingresso ad un demultiplexer 1-4 mentre le uscite sono collegate rispettivamente al contatore dei secondi, minuti ed ore, il segnale di sel\_demux è utilizzato per pilotare il demultiplexer così da decidere a quale dei 3 contatori inviare il segnale di enable\_set. Le stringhe in uscita dal cronometro hours, minute, second sono i valori di conteggio dei singoli contatori. I contatori commutano sul fronte di discesa, sono collegati in parallelo per evitare il problema della propagazione presente nel collegamento in serie, in particolare il contatore dei secondi commuta ogni volta volta che arriva il clock, il contatore dei minuti invece quando arriva il clock e l'uscita y del contatore dei secondi è alta, infine il contatore delle ore commuta quando arriva il clock e sono alti i segnali y sia del contatore dei secondi sia dei minuti. Per realizzare questo schema mettiamo l'uscita del contatore dei secondi in AND con il clock, in questo modo, se sono attivi sul fronte di discesa, quando il clock si abbassa l'uscita dei secondi si alza e il contatore dei minuti commuta solo al colpo di clock successivo, quando trova alto sia l'uscita dei secondi sia il clock. All'ingresso del terzo componente invece mettiamo in AND le uscite dei secondi, delle ore e il segnale di clock, in questo modo commuta solo quando le due uscite sono alte ed arriva il clock.

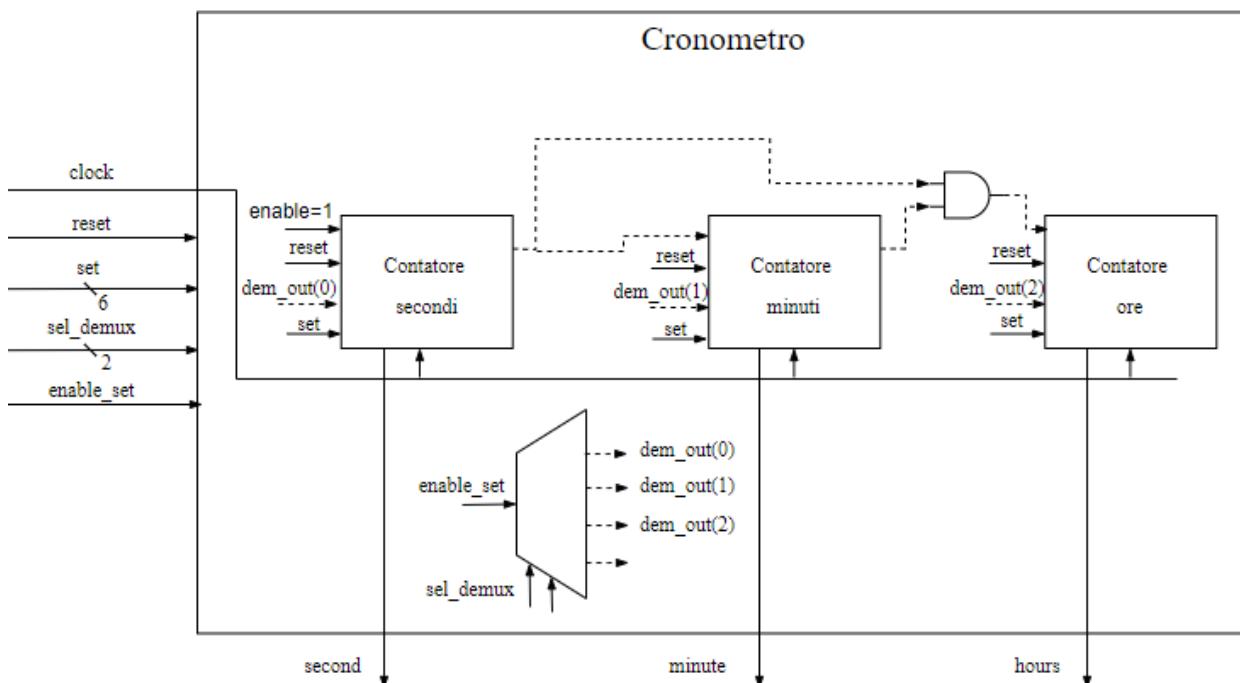


Figura 5.1.2: Schema cronometro

### 5.1.2 Codice

Di seguito è riportato il codice con cui è stato realizzato il **contatore** behavioral

## Contatore

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
entity contatore is
generic(
    N: integer := 2;
    L: integer := 1
);
Port (
    clk: in std_logic;
    rst: in std_logic;
    set: in std_logic_vector(0 to L-1);
    enSet: in std_logic;
    y: out std_logic;
    number: out std_logic_vector(0 to L-1);
    enable: in std_logic
);
end contatore;
architecture Behavioral of contatore is

begin
process(clk,rst,enSet)
variable count: integer := 0;
variable setted: integer := 0;
begin
    if(rst = '1') then
        count := 0;
    elsif(enSet = '1') then
        count := TO_INTEGER(unsigned(set));
        setted := 1;
    elsif falling_edge (clk) then
        if(enable='1') then
            y <= '0';
            if(setted = 1 and count=N-1) then
                y <= '1';
                setted := 0;
            end if;
            if(count = N-2) then
                y <= '1';
            end if;
            if(count = N-1) then
                count := 0;
            else
                count := count +1;
            end if;
        end if;
    end if;
    number <= std_logic_vector(TO_UNSIGNED(count,L));
end process;
end Behavioral;

```

L'entity rispecchia quanto visto nello schema in figura 5.1.1, in ingresso abbiamo il segnale di clock clk, il reset rst, il vettore set, il segnale enSet per impostare il conteggio al valore contenuto in set, mentre in uscita abbiamo y ed il valore di conteggio number. Sono adoperate 2 variabili count che serve per memorizzare il valore di conteggio, setted che è un flag usato per gestire il caso in cui sia settato un valore di conteggio tramite enSet\set. L'architecture è realizzata da un process, nella sensitivity list è presente clk, rst, enSet così da rendere il reset e il set di un valore di conteggio asincroni rispetto al clock. Le prime due condizioni servono per gestire il reset e set del contatore, mentre il conteggio vero e proprio avviene nel terzo caso (elsif falling\_edge(clk) then), il segnale y è sempre settato a '0', per poi essere modificato dalle istruzioni successive se opportuno. La condizione if(count = N-2) serve ad alzare il segnale y $\leq$ '1' sul fronte di discesa del clock quando il conteggio vale N-2 e quindi sta per scattare a N-1, ad esempio nel caso del contatore dei secondi sul fronte di discesa del clock quando il conteggio vale 58 e sta per scattare il 59. La condizione if(count = N-1) serve per settare a 0 il contatore nel caso che la condizione sia vera, oppure nel caso sia falsa per incrementare di 1 il valore di conteggio. E' necessario gestire il caso in cui il valore settato tramite enSet\set sia proprio N-1, poiché in questa condizione il segnale di y non è alzato direttamente dalla condizione if(count = N-2) descritta prima, si controlla quindi la condizione if(setted = 1 and count = N-1) se risulta vera setted viene messa a 0 ed y ad 1.

## Cronometro

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity cronometro is
  Port (
    clock: in std_logic;
    reset: in std_logic;
    SET: in std_logic_vector(0 to 5);
    sel_demux: in std_logic_vector(1 downto 0);
    enable_set: in std_logic;
    hours: out std_logic_vector(0 to 4);
    minute: out std_logic_vector(0 to 5);
    second: out std_logic_vector(0 to 5)
  );
end cronometro;

architecture Structural of cronometro is
component contatore is
  generic(
    N : integer := 24;
    L : integer := 5
  );
  port(
    clk: in std_logic;
    rst: in std_logic;
    set: in std_logic_vector(0 to L-1);
    enSet: in std_logic;
    y: out std_logic;
    number: out std_logic_vector(0 to L-1);
  );
end component;
begin
  U1: contatore
    generic map(N => 24, L => 5)
    port map(clk, rst, set, enSet, y, number);
end;

```

```

        enable: in std_logic
    );
end component;

component demux_1_4 is
port(
    b0: in std_logic;
    s: in std_logic_vector(1 downto 0);
    y: out std_logic_vector(0 to 3)
);
end component;

signal dem_out: std_logic_vector(0 to 3);
signal y_out: std_logic_vector(0 to 2);
signal t: std_logic_vector(0 to 1);
signal selezione: std_logic_vector(1 downto 0) := "10";
signal abilitazione: std_logic := '1';

begin

demux: demux_1_4
port map(
    b0 => enable_set,
    s => sel_demux,
    y => dem_out
);

secondi: contatore
generic map(
    N => 60,
    L => 6
)
port map(
    clk => clock,
    rst => reset,
    set => SET,
    enSet => dem_out(0),
    y => y_out(0),
    number => second,
    enable => '1'
);

t(0) <= y_out(0);

minuti: contatore
generic map(
    N => 60,
    L => 6
)
port map(
    clk => clock,
    rst => reset,

```

```

        set => SET,
        enSet => dem_out(1),
        y => y_out(1),
        number => minute,
        enable => t(0)
    );
t(1) <= y_out(0) and y_out(1);

ore: contatore
generic map(
    N => 24,
    L => 5
)
port map(
    clk => clock,
    rst => reset,
    set => SET(0 to 4),
    enSet => dem_out(2),
    y => y_out(2),
    number => hours,
    enable => t(1)
);
end Structural;

```

Il **cronometro** invece è realizzato mediante approccio strutturale, sfruttando i componenti contatore e demultiplexer. L'entity rispecchia lo schema illustrato in figura 5.1.2. Vengono istanziati 3 contatori dallo stesso componente contatore, sfruttando l'utilizzo dei generics e modificando i parametri N ed L, per i secondi ed i minuti abbiamo N = 60, L = 6, mentre per le ore N=24, L = 5. Per la realizzazione del collegamento in parallelo dei contatori sono utilizzati due vettori di segnali (std\_logic\_vector), y\_out(0 to 2) e t(0 to 1). Le uscite y dei 3 contatori sono collegati al vettore di appoggio y\_out, i secondi a y\_out(0), minuti a y\_out(1) e le ore ad y\_out(2). Il segnale t è utilizzato per realizzare i collegamenti intermedi tra i contatori, in particolare le porte logiche AND sono realizzate con approccio dataflow, t(0) è il risultato dell'operazione di and tra y\_out(0)(uscita dei secondi) ed il clock, viene messo quindi in ingresso al contatore dei minuti, in t(1) invece è presente il risultato dell'and tra y\_out(0), y\_out(1) ed il clock, e viene messo in ingresso al contatore delle ore, mentre in ingresso al contatore dei secondi c'è il clock. I segnali di uscita hours, minute, second sono usati per immettere in uscita il valore di conteggio dei rispettivi contatori. Infine per gestire il set del valore di conteggio ad un determinato valore ad ogni contatore viene dato in ingresso il segnale SET (std\_logic\_vector), mentre come segnale di abilitazione del set ad ogni contatore è collegata un'uscita differente del demultiplexer, in questo modo è selezionare il contatore a cui dare il segnale di set.

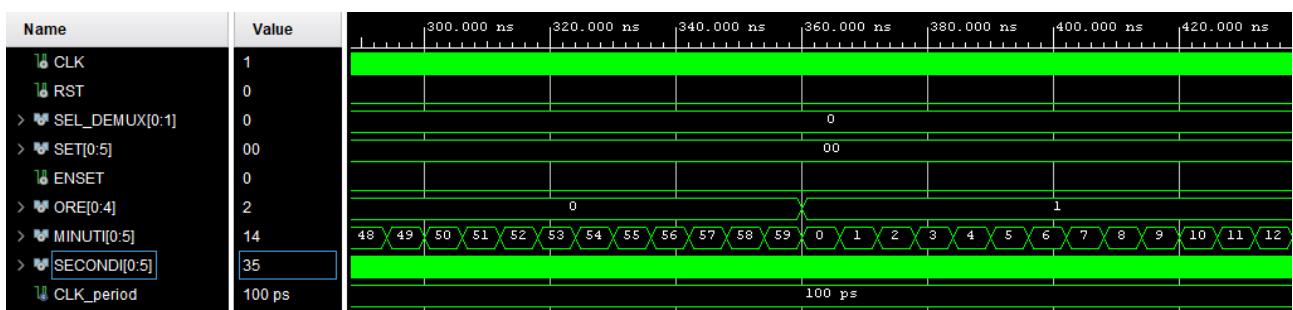


Figura 5.1.3: Simulazione cronometro

## 5.2 Traccia 2

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

### 5.2.1 Soluzione

Viene sfruttata l'implementazione del cronometro descritta nella sezione 5.1, in particolare è realizzato un nuovo modulo con approccio strutturale cronometroOnDisplay, il quale utilizza il componente cronometro, div per generare la base dei tempi, il componente display\_seven\_segments fornito per utilizzare i display 7 segmenti, ed il componente conversion utilizzato per convertire il formato delle stringhe di bit in uscita dal cronometro in modo da adattarle alla visualizzazione sui display 7 segment.

Per realizzare la base dei tempi, utilizziamo un divisore di frequenza, realizzato mediante il componente div, il clock fornito dalla scheda è pari a 100 MHz, il divisore lo rende pari a 1 Hz, adatto quindi ad essere messo in ingresso al contatore dei secondi. Il divisore è realizzato con approccio behavioral, in particolare si definisce un valore costante count\_max\_value definito secondo l'equazione

$$count\_max\_value = \frac{clock\_frequency\_in}{clock\_frequency\_out} - 1$$

Definendo la frequenza desiderata in uscita, si calcola in questo modo il valore di conteggio a cui deve arrivare il contatore, quando raggiungere tale valore il valore di conteggio viene azzerato e si alza il segnale clockfx, che rappresenta il clock generato alla frequenza desiderata.

Per visualizzare i secondi, minuti ed ore sui display 7 segment sono utilizzati 3 istanze del componente converter, dato che l'output del componente cronometro sono due stringhe da 6 bit (secondi e minuti) ed una stringa da 5 bit (ore), mentre per visualizzare le cifre in modo corretto è necessario codificare ogni cifra da mostrare su 4 bit. Sono istanziati 3 conversion uno per i secondi, uno per i minuti ed uno per le ore. Il blocco prende in ingresso una stringa da 6 bit (nel caso delle ore l'ultimo bit è messo sempre a 0) e produce in output una stringa da 8 bit su cui sono codificate le due cifre da mostrare. Ad esempio nel caso in cui l'output del cronometro per la stringa dei secondi è “010111” = 23 il blocco conversion produce in output la stringa “00100011” che codifica separatamente le cifre

2 e 3 su 4 bit per cifra. Infine gli output dei 3 blocchi conversion sono passati al componente display seven segment come primi 24 bit in modo che vengano visualizzati sui primi 6 display 7 segment, mentre gli ultimi 8 bit richiesti (il componente display seven segment richiede 32 bit in ingresso, 4 per ogni cifra) sono posti a 0.

Per l'acquisizione dei valori di inizializzazione abbiamo utilizzato switch e bottoni, in particolare è presente un bottone per il reset ed uno per il set, essi sono collegati direttamente al componente cronometro che gestisce come descritto alla sezione 5.1 i segnali per il reset e set, inoltre è utilizzato un debouncer per ogni bottone. Sono utilizzati 6 switch per settare il valore di inizializzazione dei secondi, minuti oppure delle ore, inoltre tramite 2 switch ulteriori è possibile selezionare se il valore di set deve essere impostato per i secondi, i minuti oppure per le ore.

#### 5.2.1.1 Schematici

Per il componente cronometro on display (figura 5.1.4), abbiamo in input il segnale clock\_in, dato direttamente dalla scheda, pari a 100 MHz, reset\_n dato dal pulsante reset della scheda utilizzato per

resetare il display ed i debouncer dei bottoni. Il segnale SET\_IN (std\_logic\_vector) di 6 bit utilizzato per inizializzare il valore di conteggio impostato tramite 6 switch, SEL\_DEMUX assegnato tramite 2 switch, per selezionare a quale contatore settare il valore contenuto in SET\_IN, BTN\_ENABLE\_SET e BTN\_RESET (diverso dal segnale reset\_n) sono i bottoni utilizzati per il set e reset del cronometro.

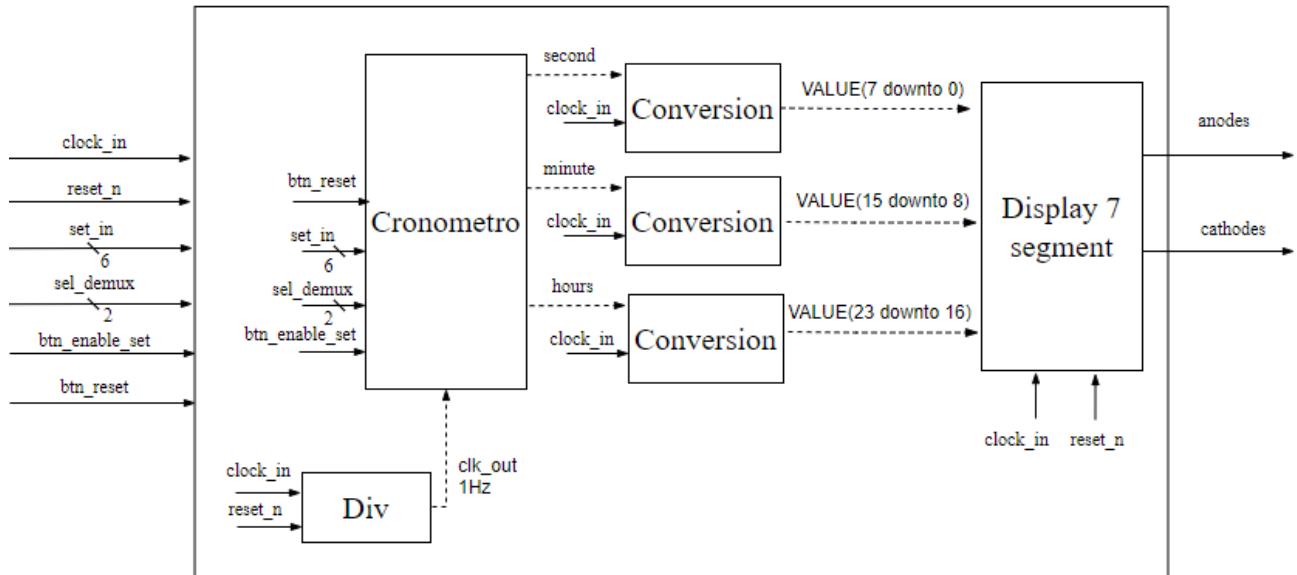


Figura 5.2.1: Schema Cronometro su board

## 5.2.2 Codice Conversion

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity conversion is
Port (
    vIn: in std_logic_vector(0 to 5);
    vOut: out std_logic_vector(7 downto 0);
    clk: in std_logic
);
end conversion;

architecture Behavioral of conversion is
begin
process(clk)
begin
if rising_edge(clk) then
    case to_integer(unsigned(vIn)) is
    when 0|10|20|30|40|50 => vOut(3 downto 0) <= "0000";
    when 1|11|21|31|41|51 => vOut(3 downto 0) <= "0001";
    when 2|12|22|32|42|52 => vOut(3 downto 0) <= "0010";
    when 3|13|23|33|43|53 => vOut(3 downto 0) <= "0011";
    when 4|14|24|34|44|54 => vOut(3 downto 0) <= "0100";
    end case;
end if;
end process;
end;

```

```

when 5|15|25|35|45|55 => vOut(3 downto 0) <= "0101";
when 6|16|26|36|46|56 => vOut(3 downto 0) <= "0110";
when 7|17|27|37|47|57 => vOut(3 downto 0) <= "0111";
when 8|18|28|38|48|58 => vOut(3 downto 0) <= "1000";
when 9|19|29|39|49|59 => vOut(3 downto 0) <= "1001";
when others => vOut(3 downto 0) <= "1111";
end case;

case to_integer(unsigned(vIn)) is
    when 0 to 9 => vOut(7 downto 4) <= "0000";
    when 10 to 19 => vOut(7 downto 4) <= "0001";
    when 20 to 29 => vOut(7 downto 4) <= "0010";
    when 30 to 39 => vOut(7 downto 4) <= "0011";
    when 40 to 49 => vOut(7 downto 4) <= "0100";
    when 50 to 59 => vOut(7 downto 4) <= "0101";
    when others => vOut(7 downto 4) <= "1111";
end case;
end if;
end process;
end Behavioral;

```

## Div

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity div is
generic(
    clock_frequency_in : integer := 50000000;
    clock_frequency_out : integer := 5000000
);
Port ( clk_in : in STD_LOGIC;
       rst_in : in STD_LOGIC;
       clk_out : out STD_LOGIC
);
end div;
architecture Behavioral of div is
    signal clockfx : std_logic := '0';
    constant count_max_value : integer := 
clock_frequency_in/(clock_frequency_out)-1;
    signal counter : integer range 0 to count_max_value := 0;
begin
    clk_out <= clockfx;
    count_for_division: process(clk_in, rst_in)
begin
    if rst_in = '1' then
        counter <= 0;
        clockfx <= '0';
    elsif rising_edge (clk_in) then
        if counter = count_max_value then
            clockfx <= '1';
            counter <= 0;
        else

```

```

        clockfx <= '0';
        counter <= counter + 1;
    end if;
end if;
end process;

end Behavioral;
```

## Cronometro on display

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity cronometroOnDisplay is
  Port (
    clock_in: in std_logic;
    reset_n: in std_logic;
    SET_IN: in std_logic_vector(0 to 5);
    SEL_DEMUX: in std_logic_vector(0 to 1);
    BTN_ENABLE_SET: in std_logic;
    BTN_RESET: in std_logic;
    anodes_out: out std_logic_vector(7 downto 0);
    chatodes_out: out std_logic_vector(7 downto 0)
  );
end cronometroOnDisplay;

architecture Structural of cronometroOnDisplay is
component cronometro is
  Port (
    clock: in std_logic;
    reset: in std_logic;
    SET: in std_logic_vector(0 to 5);
    sel_demux: in std_logic_vector(0 to 1);
    enable_set: in std_logic;
    hours: out std_logic_vector(0 to 4);
    minute: out std_logic_vector(0 to 5);
    second: out std_logic_vector(0 to 5)
  );
end component;
component display_seven_segments is
  Generic(
    CLKIN_freq : integer := 100000000;
    CLKOUT_freq : integer := 500);
  Port ( CLK : in STD_LOGIC;
         RST : in STD_LOGIC;
         VALUE : in STD_LOGIC_VECTOR (31 downto 0);
         ENABLE : in STD_LOGIC_VECTOR (7 downto 0); -- decide quali cifre abilitare
         DOTS : in STD_LOGIC_VECTOR (7 downto 0); -- decide quali punti visualizzare
         ANODES : out STD_LOGIC_VECTOR (7 downto 0);
         CATHODES : out STD_LOGIC_VECTOR (7 downto 0));
end component;
end architecture;
```

```

component ButtonDebouncer is
    generic (
        CLK_period: integer := 10; -- periodo del clock della
board 10 nanosecondi
        btn_noise_time: integer := 6500000 --intervallo di tempo
in cui si ha l'oscillazione del bottone
                                         --assumo che duri
6.5ms=6500microsec=6500000ns
    );
    Port ( RST : in STD_LOGIC;
            CLK : in STD_LOGIC;
            BTN : in STD_LOGIC;
            CLEARED_BTN : out STD_LOGIC);
end component;
component div is
    generic(
        clock_frequency_in : integer := 50000000;
        clock_frequency_out : integer := 5000000);
    port (
        clk_in : in STD_LOGIC;
        rst_in : in STD_LOGIC;
        clk_out : out STD_LOGIC);
end component div;
component conversion is
    Port (
        vIn: in std_logic_vector(0 to 5);
        vOut: out std_logic_vector(0 to 7);
        clk: in std_logic);
end component;
signal read_temp1: std_logic;
signal read_temp2: std_logic;
signal reset_in: std_logic;
signal secondi_temp: std_logic_vector(0 to 5);
signal minuti_temp: std_logic_vector(0 to 5);
signal ore_temp: std_logic_vector(0 to 4);
signal clock_div: std_logic;
signal sec_dis: std_logic_vector(0 to 7);
signal min_dis: std_logic_vector(0 to 7);
signal ore_dis: std_logic_vector(0 to 7);
begin
reset_in <= not reset_n;
contatore: cronometro
    port map (
        clock => clock_div,
        reset => read_temp2,
        SET => SET_IN,
        sel_demux => SEL_DEMUX,
        enable_set => read_temp1,
        hours => ore_temp,
        minute => minuti_temp,
        second => secondi_temp);
divisore: div

```

```

generic map (
    clock_frequency_in => 100000000,
    clock_frequency_out  => 1)
port map (
    clk_in => clock_in,
    rst_in => reset_in,
    clk_out => clock_div);
btn1: ButtonDebouncer
    generic map (
        CLK_period => 10, -- periodo del clock della board 10
nanosecondi
        btn_noise_time => 650000000)
    port map (
        RST => reset_in,
        CLK => clock_in,
        BTN => BTN_ENABLE_SET,
        CLEARED_BTN => read_temp1);
btn2: ButtonDebouncer
    generic map (
        CLK_period => 10, -- periodo del clock della board 10
nanosecondi
        btn_noise_time => 650000000)
    port map (
        RST => reset_in,
        CLK => clock_in,
        BTN => BTN_RESET,
        CLEARED_BTN => read_temp2);
conv1: conversion
    port map(
        vIn => secondi_temp,
        vOut => sec_dis,
        clk => clock_in);
conv2: conversion
    port map(
        vIn => minuti_temp,
        vOut => min_dis,
        clk => clock_in);
conv3: conversion
    port map(
        vIn(1 to 5) => ore_temp,
        vIn(0) => '0',
        vOut => ore_dis,
        clk => clock_in);
display: display_seven_segments
generic map (
    CLKIN_freq => 100000000,
    CLKOUT_freq => 500)
port map (
    CLK => clock_in,
    RST => reset_in,
    VALUE(7 downto 0) => sec_dis,
    VALUE(15 downto 8) => min_dis,

```

```

    VALUE(23 downto 16) => ore_dis,
    VALUE(31 downto 24) => "00000000",
    ENABLE => "00111111",
    DOTS => "00010101",
    ANODES => anodes_out,
    CATHODES => chatodes_out);
end Structural;

```

## 5.3 Traccia 3

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

### 5.3.1 Soluzione

Per la realizzazione queste la funzionalità di memorizzare e visualizzare gli intertempi del cronometro, sono state apportate delle modifiche al progetto sviluppato al punto 5.2. In particolare è stato realizzato il componente intertempi con approccio behavioral, il quale in seguito all'azionamento di uno switch(catch\_inter) memorizza l'intertempo in una memoria interna, gestita come vettore circolare, il numero massimo di intertempi che si possono memorizzare è di 8, inoltre lo stesso componente in seguito alla pressione di un ulteriore pulsante(BTN\_enet), restituisce uno degli intertempi memorizzati in precedenza con ordine FIFO. Per permettere la visualizzazione sui display sia del conteggio del cronometro sia degli intertempi è stato realizzato il componente manager, il quale riceve in input sia l'output del cronometro che del componente intertempi e tramite un segnale di selezione collegato ad uno switch(mode) della board, decide quale dei due ingressi inoltrare al componente display manager per la visualizzazione. Per semplicità è stata disabilitata la possibilità di inserire un orario iniziale al cronometro, quindi al componente cronometro vengono passati dei segnali fittizi per quanto riguarda i segnali SET(6 bit), sel\_demux(2 bit) ed enable\_set, rispettivamente "000000", "00", "0". Il bottone BTN\_RESET è utilizzato per il reset del cronometro, mentre reset\_n è utilizzato per il reset degli altri componenti

#### 5.3.1.1 Schematici

In figura 5.3.1 è illustrato lo schema del cronometro su board con intertempi. Al componente intertempi sono forniti in ingresso i segnali catch\_inter e BTN\_enet utilizzati rispettivamente per catturare l'intertempo attuale e per mandare in output uno degli intertempi memorizzati, inoltre sono immessi in ingresso anche i 3 segnali second, minute ed hours utilizzati dal componente per prelevare l'ora da memorizzare come intertempo quando scatta il segnale di BTN\_enet. Il componente manager riceve in ingresso l'output del cronometro e l'output del componente intertempi, in base al valore del segnale mode(switch) immette in uscita l'orario fornito dal cronometro oppure l'orario dell'intertempo memorizzato che il componente intertempi ha messo in uscita, l'output del componente manager viene elaborato dai 3 blocchi conversion, i quali come nel progetto 5.2 servono per adattare la codifica dell'orario su 17 bit alla codifica di 4 bit per cifra utilizzata dal display manager.

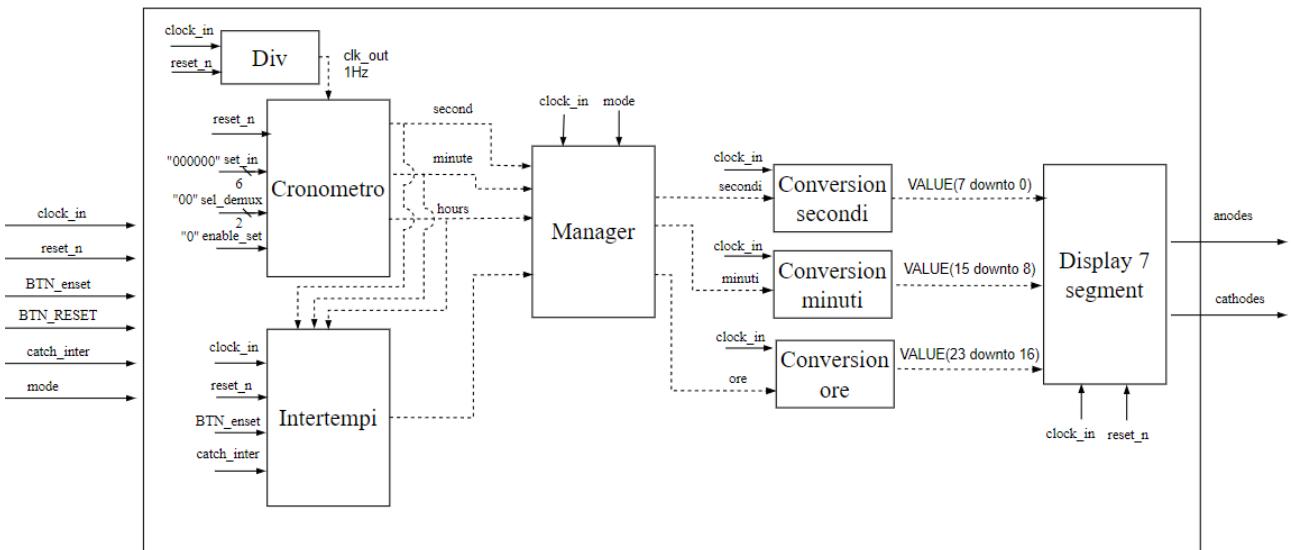


Figura 5.3.1: Schema cronometro su board con intertempi

### 5.3.2 Codice

#### Intertempi

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity intertempi is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    enSet: in std_logic;
    memSet: in std_logic;
    crono: in std_logic_vector(0 to 16);
    interOut: out std_logic_vector(0 to 16)
  );
end intertempi;
architecture Behavioral of intertempi is

signal last_enset: std_logic := '0';
type rom_type is array (7 downto 0) of std_logic_vector(16 downto 0);
signal ROM : rom_type;
signal lastMemSet : std_logic;
begin
begin
process(clk)
variable head : integer := 0;
variable tail: integer := 0;
begin
  if rising_edge(clk) then
    if(memSet='1' and lastMemSet='0') then
      ROM(head) <= crono;
      if(head = 7) then

```

```

        head := 0;
    else head := head +1;
    end if;
end if;
if(enSet = '1' and last_enset = '0') then
    last_enset <= '1';
    interOut <= ROM(tail);
    if(tail = 7) then
        tail := 0;
    else tail := tail +1;
    end if;
else last_enset <= '0';
end if;
lastMemSet <= memSet;
end if;
end process;
end Behavioral;
```

L'entity rispecchia quanto descritto in 5.3.1.1, il componente è attivo sul fronte di salita del clock, è stato definito un nuovo tipo denominato rom\_type il quale è un array(7 downto 0) di std\_logic\_vector(16 downto 0), quindi realizza la memoria per gestire fino ad 8 intertempi, ognuno codificato su 17 bit(6 per i secondi, 6 per i minuti e 5 per le ore). Il primo if scatta sul fronte di salita del segnale memSet, quindi gestisce la memorizzazione degli intertempi, infatti nel corpo dell'if è presente il codice per memorizzare il segnale crono proveniente dal componente cronometro in ROM(head), dove head tiene il riferimento logico della locazione libera su cui memorizzare, nel momento in cui head raggiunge il valore 7, il suo valore viene settato a 0, per gestire la memoria come vettore logico circolare. Il codice presente nel secondo if, scatta sul fronte di salita del segnale enSet, serve per mettere in uscita uno degli intertempi memorizzati in precedenza, in questo caso i valori memorizzati vengono consumati con logica FIFO, utilizzando una variabile tail che tiene conto della locazione dove è presente l'intertempo da mettere in output.

## Manager

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity manager is
Port (
    clock: in std_logic;
    cronoVec: in std_logic_vector(0 to 16);
    interVec: in std_logic_vector(0 to 16);
    displayOut: out std_logic_vector(0 to 16);
    sel: in std_logic
);
end manager;
architecture Behavioral of manager is
begin
process(clock)
begin
    if rising_edge(clock) then
        if(sel = '1') then
            displayOut <= interVec;
        else displayOut <= cronoVec;
    end if;
end if;
end process;
end Behavioral;
```

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

Questo componente è estremamente semplice, in sostanza realizza il comportamento di un demultiplexer con 17 bit per ogni linea di ingresso.

# Capitolo 6

## 6. Sistema di testing

### 6.1 Traccia

- a) Progettare, implementare in VHDL e verificare mediante simulazione un sistema in grado di testare in maniera automatica una macchina combinatoria M avente 4 ingressi e 3 uscite binarie sottoponendole N ingressi diversi (si considerino una macchina M e un numero di input N a scelta dello studente). Gli N valori di input per il test devono essere letti da una ROM, in cui essi sono precaricati, in corrispondenza di un segnale read. Le N uscite fornite della macchina in corrispondenza di ciascuno degli input devono essere memorizzate in una memoria interna, che deve poter essere svuotata in qualsiasi momento in presenza di un segnale di reset.
- b) Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando due bottoni per i segnali di read e reset rispettivamente e i led per la visualizzazione delle uscite della macchina istante per istante.

### 6.2 Soluzione

Anche in questo caso, abbiamo risolto l'esercizio utilizzando un approccio modulare, mediante *decomposizione funzionale*: siamo partiti dalla realizzazione, con un livello di astrazione di tipo *behaviorial*, di una memoria ROM, abbiamo poi utilizzato due versioni differenti di questo componente, per memorizzare i 4 bit di input del componente da testare ed i 3 bit di uscita validi. Inoltre per memorizzare l'output reale del componente da testare abbiamo utilizzato un'altra memoria realizzata sempre con un livello di astrazione behavioral. Il componente Compare, utilizzato per confrontare l'output reale della macchina e l'output corretto è stato descritto sempre tramite approccio behavioral, mentre il componente da testare, ovvero il RippleCarryAdder è stato realizzato tramite approccio structural adoperando due FullAdder. Per la realizzazione del sistema complessivo è stato adoperato un approccio structural, collegando tutti i componenti descritti in precedenza. Infine, è stato utilizzato un componente contatore, per memorizzare l'indice della locazione della rom da cui leggere l'input da fornire alla macchina da testare, inoltre lo stesso indice è utilizzato per memorizzare l'output prodotto dalla macchina testata.

#### 6.2.1 Schematici

La macchina è suddivisa in unità operativa ed unità di controllo. **L'unità operativa** (figura 6.1) realizza l'immissione dei dati di input prelevati dalla ROM nella macchina da testare, inoltre provvede a memorizzare gli output ed a confrontarli con gli output validi, presenti in un ulteriore ROM. L'input è composto da 4 bit, essi rappresentano i due numeri che la macchina di test (Ripple Carry Adder) dovrà elaborare, mentre il segnale di output è composto da 3 bit, poiché oltre i due bit per rappresentare il risultato è presente anche un eventuale riporto. L'unità operativa è realizzata tramite approccio strutturale, collegando vari componenti tra cui: 2 memorie ROM, la prima contiene gli input da fornire alla macchina da testare(4 bit), la seconda contiene gli output(3 bit) validi per gli ingressi presenti nella prima ROM, mentre per la memorizzazione dell'output reale della macchina da testare è utilizzata una terza memoria. Per effettuare la comparazione degli output è utilizzato un

componente denominato Compare. L'unità operativa presenta in ingresso diversi segnali, il clock, un segnale di reset, e vari segnali di abilitazione per i componenti interni appena descritti quali :

- `read_rom`: utilizzato per abilitare la lettura dal componente ROM Input, del prossimo valore di input da fornire alla macchina da testare
- `read_rom_out`: serve ad abilitare la lettura dal componente ROM Output del valore di output valido per l'input appena fornito
- `write_memo_out`: abilita la memorizzazione dell'output reale della macchina
- `load`, `enable`: sono utilizzati rispettivamente per caricare i due output(reale, valido da ROM) da confrontare nel componente compare
- `enCount`: per far scattare il contatore, così da incrementare il valore di conteggio

In uscita troviamo solamente i segnali `test` che indica il risultato del confronto tra output reale e output valido per l'input attualmente fornito alla macchina, e `fine` che indica se sono stati testati o meno tutti gli ingressi presenti nella ROM.

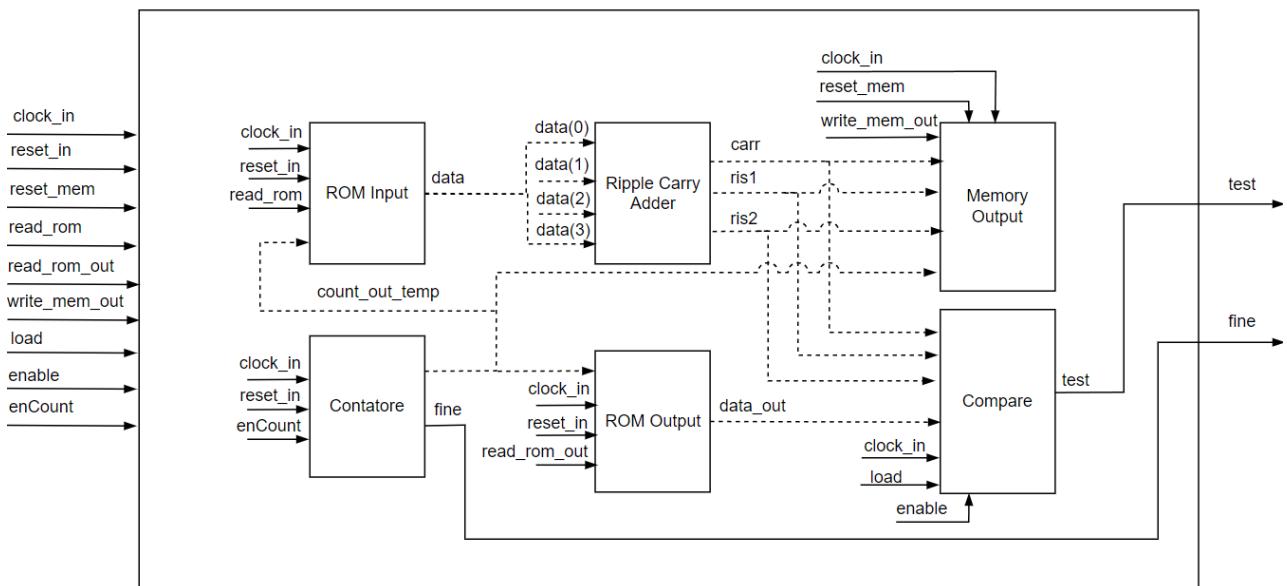


Figura 6.1: Unità Operativa

**L'unità di controllo** invece è realizzata tramite approccio behavioral e realizza l'automa(figura 6.2) utilizzato per azionare i vari componenti dell'unità operativa. Nel port map troviamo gli stessi segnali presenti nell'unità operativa, in piùabbiamo un segnale in ingresso start, utilizzato per abilitare l'unità di controllo. L'automa presenta 7 stati

- `idle`: in questo stato la macchina aspetta che il segnale di start diventi alto, per passare al secondo stato `readROM`
- `readROM`: tramite i segnali `read_rom` e `read_rom_out` viene abilitata la lettura dalle due ROM Input\Output, rispettivamente dell'ingresso da fornire alla macchina da testare e dell'output valido per quell'input, si passa allo stato successivo `acquisition`
- `acquisition`: tramite il segnale `load` viene abilitato il caricamento dell'output reale prodotto dalla macchina da testare e dell'output valido nel componente `Compare`, si passa allo stato `compare`

- compare: in questo stato tramite il segnale enable si abilita il componente Compare, esso effettua il confronto ed immette in uscita il risultato tramite il segnale test, si passa allo stato memorization
- memorization: si abilita l'unità operativa a memorizzare l'output prodotto dalla macchina da testare nella memoria, tramite il segnale write\_mem\_out, si passa allo stato increase
- increase: in questo stato si abilita tramite il segnale enCount, il contatore presente nell'unità operativa, in modo da far incrementare il valore di conteggio, il prossimo stato è endtest
- endtest: si verifica se il segnale di fine è alto o meno, nel caso sia alto ciò significa che sono stati effettuati tutti i test quindi si passa allo stato di idle, in caso contrario si passa allo stato readROM da cui ricomincia la sequenza di operazioni per testare il valore di input successivo.

In ogni stato vengono abbassati i segnali alzati nello stato precedente. Infine, la macchina complessiva Sistema Testing è realizzata collegando l'unità di controllo e l'unità operativa.

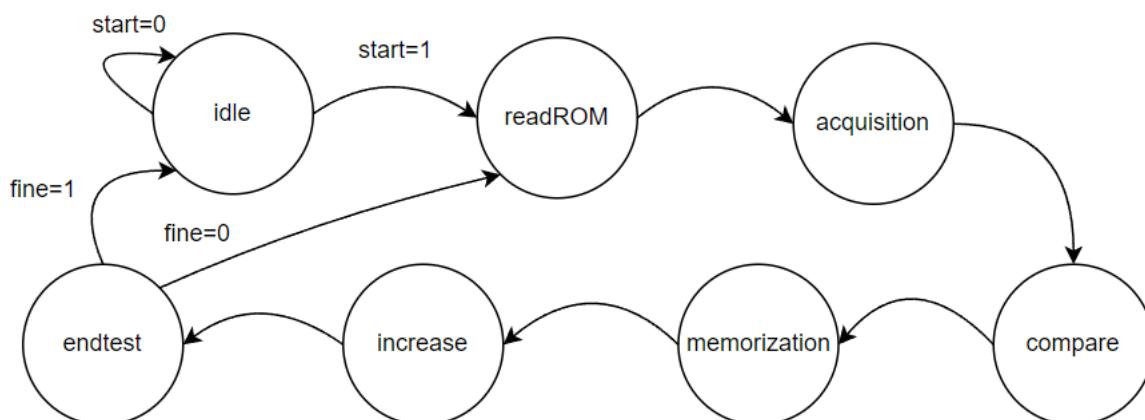


Figura 6.2: Automa Unità di Controllo

### 6.3 Codice

#### ROM Input

I valori di input sono memorizzati in value che è di tipo init\_rom, nel process essi vengono caricati in RO, inoltre il process è sensibile al segnale clk, la condizione if(READ='1') fa sì che quando viene dato alto il segnale di READ, il componente immetta in uscita al segnale DATA l'input i-esimo memorizzato nella ROM. Riportiamo solo il codice della ROM Input poiché i componenti ROM Output e Mem Output sono stati realizzati allo stesso modo.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity ROM is
  generic(
    N: integer := 5
  );
  port(
    CLK : in std_logic;
    RST : in std_logic;
    READ : in std_logic;

```

```

count: in integer;
DATA : out std_logic_vector(3 downto 0)
);
end ROM;
architecture behavioral of ROM is
type rom_type is array (0 to N-1) of std_logic_vector(3 downto 0);
type init_rom is array (0 to 15) of std_logic_vector(3 downto 0);
signal ROM : rom_type;
signal value : init_rom := (
"0101",
"1001",
"1100",
"1101",
"0011",
"0111",
"0000",
"1010",
"1110",
"0001",
"0010",
"1011",
"1111",
"1000",
"0100",
"0110");
signal last_read: std_logic;
begin
process(clk)
begin
if rising_edge(clk) then
    init: for i in 0 to N-1 loop
        ROM(i) <= value(i);
    end loop init;
    if(READ = '1') then
        DATA <= ROM(count);
    end if;
end if;
end process;
end Behavioral;

```

## Compare

Quando viene dato alto il segnale di load significa che i valori in ingresso sono significativi, quindi il componente preleva i valori in input da confrontare(value1 e value2) e ne copia il contenuto nei segnali value1\_temp e value2\_temp in modo che se anche gli input dovessero cambiare ne è presente una copia fatta nell'istante in cui erano significativi gli ingressi. La seconda condizione nel momento in cui il segnale di enable viene dato alto, effettua il confronto tra i due valori salvati in precedenza (value1\_temp e value2\_temp) ed immette in uscita il risultato del confronto tramite il segnale di comp.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Compare is
  Port (
    clk: in std_logic;
    value1: in std_logic_vector(2 downto 0);
    value2: in std_logic_vector(2 downto 0);
    load: in std_logic;
    enable: in std_logic;
    comp: out std_logic
  );
end Compare;
architecture Behavioral of Compare is
  signal value1_temp: std_logic_vector(2 downto 0);
  signal value2_temp: std_logic_vector(2 downto 0);
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if(load = '1') then
        value1_temp <= value1;
        value2_temp <= value2;
      elsif(enable = '1') then
        if((value1_temp(0)      =      value2_temp(0)) and
           (value1_temp(1)      =      value2_temp(1)) and
           (value1_temp(2)      =      value2_temp(2))) then
          comp <= '1';
        else
          comp <= '0';
        end if;
      end if;
    end if;
  end process;
end Behavioral;

```

## Contatore

Il contatore è attivo sul fronte di discesa, quando il segnale enable è alto viene controllato il valore di conteggio, se esso è pari ad N-1 significa che abbiamo effettuato tutti i casi di test, quindi il conteggio viene resettato e viene posto alto il segnale di fine, per segnalare all'unità di controllo che sono terminati i casi di test da eseguire, in caso contrario il valore di conteggio viene incrementato.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
entity contatore is
  generic(
    N: integer := 2
  );
  Port (
    clk: in std_logic;
    rst: in std_logic;

```

```

enable: in std_logic;
fine: out std_logic;
count_out: out integer
);
end contatore;
architecture Behavioral of contatore is

signal ty: std_logic;
signal last_enset: std_logic;
signal last_clk: std_logic;

begin
process(clk,rst,enable)
variable count: integer := 0;
begin
    if(rst = '1') then
        count := 0;
        fine <= '0';
    end if;
    if falling_edge(clk) then
        if(enable = '1') then
            if(count = N-1) then
                count := 0;
                fine <= '1';
            else
                count := count +1;
                fine <= '0';
            end if;
        end if;
        count_out <= count;
    end process;
end Behavioral;

```

## Ripple Carry Adder

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity RippleCarryAdder is
Port (
    b0: in std_logic;
    b1: in std_logic;
    a0: in std_logic;
    a1: in std_logic;
    c_out: out std_logic;
    s0: out std_logic;
    s1: out std_logic
);
end RippleCarryAdder;
architecture Structural of RippleCarryAdder is
component FullAdder is
Port (

```

```

    a: in std_logic;
    b: in std_logic;
    carry_in: in std_logic;
    carry_out: out std_logic;
    ris: out std_logic
);
end component;
signal carry_temp: std_logic;
begin
F1: FullAdder
    port map(
        a => a0,
        b => b0,
        carry_in => '0',
        carry_out => carry_temp,
        ris => s0
    );
F2: FullAdder
    port map(
        a => a1,
        b => b1,
        carry_in => carry_temp,
        carry_out => c_out,
        ris => s1
    );
end Structural;

```

## Unità Operativa

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Unita_Operativa is
    Port (
        clock_in: in std_logic;
        reset_in: in std_logic;
        reset_mem: in std_logic;
        read_rom: in std_logic;
        read_rom_out: in std_logic;
        write_mem_out: in std_logic;
        load: in std_logic;
        enable: in std_logic;
        enCount: in std_logic;
        fine: out std_logic;
        test: out std_logic
    );
end Unita_Operativa;

architecture Structural of Unita_Operativa is

component RippleCarryAdder is
    port(
        b0: in std_logic;

```

```

        b1: in std_logic;
        a0: in std_logic;
        a1: in std_logic;
        c_out: out std_logic;
        s0: out std_logic;
        s1: out std_logic
    );
end component;

component ROM is
    generic(
        N: integer := 5
    );
    port(
        CLK : in std_logic;
        RST : in std_logic;
        READ : in std_logic;
        count: in integer;
        DATA : out std_logic_vector(3 downto 0)
    );
end component;

component ROM_output is
    generic(
        N: integer := 5
    );
    port(
        CLK : in std_logic;
        RST : in std_logic;
        READ : in std_logic;
        count: in integer;
        DATA : out std_logic_vector(2 downto 0)
    );
end component;

component Mem_output is
    generic(
        N: integer := 5
    );
    Port (
        clock: in std_logic;
        reset: in std_logic;
        write: in std_logic;
        count: in integer;
        value: in std_logic_vector(2 downto 0)
    );
end component;

component Compare is
    Port (
        clk: in std_logic;
        value1: in std_logic_vector(2 downto 0);

```

```

value2: in std_logic_vector(2 downto 0);
load: in std_logic;
enable: in std_logic;
comp: out std_logic
);
end component;

component contatore is
  generic(
    N: integer := 2
  );
  Port (
    clk: in std_logic;
    rst: in std_logic;
    enable: in std_logic;
    fine: out std_logic;
    count_out: out integer
  );
end component;

signal data: std_logic_vector(3 downto 0);
signal data_out: std_logic_vector(2 downto 0);
signal carr: std_logic;
signal ris1: std_logic;
signal ris2: std_logic;
signal count_out_temp: integer;

begin

rom_input: ROM
  generic map(
    N => 8
  )
  port map(
    CLK => clock_in,
    RST => reset_in,
    READ => read_rom,
    count => count_out_temp,
    DATA => data
  );

rom_out: ROM_output
  generic map(
    N => 8
  )
  port map(
    CLK => clock_in,
    RST => reset_in,
    READ => read_rom_out,
    count => count_out_temp,
    DATA => data_out
  );

```

```

) ;

adder: RippleCarryAdder
  port map(
    b0 => data(0),
    b1 => data(1),
    a0 => data(2),
    a1 => data(3),
    c_out => carr,
    s0 => ris1,
    s1 => ris2
  ) ;

mem_out: Mem_output
  generic map(
    N => 8
  )
  Port map(
    clock => clock_in,
    reset => reset_mem,
    write => write_mem_out,
    count => count_out_temp,
    value(0) => carr,
    value(1) => ris2,
    value(2) => ris1
  );
);

comp: Compare
  Port map(
    clk => clock_in,
    value1 => data_out,
    value2(2) => carr,
    value2(1) => ris2,
    value2(0) => ris1,
    load => load,
    enable => enable,
    comp => test
  );
);

cont: contatore
  generic map(
    N => 8
  )
  Port map(
    clk => clock_in,
    rst => reset_in,
    enable => enCount,
    fine => fine,
    count_out => count_out_temp
  );
);

```

```
end Structural;
```

## Unità di Controllo

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Unita_Controllo is
  Port (
    clock: in std_logic;
    reset: in std_logic;
    start: in std_logic;
    reset_out: out std_logic;
    read_rom: out std_logic;
    read_rom_out: out std_logic;
    write_mem_out: out std_logic;
    load: out std_logic;
    enCount: out std_logic;
    fine: in std_logic;
    enable: out std_logic
  );
end Unita_Controllo;

architecture Behavioral of Unita_Controllo is

  type stato is (idle,readROM,acquisition,compare,memorization,
increase,endtest);
  signal stato_corrente : stato := idle;

  begin
  process(clock, reset)
  begin
    if(reset = '1') then
      stato_corrente <= idle;
    end if;
    if rising_edge(clock) then
      case stato_corrente is
      when idle =>
        reset_out <= '1';
        read_rom <= '0';
        read_rom_out <= '0';
        write_mem_out <= '0';
        load <= '0';
        enCount <= '0';
        enable <= '0';
        if(start = '1') then
          stato_corrente <= readROM;
        else stato_corrente <= idle;
        end if;
      when readROM =>
        reset_out <= '0';
        read_rom <= '1'; --Legge l'input da fornire in ingresso,
dalla locazione count e lo immette in data
      end case;
    end if;
  
```

```

        read_rom_out <= '1'; --Legge output valido da ROM, dalla
locazione count e lo immette in data_out
        --Il RippleCarryAdder effettua la somma degli operandi
presenti in data
        write_mem_out <= '0';
        load <= '0';
        enCount <= '0';
        enable <= '0';
        stato_corrente <= acquisition;
when acquisition =>
    reset_out <= '0';
    read_rom <= '0';
    read_rom_out <= '0';
    write_mem_out <= '0';
    load <= '1';
    --Il Compare carica l'output valido dalla ROM
RippleCarryAdder(ris1,ris2,carr) e l'output valido dalla ROM
data_out(ROM Output)
    enCount <= '0';
    enable <= '0';
    stato_corrente <= compare;
when compare =>
    reset_out <= '0';
    read_rom <= '0';
    read_rom_out <= '0';
    write_mem_out <= '0';
    load <= '0';
    enCount <= '0';
    enable <= '1'; --Il Compare effettua il confronto e mette
in output il risultato
                    --tramite il segnale test
    stato_corrente <= memorization;
when memorization =>
    reset_out <= '0';
    read_rom <= '0';
    read_rom_out <= '0';
    write_mem_out <= '1'; --Salva in memoria l'output del
Ripple Carry Adder
    load <= '0';
    enCount <= '0';
    enable <= '0';
    stato_corrente <= increase;
when increase =>
    reset_out <= '0';
    read_rom <= '0';
    read_rom_out <= '0';
    write_mem_out <= '0';
    load <= '0';
    enCount <= '1'; --Incrementa il valore di conteggio
    enable <= '0';
    stato_corrente <= endtest;
when endtest =>

```

```

    reset_out <= '0';
    read_rom <= '0';
    read_rom_out <= '0';
    write_mem_out <= '0';
    load <= '0';
    enCount <= '0';
    enable <= '0';
    if(fine = '1') then
        --fine è dato dal Contatore, se vale 1, significa che
        sono stati fatti tutti i test
        stato_corrente <= idle; --prossimo stato idle
    else
        stato_corrente <= readROM;
    end if;
    end case;
end if;
end process;
end Behavioral;

```

## Sistema di Testing

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SistemaTesting is
    Port (
        clock_in: in std_logic;
        reset_in: in std_logic;
        start: in std_logic;
        reset_mem: in std_logic;
        test: out std_logic
    );
end SistemaTesting;

architecture Structural of SistemaTesting is
component Unita_Operativa is
    Port (
        clock_in: in std_logic;
        reset_in: in std_logic;
        reset_mem: in std_logic;
        read_rom: in std_logic;
        read_rom_out: in std_logic;
        write_mem_out: in std_logic;
        load: in std_logic;
        enable: in std_logic;
        enCount: in std_logic;
        fine: out std_logic;
        test: out std_logic
    );
end component;

component Unita_Controllo is
    Port (

```

```

clock: in std_logic;
reset: in std_logic;
start: in std_logic;
reset_out: out std_logic;
read_rom: out std_logic;
read_rom_out: out std_logic;
write_mem_out: out std_logic;
load: out std_logic;
enCount: out std_logic;
fine: in std_logic;
enable: out std_logic
);
end component;

```

```

signal read_rom_temp: std_logic;
signal read_rom_out_temp: std_logic;
signal write_mem_out_temp: std_logic;
signal load_temp: std_logic;
signal enable_temp: std_logic;
signal reset_temp: std_logic;
signal enCount_temp: std_logic;
signal fine_temp: std_logic;

```

```
begin
```

UO: Unita\_Operativa

```

port map(
    clock_in => clock_in,
    reset_in => reset_temp,
    reset_mem => reset_mem,
    read_rom => read_rom_temp,
    read_rom_out => read_rom_out_temp,
    write_mem_out => write_mem_out_temp,
    load => load_temp,
    enable => enable_temp,
    enCount => enCount_temp,
    fine => fine_temp,
    test => test
);

```

UC: Unita\_Controllo

```

port map(
    clock => clock_in,
    reset => reset_in,
    start => start,
    reset_out => reset_temp,
    read_rom => read_rom_temp,
    read_rom_out => read_rom_out_temp,
    write_mem_out => write_mem_out_temp,
    load => load_temp,
    enCount => enCount_temp,
    fine => fine_temp,

```

```

    enable => enable_temp
  );
end Structural;

```

## 6.4 Simulazione

Per la simulazione, dato che i valori di input da testare sono presenti già nella ROM dell'unità operativa, è stato sufficiente alzare il segnale di start, per avviare il Sistema di Testing, di seguito è mostrato il codice del testbench effettuato:

### Testbench

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TB_testing is
end TB_testing;
architecture Behavioral of TB_testing is
component SistemaTesting is
  Port (
    clock_in: in std_logic;
    reset_in: in std_logic;
    start: in std_logic;
    reset_mem: in std_logic;
    test: out std_logic
  );
end component;

signal clk: std_logic;
signal start: std_logic := '0';
signal reset_mem: std_logic := '0';
signal rst: std_logic := '0';
signal uscita_test: std_logic := '0';
constant CLK_period : time := 10 ns;
begin
  uut: SistemaTesting
    port map(
      clock_in => clk,
      reset_in => rst,
      start => start,
      reset_mem => reset_mem,
      test => uscita_test
    );
-- Clock process definitions
  CLK_process :process
  begin
    clk <= '0';
    wait for CLK_period/2;
    clk <= '1';
    wait for CLK_period/2;
  end process;

  stim_proc: process

```

```

begin
    -- hold reset state for 100 ns.
    wait for 100 ns;
    wait for CLK_period*10;
    start <= '1';
    wait for 50 ns;
    start <= '0';
    wait;
end process;
end Behavioral;

```

Come possiamo osservare in figura 6.3, tutti i casi di test vanno a buon fine, dato che il segnale uscita\_test rimane alto per tutti i casi di test effettuati.

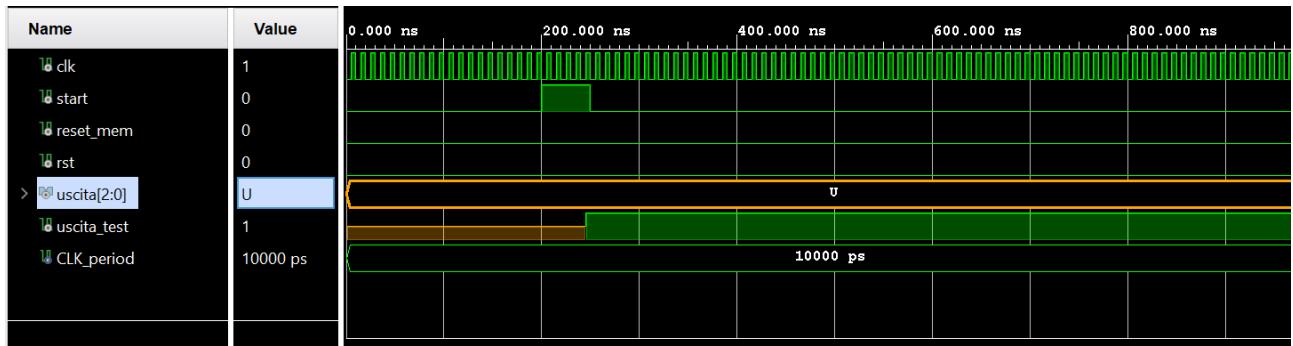


Figura 6.3: Simulazione Sistema di Testing

## 6.5 Sintesi su Board

Il Sistema di Testing è stato implementato sulla board modificando il progetto descritto in precedenza. L’unità operativa è inalterata, mentre sono state apportate delle modifiche all’unità di controllo, in particolare vengono adoperati 2 pulsanti distinti reset\_in e reset\_mem, rispettivamente per il reset della macchina di testing e per il reset della memoria dove sono salvati gli output del Ripple Carry Adder, 2 ulteriori pulsanti per interagire con lo macchina, il primo denominato startBTN il quale serve per dare il segnale di start all’unità di controllo, il secondo readBTN che serve per abilitare il l’immissione dell’input successivo nel Ripple Carry Adder, infine è presente un led utilizzato per visualizzare lo stato del segnale test, che indica se il caso di test attuale è andato a buon fine o meno. Il nuovo automa dell’unità di controllo è identico a quello descritto nel progetto in precedenza, tranne che per lo stato di readROM dove si attende la pressione del pulsante per fornire in input al Ripple Carry Adder il nuovo ingresso

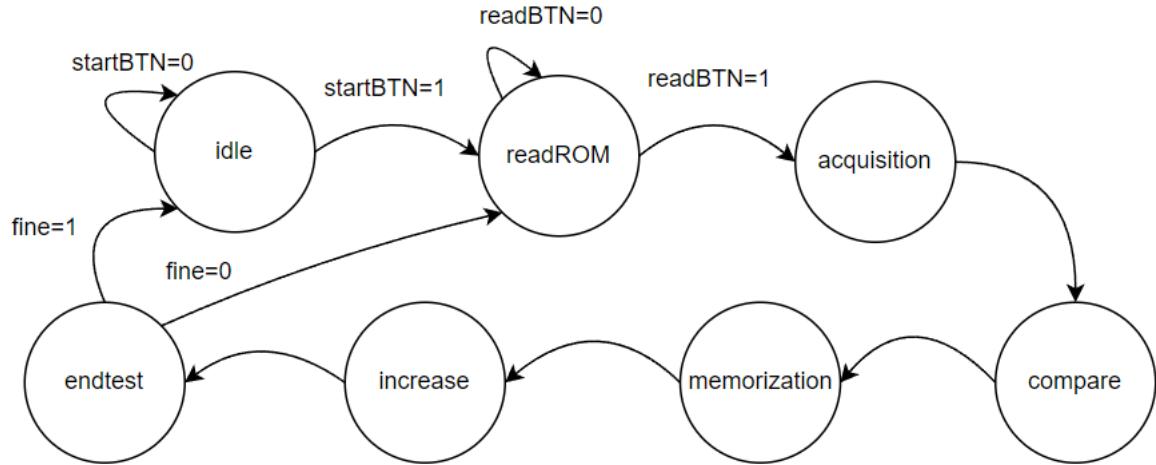


Figura 6.4: Unità di Controllo su Board

Di seguito è riportata la porzione di codice dell'unità di controllo, modificata per codificare il nuovo automa

```

when idle =>
    reset_out <= '1';
    read_rom <= '0';
    read_rom_out <= '0';
    write_mem_out <= '0';
    load <= '0';
    enCount <= '0';
    enable <= '0';
    if(start = '1' and last_start = '0') then
        last_start <= '1';
        stato_corrente <= readROM;
    else
        last_start <= '0';
        stato_corrente <= idle;
    end if;
when readROM =>
    reset_out <= '0';
    write_mem_out <= '0';
    load <= '0';
    enCount <= '0';
    enable <= '0';
    if(read = '1' and last_read = '0') then
        last_read <= '1';
        read_rom <= '1';
        read_rom_out <= '1';
        stato_corrente <= acquisition;
    else
        last_read <= '0';
        stato_corrente <= readROM;
    end if;

```

Per la gestione dei pulsanti è stato creato un modulo intermedio denominato Unità Controllo Button realizzato tramite approccio strutturale, esso serve per collegare i 2 pulsanti startBTN e readBTN all'unità di controllo, utilizzando 2 Button Debouncer per filtrare i segnali provenienti dai pulsanti startBTN e readBTN.

Di seguito è riportato la porzione di codice dell'unità Unità Controllo Button che collega i 3 componenti:

```
UC: Unita_Controllo
    port map (
        clock => clock,
        reset => reset,
        start => btn_temp1,
        reset_out => reset_out,
        read_rom => read_rom,
        read_rom_out => read_rom_out,
        write_mem_out => write_mem_out,
        load => load,
        enCount => enCount,
        read => btn_temp2,
        fine => fine,
        enable => enable
    );
    
BTN1: ButtonDebouncer
    generic map (
        CLK_period => 10, -- periodo del clock della board 10
nanosecondi
        btn_noise_time => 650000000 --intervallo di tempo in
cui si ha l'oscillazione del bottone
                                         --assumo che duri
6.5ms=6500microsec=6500000ns
    )
    port map (
        RST => reset,
        CLK => clock,
        BTN => start_btn,
        CLEARED_BTN => btn_temp1
    );
    
BTN2: ButtonDebouncer
    generic map (
        CLK_period => 10, -- periodo del clock della board 10
nanosecondi
        btn_noise_time => 650000000 --intervallo di tempo in
cui si ha l'oscillazione del bottone
                                         --assumo che duri
6.5ms=6500microsec=6500000ns
    )
    port map (
        RST => reset,
```

```
CLK => clock,  
BTN => read_btn,  
CLEARED_BTN => btn_temp2  
) ;
```

Infine, il Sistema di Testing è realizzato componendo l'unità operativa ed il modulo unità di controllo button.

# 7.Comunicazione con Handshaking

## 7.1 Traccia

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

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

## 7.2 Cenni teorici

Quando dobbiamo far comunicare tra di loro entità differenti, che magari devono cooperare verso un obiettivo comune, è necessario utilizzare dei protocolli per regolamentare tale comunicazione. Nasce allora un problema di sincronizzazione, un problema di protocollo. Possiamo classificare i protocolli in tre categorie:

- Protocolli sincroni
- Protocolli asincroni
- Protocolli semisincroni

In particolare, i protocolli asincroni sono basati sul concetto di **evento**. Le entità comunicano alla ricezione di eventi e finché non si verificano eventi, non trasmettono. Tra questi protocolli, troviamo l'**handshaking completo** per il quale abbiamo i seguenti eventi:

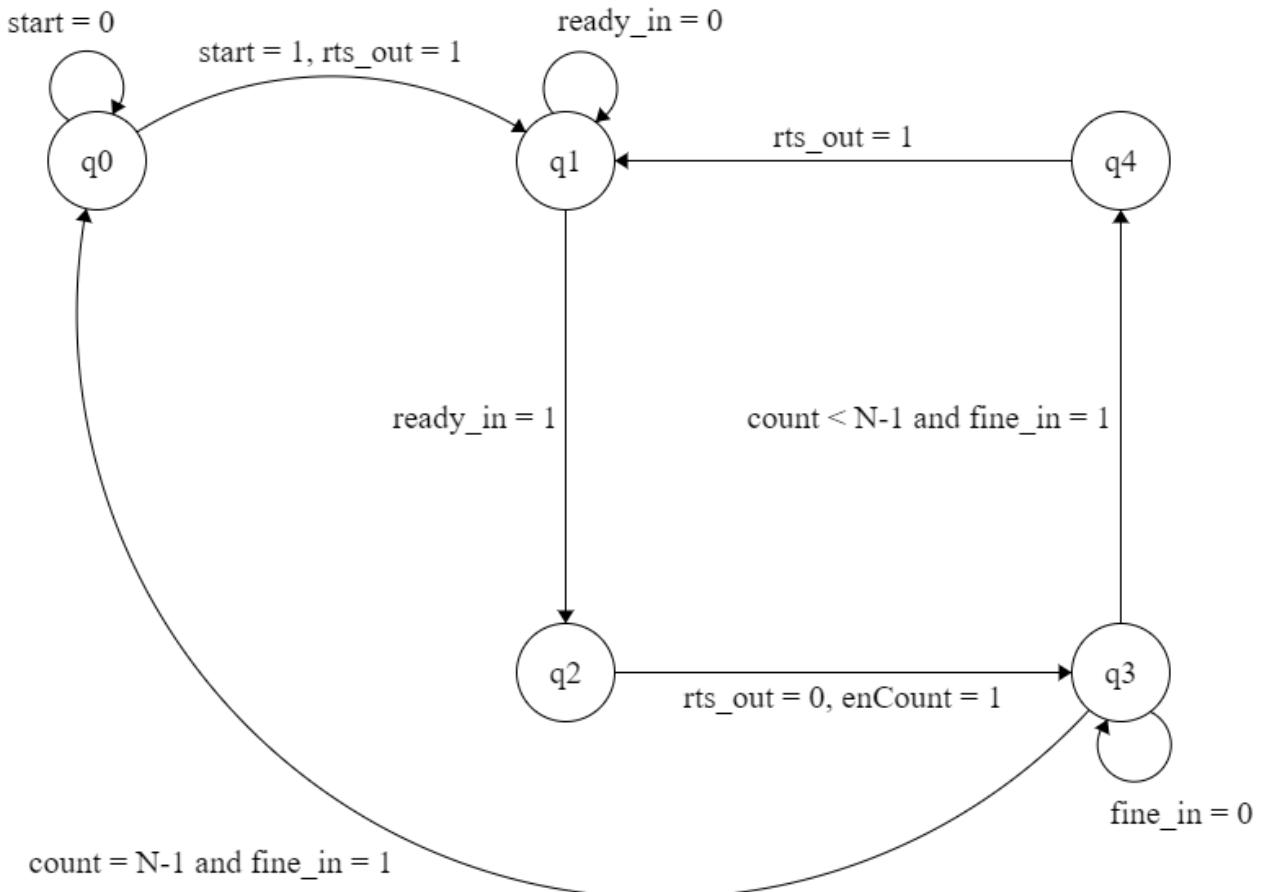
1. L'entità A manda il dato
2. L'entità A dà il via
3. L'entità B risponde di essere pronta
4. L'entità B invia un messaggio per avvertire di aver terminato l'operazione

## 7.3 Soluzione

Abbiamo implementato un sistema composto da due nodi, A (Trasmettitore) e B (Ricevitore), che comunicano tra loro mediante il protocollo asincrono di handshaking completo. Il nodo A presenta una memoria interna, ROM, in cui sono memorizzate  $N=10$  stringhe da  $M=3$  bit per la trasmissione. Il nodo B è caratterizzato da due memorie interne, ROM\_B, nella quale sono memorizzate rispettivamente  $N=10$  stringhe da  $M=3$  bit e ROM\_S, in cui memorizziamo il risultato della somma tra le stringhe del trasmettitore e quelle del ricevitore (non sono stati considerati eventuali riporti). Entrambi i sistemi sono caratterizzati da un componente contatore per scandire la trasmissione/ricezione delle stringhe e terminare la comunicazione. Infine, al fine di effettuare la somma tra le stringhe è stato sviluppato, all'interno del sistema B, un componente “adder” con un approccio di tipo comportamentale.

Innanzitutto, abbiamo realizzato gli automi che permettono di descrivere con precisione e in maniera formale il comportamento dei due sistemi. A partire da essi abbiamo sviluppato il codice in VHDL.

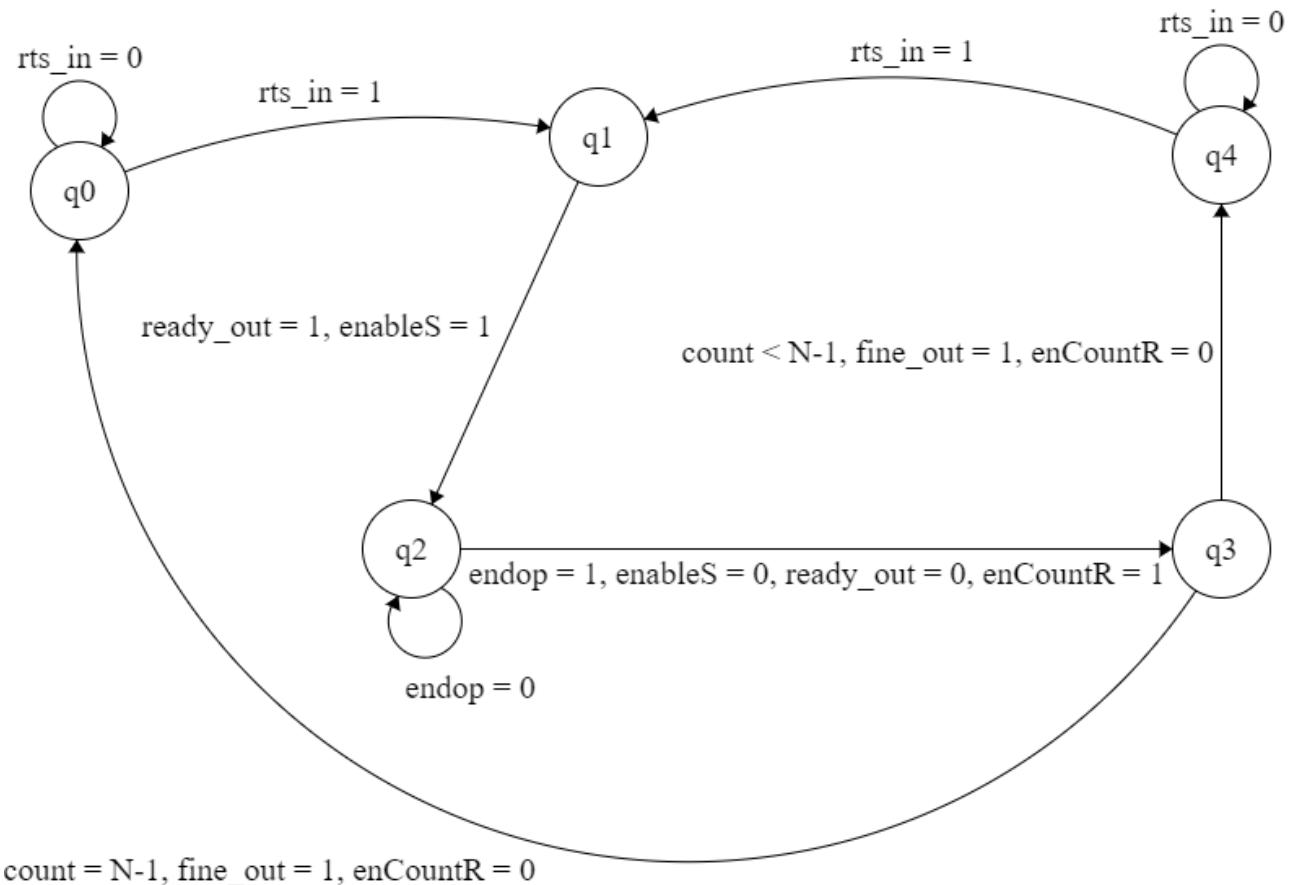
### 7.3.1 Automa Trasmettitore



Per comprendere al meglio l'automa effettuiamo l'analisi stato per stato.

- **Q0:** Il processo di trasmissione è caratterizzato da un segnale di *start* che scandisce l'inizio del trasferimento delle  $N$  stringhe. Appena tale segnale diventa alto, il trasmettitore darà il via alla trasmissione alzando il segnale *rts\_out*, inviando la prima stringa e passando in q1.
- **Q1:** Una volta giunti in questo stato, si passerà al successivo (q2) solo una volta che il segnale *ready\_in* sarà alto e quindi solo quando il ricevitore comunicerà di essere pronto alla ricezione.
- **Q2:** In questo stato viene abbassato il segnale *rts\_out* dato che ormai il ricevitore ha comunicato di essere pronto. Inoltre, viene alzato il segnale di enable del contatore, *enCount*, in modo tale da incrementare il contatore per la trasmissione successiva.
- **Q3:** Appena giunti in questo stato, al successivo colpo di clock, come per ognuno di essi, si abbassa il segnale *enCount* in quanto vogliamo che venga incrementato solo di uno il contatore. Si rimane in q3 fin quando il ricevitore alzerà il segnale *fine\_in*, il quale scandisce la fine dell'operazione. Nel momento in cui verrà alzato tale segnale, si andrà nello stato q4 se il valore di *count* è minore di  $N-1$ , mentre si andrà nello stato q0, se esso è pari a  $N-1$  e quindi tutte le stringhe sono state inviate, in attesa nuovamente del segnale di start.
- **Q4:** Giunti in questo stato, il trasmettitore darà il via alla trasmissione successiva alzando il segnale *rts\_out*, inviando la stringa e passando in q1.

### 7.3.2 Automa Ricevitore



Per comprendere al meglio l'automa effettuiamo l'analisi stato per stato.

- **Q0:** Lo stato iniziale del ricevitore è caratterizzato dall'attesa del segnale *rts\_in*, il quale risulterà alto nel momento in cui il trasmettitore darà il via alla trasmissione e manderà il dato. Una volta fatto ciò si passa allo stato q1.
- **Q1:** Una volta giunti in questo stato, il ricevitore alzerà il segnale *ready\_out*, tramite il quale avviserà il trasmettitore di essere pronto, e il segnale *enableS* mediante il quale verrà data l'abilitazione al componente “adder” per effettuare la somma.
- **Q2:** In questo stato il trasmettitore aspetta che l'adder abbia terminato l'operazione di somma (*endop* = 1). Fatto ciò, abbassa il segnale di abilitazione dell'adder e il segnale *ready\_out* dato che ormai il dato è stato ricevuto. Infine, abilita il contatore alzando il segnale *enCountR*, in modo tale da incrementare il contatore per la trasmissione successiva.
- **Q3:** Appena giunti in questo stato, al successivo colpo di clock, come per ognuno di essi, si abbassa il segnale *enCount* in quanto vogliamo che venga incrementato solo di uno il contatore. Se il conteggio è minore di N-1, si alza il segnale *fine\_out* per indicare la fine dell'operazione e si passa allo stato q4 in attesa della stringa successiva. Se invece il conteggio è pari a N-1 e quindi è stata effettuata la ricezione dell'ultima stringa, si alza il segnale *fine\_out* e si passa allo stato q0 in attesa di ulteriori trasmissioni.
- **Q4:** Giunti in questo stato, si aspetta nuovamente il segnale *rts\_in* che darà il via alla ricezione successiva passando in q1. Si può notare che tale stato esegue le stesse operazioni dello stato q0. È stato inserito non solo per motivi di simmetria tra gli automi del ricevitore e del trasmettitore, ma anche per ritornare allo stato iniziale solo una volta effettuata la ricezione delle N stringhe.

### 7.3.3 Schematici

Il **SystemA** è caratterizzato da tre componenti fondamentali: Trasmettitore, Contatore e ROM, come mostra la figura 7.1. In ingresso il blocco riceve il *clock*, il *reset*, il segnale di start e i segnali *readyA* e *fineA*, provenienti dal ricevitore, i quali indicano rispettivamente che il ricevitore è pronto e la fine dell'operazione. In uscita abbiamo il dato (*dataA*) di M bit che poi sarà trasmesso al ricevitore e il segnale *rtsA*, tramite la quale il sistema dà il via alla trasmissione. Il blocco Trasmettitore, descritto successivamente nello specifico, è stato sviluppato al fine di realizzare l'handshaking dalla parte del trasmettitore e quindi l'automa descritto nel paragrafo precedente. Il Contatore è utilizzato per scandire la trasmissione delle stringhe e terminare la comunicazione. Infine, la ROM contiene le N stringhe da inviare.

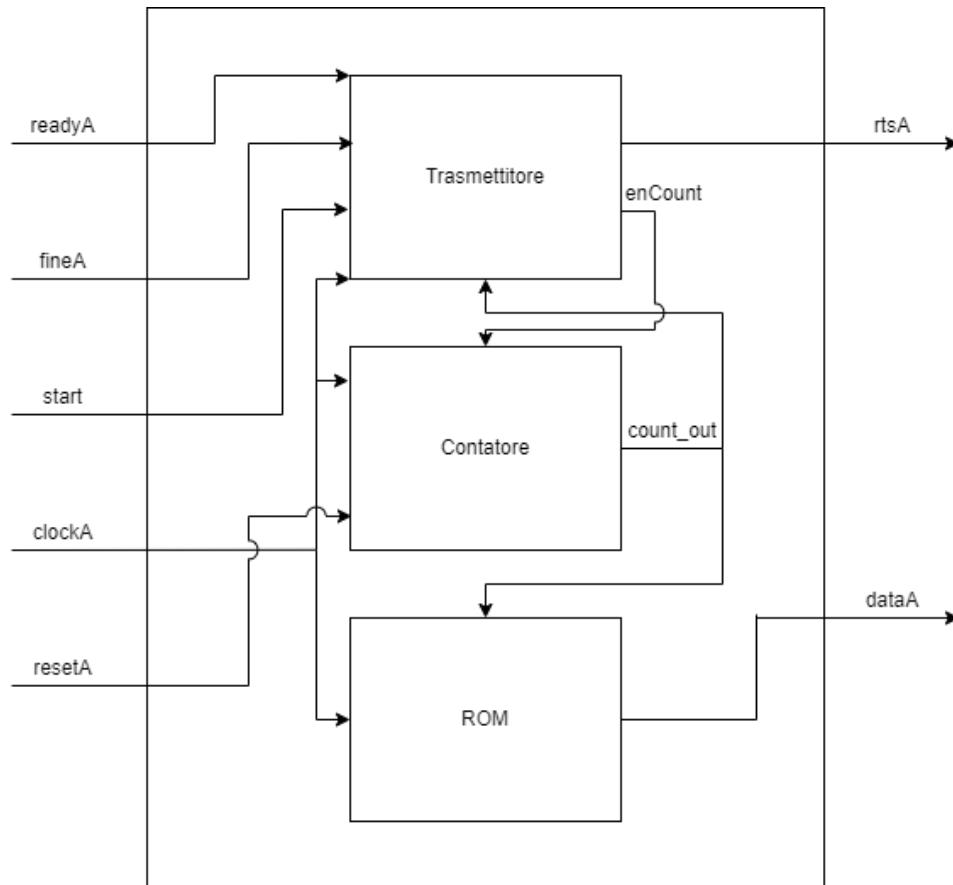


Fig. 7.1 Schema SystemA

Vediamo invece il **SystemB**. Esso presenta cinque componenti fondamentali: Ricevitore, Contatore, ROM\_B, ROM\_S e adder, come mostra la figura 7.2. In ingresso il blocco riceve il *clock*, il *reset*, il segnale *rtsB*, tramite la quale il trasmettitore dà il via alla trasmissione e il dato (*dataB*) di M bit. In uscita abbiamo i segnali *readyB* e *fineB*, i quali andranno al trasmettitore, e la stringa di M bit che rappresenta la somma (*sommaB*). Il blocco Ricevitore, descritto successivamente nello specifico, è stato sviluppato al fine di realizzare l'handshaking dalla parte del ricevitore e quindi l'automa descritto nel paragrafo precedente. Il Contatore è utilizzato per scandire la trasmissione delle stringhe e terminare la comunicazione. Inoltre, possiamo vedere la presenza delle due memorie, ROM\_B e ROM\_S, che contengono rispettivamente le N stringhe di M bit del ricevitore e quelle che risultano dal calcolo della somma effettuata dal componente adder.

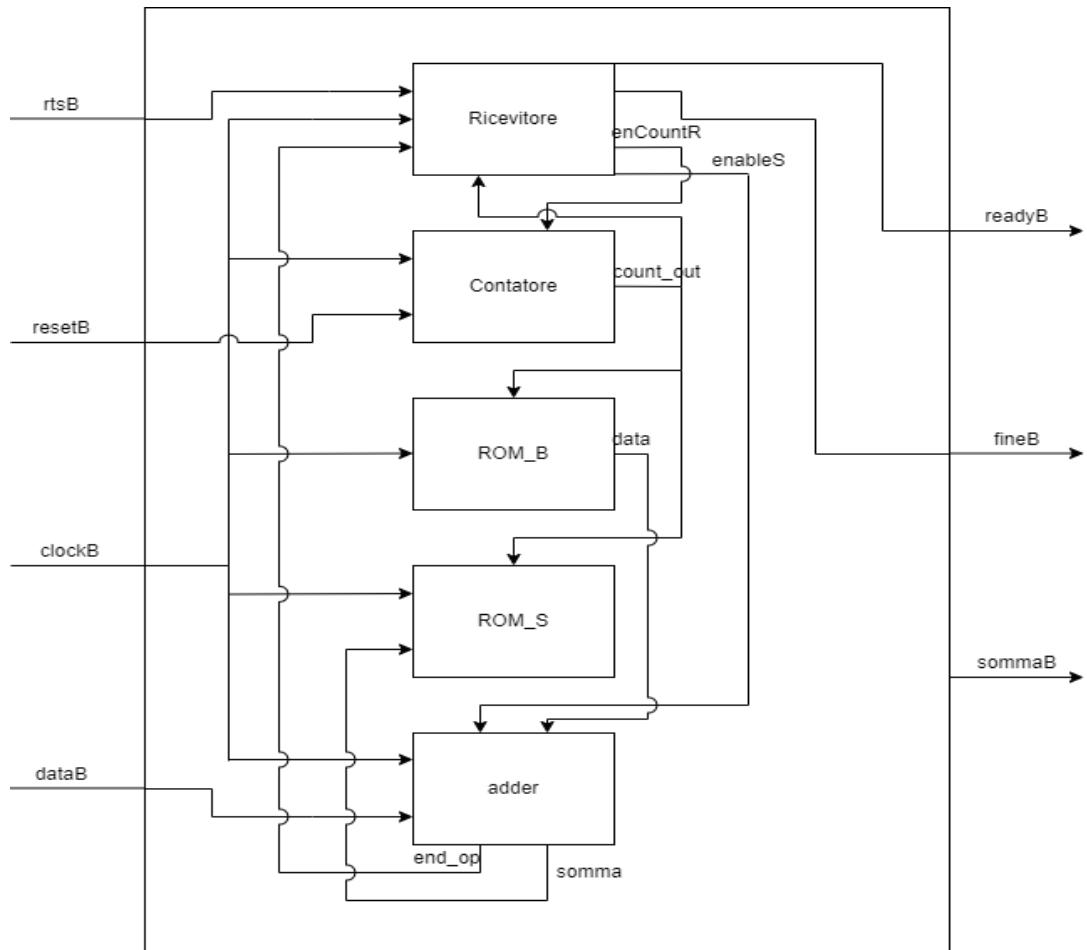


Fig. 7.2 Schema SystemB

L' **Hanshaking\_system** è stato realizzato mediante un approccio strutturale, ovvero componendo i sistemi descritti precedentemente, come mostra la figura 7.3. In ingresso abbiamo i segnali di *clock* e di *reset*, che vengono mandati ad entrambi i nodi e il segnale di *start*, che scandisce l'inizio della trasmissione, dato solo al primo. In uscita abbiamo la stringa di M bit (*y\_out*) che rappresenta la somma tra la stringa inviata dal trasmettitore e quella del ricevitore.

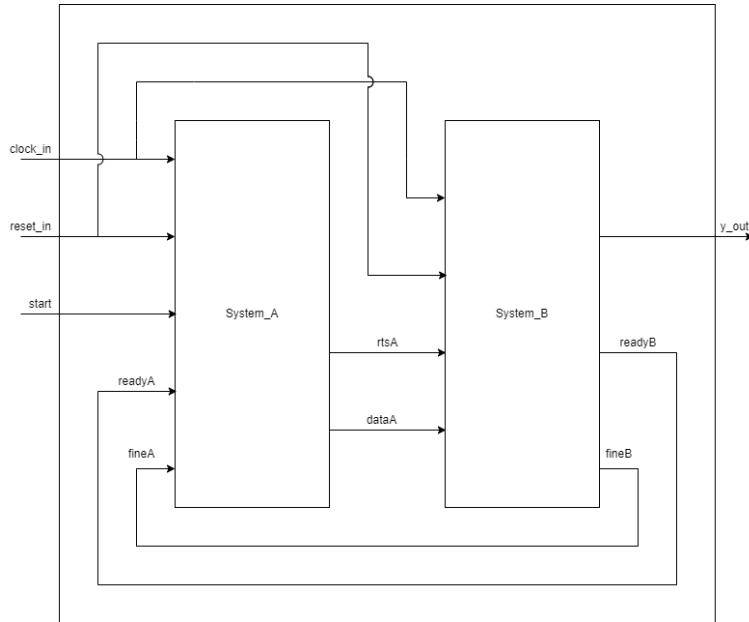


Fig. 7.3 Schema Handshaking\_system

### 7.3.4 Codice

Partiamo dall'analisi dei singoli blocchi del nodo A.

- **Trasmettitore**

```
entity Trasmettitore is
    generic(
        N : integer := 10;
        M : integer := 10
    );
    Port (
        clock: in std_logic;
        rts_out: out std_logic;
        count: in integer;
        ready_in: in std_logic;
        fine_in: in std_logic;
        enCount: out std_logic;
        start: in std_logic
    );
end Trasmettitore;

architecture Behavioral of Trasmettitore is

type stato is (q0,q1,q2,q3,q4);
signal stato_corrente : stato := q0;

begin
process(clock)
begin
    if rising_edge(clock) then
        case stato_corrente is
            when q0 =>
                if(start = '1') then
                    rts_out <= '1';
                    stato_corrente <= q1;
                else
                    stato_corrente <= q0;
                end if;
            when q1 =>
                if(ready_in = '1') then stato_corrente <= q2;
                else stato_corrente <= q1;
                end if;
            when q2 =>
                rts_out <= '0';
                enCount <= '1';
                stato_corrente <= q3;
            when q3 =>
                enCount <= '0';
                if(count < N-1 and fine_in = '1') then
                    stato_corrente <= q4;
                elsif( count = N-1 and fine_in = '1') then stato_corrente <= q0;
                else stato_corrente <= q3;
                end if;
            when q4 =>
                rts_out <= '1';
                stato_corrente <= q1;
        end case;
    end if;
end process;
end Behavioral;
```

Questo blocco è stato descritto a livello comportamentale, infatti l'architecture è realizzata da un process caratterizzato dalla presenza del clock nella sensitivity lists. In primis sono stati definiti gli stati. Successivamente, abbiamo implementato l'automa descritto precedentemente tramite il costrutto “when....case”. Infine, è stato utilizzato “generic”, i generici sono una forma locale di costante a cui può essere assegnato un valore quando istanziamo un componente . Possiamo chiamare lo stesso componente VHDL più volte e assegnare valori diversi al generico.

- **Contatore**

```

entity contatore is
  generic(
    N: integer := 2
  );
  Port (
    clk: in std_logic;
    rst: in std_logic;
    enable: in std_logic;
    count_out: out integer
  );
end contatore;

architecture Behavioral of contatore is

begin
process(clk,rst,enable)
variable count: integer := 0;
begin
  if(rst = '1') then
    count := 0;
  end if;
  if rising_edge(clk) then
    if(enable = '1') then
      if(count = N-1) then
        count := 0;
      else
        count := count +1;
      end if;
    end if;
  end if;
  count_out <= count;
end process;
end Behavioral;

```

Anche il contatore è stato descritto a livello comportamentale, con un process caratterizzato dalla presenza dei segnali clock, rst ed enable nella sensitivity lists. Il reset è sincrono e quando è alto la variabile di conteggio (count) viene posta a 0. Sul fronte di salita del clock, se il segnale di enable è alto, incrementiamo il contatore solo se non ha contato fino a N-1 (dato che parte da 0), in caso contrario la variabile di conteggio viene azzerata. Infine, l'ultima istruzione sequenziale che viene effettuata è l'assegnazione di count al segnale di uscita count\_out, di tipo integer.

- **ROM**

```

entity ROM is
  generic(
    N: integer := 10;
    M: integer := 3
  );
  Port (
    clock: in std_logic;
    count: in integer;
    data: out std_logic_vector(0 to M-1)
  );
end ROM;

architecture Behavioral of ROM is

type rom_type is array (N-1 downto 0) of std_logic_vector(0 to M-1);
signal ROM : rom_type := (
"101",
"000",
"011",
"111",
"001",
"001",
"101",
"010",
"011",
"001");

begin
process(clock,count)
begin
  if rising_edge(clock) then
    data <= ROM(count);
  end if;
end process;
end Behavioral;

```

La ROM è stata descritta a livello comportamentale con un process caratterizzato dalla presenza del clock nella sensitivity lists. Essa contiene N=10 stringhe da M=3 bit che verranno trasmesse al ricevitore per poi effettuare la somma. Presenta in ingresso il segnale count proveniente dal contatore che serve per definire quale delle stringhe bisogna mandare in uscita. Sul fronte di salita del clock si assegna al segnale di uscita (data) una delle stringhe della ROM.

Successivamente, secondo l'approccio strutturale abbiamo composto i vari blocchi realizzati così da ottenere il **SystemA**. Sono stati inoltre usati due segnali interni che modellano i cavi di collegamento tra i componenti.

```

entity System_A is
    generic(
        N : integer := 10;
        M : integer := 3
    );
    Port (
        clockA: in std_logic;
        resetA: in std_logic;
        rtsA: out std_logic;
        readyA: in std_logic;
        fineA: in std_logic;
        start: in std_logic;
        dataA: out std_logic_vector(0 to M-1)
    );
end System_A;

architecture Structural of System_A is

component Trasmettitore is
    generic(
        N : integer := 10;
        M : integer := 3
    );
    Port (
        clock: in std_logic;
        rts_out: out std_logic;
        count: in integer;
        ready_in: in std_logic;
        fine_in: in std_logic;
        enCount: out std_logic;
        start: in std_logic
    );
end component;
component contatore is
    generic(
        N: integer := 10
    );
    Port (
        clk: in std_logic;
        rst: in std_logic;
        enable: in std_logic;
        count_out: out integer
    );
end component;

```

```

component ROM is
    generic(
        N: integer := 10;
        M: integer := 3
    );
    Port (
        clock: in std_logic;
        count: in integer;
        data: out std_logic_vector(0 to M-1)
    );
end component;

signal conteggio: integer;
signal abilitazione: std_logic;

begin

T: Trasmettitore
    generic map(
        N => 10,
        M => 3
    )
    port map(
        clock => clockA,
        rts_out => rtsA,
        count => conteggio,
        ready_in => readyA,
        fine_in => fineA,
        enCount => abilitazione,
        start => start
    );

cont: contatore
    generic map(
        N => 10
    )
    port map(
        clk => clockA,
        rst => resetA,
        enable => abilitazione,
        count_out => conteggio
    );

ROM_A: ROM
    generic map(
        N => 10,
        M => 3
    )
    Port map(
        clock => clockA,
        count => conteggio,
        data => dataA
    );
end Structural;

```

Vediamo ora i singoli blocchi del nodo B.

- **Ricevitore**

```
entity Ricevitore is
    generic(
        N : integer := 10;
        M : integer := 10
    );
    Port (
        clk: in std_logic;
        rts_in : in std_logic;
        fine_out: out std_logic;
        ready_out: out std_logic;
        enCountR: out std_logic;
        enableS: out std_logic;
        endop: in std_logic;
        countR: in integer
    );
end Ricevitore;

architecture Behavioral of Ricevitore is

type stato is (q0,q1,q2,q3,q4);
signal stato_corrente : stato := q0;

begin
process(clk)
variable data_temp: integer := 0;
begin
    if rising_edge (clk) then
        case stato_corrente is
            when q0 =>
                if(rts_in = '1') then stato_corrente <= q1;
                else stato_corrente <= q0;
                end if;
            when q1 =>
                ready_out <= '1';
                enableS <= '1';
                stato_corrente <= q2;
            when q2 =>
                if(endop = '1') then
                    enableS <= '0';
                    ready_out <= '0';
                    enCountR <= '1';
                    stato_corrente <= q3;
                else
                    stato_corrente <= q2;
                end if;
            when q3 =>
                enCountR <= '0';
                if(countR < N-1) then
                    fine_out <= '1';
                    stato_corrente <= q4;
                else
                    fine_out <= '1';
                    stato_corrente <= q0;
                end if;
            when q4 =>
                if(rts_in = '1') then stato_corrente <= q1;
                else stato_corrente <= q4;
                end if;
        end case;
    end if;
end process;
end Behavioral;
```

Come per il trasmettitore, il blocco è stato descritto a livello comportamentale, infatti l'architecture è realizzata da un process caratterizzato dalla presenza del clock nella sensitivity lists. Sono stati definiti gli stati. Successivamente, abbiamo implementato l'automa descritto precedentemente tramite il costrutto “*when....case*”.

- **Contatore**

Per brevità non inseriamo il codice del contatore, dato che presenta lo stesso comportamento di quello del nodo A.

- **ROM\_B**

```
entity ROM_B is
    generic(
        N: integer := 10;
        M: integer := 3
    );
    Port (
        clock: in std_logic;
        count: in integer;
        data: out std_logic_vector(0 to M-1)
    );
end ROM_B;

architecture Behavioral of ROM_B is

type rom_type is array (N-1 downto 0) of std_logic_vector(0 to M-1);
signal ROM : rom_type := (
    "001",
    "100",
    "101",
    "010",
    "011",
    "110",
    "011",
    "001",
    "101",
    "000");

begin
process(clock,count)
begin
    if rising_edge(clock) then
        data <= ROM(count);
    end if;
end process;
end Behavioral;
```

La memoria del ricevitore presenta lo stesso comportamento di quella del nodo A. È stata inserita per visionare il contenuto al fine di controllare la correttezza del risultato della somma tra le stringhe.

- **ROM\_S**

Per brevità non è stato inserito il codice dato che presenta lo stesso comportamento delle memorie descritte precedentemente, con la sola differenza che inizialmente non contiene nulla. Successivamente, verrà riempita con i risultati della somma.

- **adder**

```

entity adder is
  generic(
    M: integer := 3
  );
  Port (
    clock: in std_logic;
    enable: in std_logic;
    data1: in std_logic_vector(0 to M-1);
    data2: in std_logic_vector(0 to M-1);
    somma: out std_logic_vector(0 to M-1);
    end_op: out std_logic
  );
end adder;

architecture Behavioral of adder is

begin
process(clock,enable)
variable data1_temp: integer := 0;
variable data2_temp: integer := 0;
variable somma_temp: integer;
begin
begin
  if rising_edge(clock) then
    if(enable = '1') then
      data1_temp := TO_INTEGER(unsigned(data1));
      data2_temp := TO_INTEGER(unsigned(data2));
      somma_temp := data1_temp + data2_temp;
      somma <= std_logic_vector(TO_UNSIGNED(somma_temp,M));
      end_op <= '1';
    end if;
  end if;
end process;
end Behavioral;

```

Il blocco adder è stato descritto a livello comportamentale con un process caratterizzato dalla presenza dei segnali di clock ed enable nella sensitivity lists. Sul fronte di salita del clock, se l'enable è alto effettua prima la conversione delle due stringhe in ingresso da std\_logic\_vector a intero, assegnando i valori a due variabili (data1\_temp, data2\_temp). Successivamente realizza la somma tra i due interi assegnando il valore alla variabile somma\_temp. Infine, effettua la conversione di quest'ultima variabile in std\_logic\_vector e alza il segnale end\_op per notificare il blocco ricevitore di aver completato le operazioni.

Successivamente, secondo l'approccio strutturale abbiamo composto i vari blocchi realizzati così da ottenere il **SystemB**. Sono stati inoltre usati sei segnali interni che modellano i cavi di collegamento tra i componenti.

```

entity System_B is
generic(
  N : integer := 10;
  M : integer := 10
);
Port (
  clockB: in std_logic;
  resetB: in std_logic;
  rtsB: in std_logic;
  readyB: out std_logic;
  fineB: out std_logic;
  dataB: in std_logic_vector(0 to M-1);
  sommaB: out std_logic_vector(0 to M-1)
);
end System_B;

```

```

architecture Structural of System_B is

signal en_temp: std_logic;
signal temp_count: integer;
signal enable_temp: std_logic;
signal data_temp: std_logic_vector(0 to M-1);
signal somma_temp: std_logic_vector(0 to M-1);
signal endop_temp: std_logic;

component Ricevitore is
generic(
    N : integer := 10;
    M : integer := 10
);
Port (
    clk: in std_logic;
    rts_in : in std_logic;
    fine_out: out std_logic;
    ready_out: out std_logic;
    enCountR: out std_logic;
    enableS: out std_logic;
    endop: in std_logic;
    countR: in integer
);
end component;

component contatore is
generic(
    N: integer := 10
);
Port (
    clk: in std_logic;
    rst: in std_logic;
    enable: in std_logic;
    count_out: out integer
);
end component;

component adder is
generic(
    M: integer := 3
);
Port (
    clock: in std_logic;
    enable: in std_logic;
    data1: in std_logic_vector(0 to M-1);
    data2: in std_logic_vector(0 to M-1);
    somma: out std_logic_vector(0 to M-1);
    end_op: out std_logic
);
end component;

component ROM_B is
generic(
    N: integer := 10;
    M: integer := 3
);
Port (
    clock: in std_logic;
    count: in integer;
    data: out std_logic_vector(0 to M-1)
);
end component;

```

```

component ROM_S is
    generic(
        N: integer := 10;
        M: integer := 3
    );
    Port (
        clock: in std_logic;
        count: in integer;
        data: in std_logic_vector(0 to M-1)
    );
end component;
begin
    begin
        R: Ricevitore
        generic map(
            N => 10,
            M => 3
        )
        Port map (
            clk => clockB,
            rts_in => rtsB,
            fine_out => fineB,
            ready_out => readyB,
            enCountR => en_temp,
            enableS => enable_temp,
            endop => endop_temp,
            countR => temp_count
        );
    Counter: contatore
    generic map (
        N => 10
    )
    Port map (
        clk => clockB,
        rst => resetB,
        enable => en_temp,
        count_out => temp_count
    );
    ADD: adder
    generic map(
        M => 3
    )
    Port map (
        clock => clockB,
        enable => enable_temp,
        data1 => dataB,
        data2 => data_temp,
        somma => somma_temp,
        end_op => endop_temp
    );
    ROM1: ROM_B
    generic map(
        N => 10,
        M => 3
    )
    Port map(
        clock => clockB,
        count => temp_count,
        data => data_temp
    );

```

```

ROM2: ROM_S
    generic map(
        N => 10,
        M => 3
    )
    Port map(
        clock => clockB,
        count => temp_count,
        data => somma_temp
    );
    sommaB <= somma_temp;
end Structural;

```

Infine, secondo l'approccio strutturale abbiamo composto i due nodi realizzati in modo tale da ottenere l'**Handshaking\_system**. Si può notare la presenza di quattro segnali interni per modellare il protocollo di handshaking tra i due nodi.

```

entity Handshaking_system is
    generic(
        M: integer := 3
    );
    Port (
        clock_in: in std_logic;
        reset_in: std_logic;
        start: in std_logic;
        Y_out: out std_logic_vector(0 to M-1)
    );
end Handshaking_system;

architecture Structural of Handshaking_system is

component System_A is
    generic(
        N : integer := 10;
        M : integer := 3
    );
    Port (
        clockA: in std_logic;
        resetA: in std_logic;
        rtsA: out std_logic;
        readyA: in std_logic;
        fineA: in std_logic;
        start: in std_logic;
        dataA: out std_logic_vector(0 to M-1)
    );
end component;
component System_B is
    generic(
        N : integer := 10;
        M : integer := 10
    );
    Port (
        clockB: in std_logic;
        resetB: in std_logic;
        rtsB: in std_logic;
        readyB: out std_logic;
        fineB: out std_logic;
        dataB: in std_logic_vector(0 to M-1);
        sommaB: out std_logic_vector(0 to M-1)
    );
end component;

```

```

signal rts_temp: std_logic;
signal ready_temp: std_logic;
signal fine_temp: std_logic;
signal data_temp: std_logic_vector(0 to M-1);

begin

A: System_A
generic map(
    N => 10,
    M => 3
)
port map(
    clockA => clock_in,
    resetA => reset_in,
    rtsA => rts_temp,
    readyA => ready_temp,
    fineA => fine_temp,
    start => start,
    dataA => data_temp
);
B: System_B
generic map(
    N => 10,
    M => 3
)
port map(
    clockB => clock_in,
    resetB => reset_in,
    rtsB => rts_temp,
    readyB => ready_temp,
    fineB => fine_temp,
    dataB => data_temp,
    sommaB => y_out
);
end Structural;

```

## 7.4 Simulazione

Per effettuare la simulazione è stato realizzato il testbench sottostante. Abbiamo fissato il periodo del clock con rispettivo process che definisce il comportamento del segnale stesso. Inoltre, sono stati determinati tutti i segnali responsabili della simulazione. Dopo 200 ns, viene alzato e poi successivamente abbassato il segnale di start, dando il via alla trasmissione.

```

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

architecture Behavioral of TB_hand is

component Handshaking_system is
generic(
    M: integer := 3
);
Port (
    clock_in: in std_logic;
    reset_in: std_logic;
    start: in std_logic;
    y_out: out std_logic_vector(0 to M-1)
);
end component;

```

```

signal clk: std_logic;
signal rst: std_logic;
signal start: std_logic := '0';
signal y: std_logic_vector(0 to 2);
constant CLK_period : time := 10 ns;

begin
  uut: Handshaking_system
    generic map(
      M => 3
    )
    port map(
      clock_in => clk,
      reset_in => rst,
      start => start,
      y_out => y
    );
  CLK_process :process
  begin
    clk <= '0';
    wait for CLK_period/2;
    clk <= '1';
    wait for CLK_period/2;
  end process;

  -- Stimulus process
  stim_proc: process
  begin
    wait for 100 ns;

    wait for CLK_period*10;

    start <= '1';

    wait for 50 ns;

    start <= '0';

    wait;
  end process;
end Behavioral;

```

Come è possibile vedere dalla figura 7.4, il sistema, una volta dato il segnale di start, inizia la trasmissione e presenta in uscita la somma tra le stringhe memorizzate nelle memorie dei due nodi. Dato che abbassiamo il segnale di start, effettuate le N=10 somme si arresta.

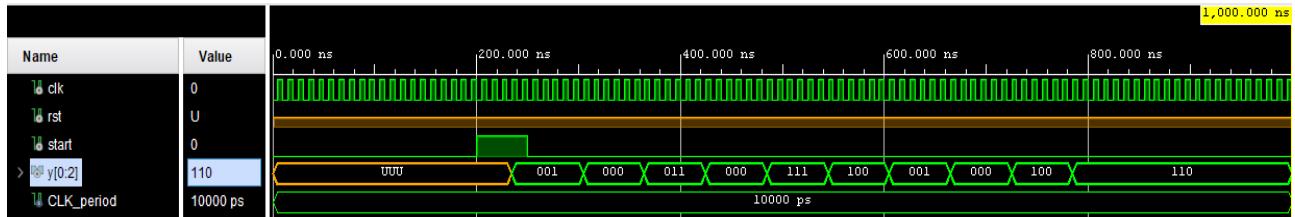


Fig. 7.4 Simulazione Handshaking\_system

# Capitolo 8

## 8. Il 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 input/output,
- d) (solo ove possibile) si sintetizzi il processore su FPGA.

### 8.2 Introduzione

Il processore MIC-1 prevede due distinte parti che comunicano tra loro al fine di implementare le microistruzioni necessarie all'esecuzione di una istruzione tratta dal programma assembler che viene scritto da un programmatore. I registri dell'architettura del processore visti dal programmatore sono solo un piccolo sottoinsieme di quelli realmente presenti nella microarchitettura: ogni istruzione assembler è un'entry point a una microsequenza che inizia da una microistruzione tratta dal control store della microarchitettura, e che procede finché la funzione dell'istruzione non è stata completamente esaurita.

### 8.3 Architettura a stack

Una particolarità di questo processore è quella di non disporre di registri generali; la sua architettura è infatti di tipo a stack: le sue istruzioni aritmetiche e logiche non hanno operandi esplicativi, ma li prelevano da una struttura LIFO allocata nella memoria principale, in cui devono essere posti in precedenza. Un'architettura a stack consente di poter memorizzare gli operandi da utilizzare nelle varie operazioni nello stack della macchina, che è un'area di memoria dedicata proprio a questa funzione. Per poter lavorare con un'architettura a stack è necessario disporre dei giusti registri per poterla indirizzare, ma anche per tenere traccia dell'operando in testa, due registri sono predisposti per questo compito: lo stack pointer (SP) e il top of stack(TOS), che ci permettono di accedere rispettivamente alla testa dello stack ed al suo valore.

### 8.4 Datapath

La parte operativa del processore MIC-1 è illustrata in figura 8.1. Sono presenti 10 registri e due bus, grazie ai quali è possibile far comunicare i registri. La ALU è un blocco che ottiene in ingresso dei bit di controllo per stabilire qual è l'operazione da effettuare, in particolare consente di fare operazioni logico\aritmetiche, illustrate in figura 8.2. I registri necessari alla comunicazione con la memoria è realizzata tramite coppie di registri, che caratterizzano due interfacce verso la memoria, le coppie sono MAR (Memory Address Register)-MDR(Memory Data Register) e PC(Program Counter)-MBR(Memory Byte Register), la prima coppia è dedicata ai dati, mentre la seconda alle istruzioni. L'interfaccia MAR-MDR mi consente di specificare l'indirizzo in memoria a partire dal quale leggere o scrivere, infatti l'MDR ha un collegamento bidirezionale con la memoria. PC ed MBR sono utilizzati per le istruzioni, l'MBR può solo leggere dalla memoria, dato che le istruzioni si possono solo leggere. La memoria di questa architettura, è organizzata in parole di 8 bit, ogni parola è identificata da un indirizzo univoco da 32 bit, quindi è possibile indirizzare all'interno della memoria

$2^{32}$  parole, c'è una leggera differenza tra l'indirizzamento tramite MAR e l'indirizzamento tramite PC, poiché quando il MAR indirizza qualcosa in memoria, trarrà a partire da questo indirizzo, i successivi 4 byte, ciò è possibile sfalsando il collegamento dei bit del registro con fili del bus che collega alla memoria, mentre tramite il PC è possibile leggere o scrivere dalla memoria un byte alla volta. Si presuppone che si abbia un cache hit ogni volta si acceda in memoria: con una tale supposizione, ciò significa che i dati sono disponibili al ciclo di clock immediatamente successivo. La comunicazione in fase di lettura prevede che al colpo di clock si pone l'indirizzo del dato(istruzione) nel MAR(nel PC nel caso di istruzioni) ed al colpo di clock successivo, nell'MDR ( nel MBR in caso di istruzioni) sarà presente il dato(o istruzione). Per la scrittura invece deve essere utilizzato un segnale di Write Enable (WE), dopo aver messo l'indirizzo nel MAR ed il dato nell'MDR, il meccanismo è illustrato in figura 8.3. Infine, uno degli alti registri principali è il registro H, usato per memorizzare temporaneamente uno degli operandi dell'ALU.

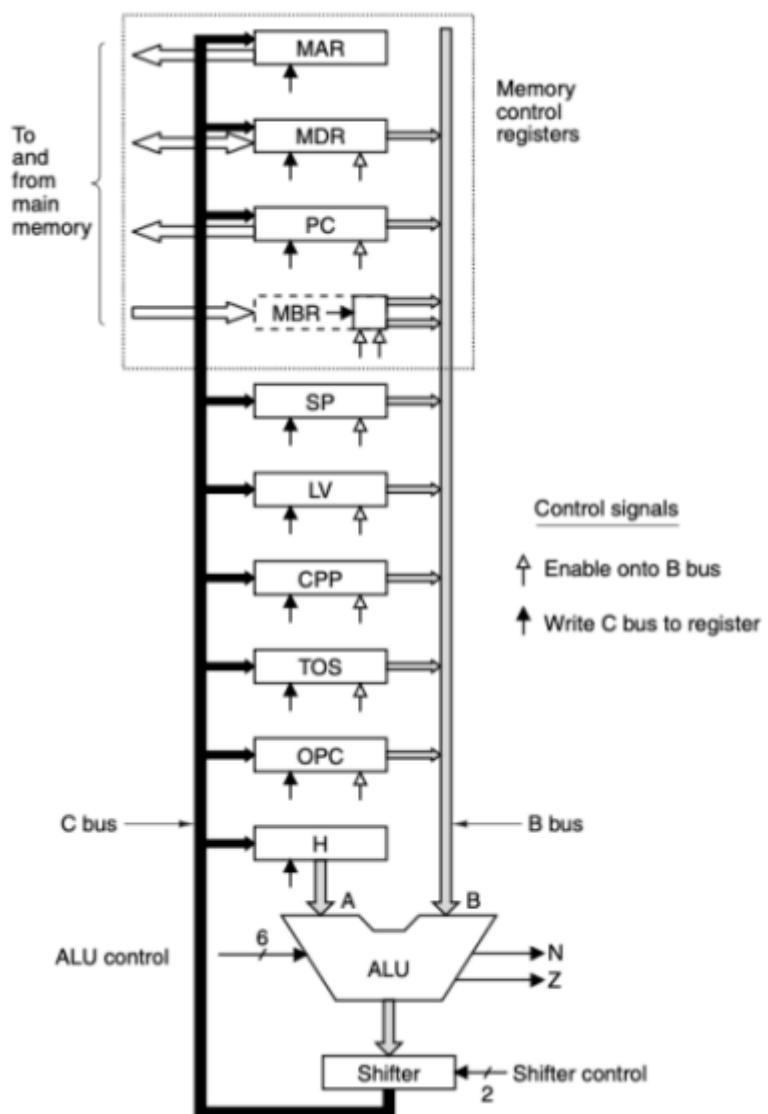


Figura 8.1: Unità operativa del MIC-1

<b>F<sub>0</sub></b>	<b>F<sub>1</sub></b>	<b>ENA</b>	<b>ENB</b>	<b>INVA</b>	<b>INC</b>	<b>Function</b>
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	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

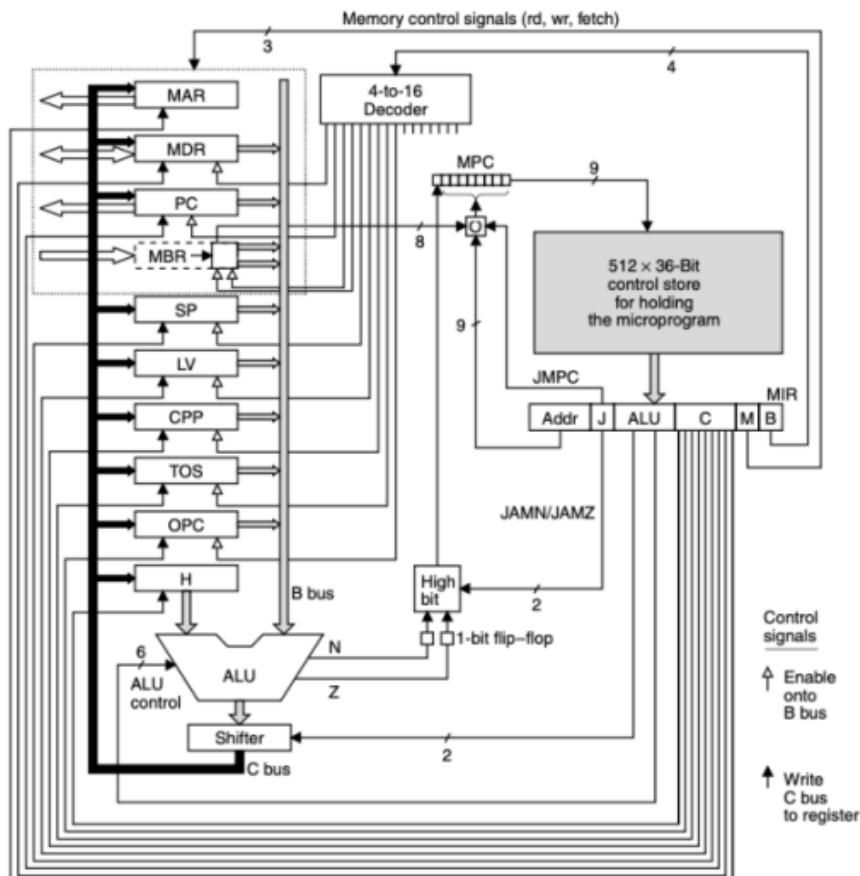
Figura 8.2: Segnali di controllo dell'ALU

## 8.5 Unità di Controllo

Sono necessari 29 (figura 8.3) segnali di abilitazione per il controllo del datapath:

- 9 segnali per la selezione del bus C
- 9 segnali per la selezione del bus B
- 8 segnali per la gestione dell'ALU
- 2 segnali di read\write per specificare se si vuole leggere o scrivere dalla memoria tramite il MAR\MDR
- 1 segnale per indicare il memory fetch dei valori presenti in PC\MBR.

I valori di tali segnali determinano le operazioni da eseguire durante un ciclo del processore.



L'unità di controllo del processore MIC-1 è realizzata in logica microprogrammata, quindi è composta di una micro-ROM e di una logica che la controlla, nella ROM sono memorizzate le microistruzioni necessarie allo svolgimento di ciascun codice operativo, cioè i segnali necessari all'unità operativa per svolgere ciascuna istruzione, l'unità di controllo andrà quindi a prelevare la sequenza di segnali di abilitazione necessari dalla micro-ROM. E' presente un generatore di indirizzi di partenza, ovvero l'indirizzo della micro-ROM a cui si deve accedere per eseguire una determinata operazione, inoltre è presente un Program Counter della micro-ROM(MPC). Le istruzioni ISA sono le istruzioni di interfaccia con il programmatore, a ciascuna di esse corrisponde una microprocedura, costituita da una serie di microistruzioni. Dato un codice operativo, bisogna determinare l'indirizzo di memoria da cui partire per operare con le microistruzioni, la tecnica del MIC-1 è di includere in ciascuna microistruzione un campo next-address, che indichi la localizzazione della prossima istruzione da eseguire. L'unità di controllo deve generare la sequenza di segnali di controllo necessari a pilotare l'unità operativa, cioè i 29 segnali descritti prima, ad ogni ciclo di clock, vengono aggiunti due campi per determinare cosa fare alla prossima istruzione, i campi: Addr e JAM. Il campo Addr (9 bit) serve ad individuare la microistruzione da eseguire al successivo ciclo di clock, mentre il campo JAM(3 bit) serve a gestire le operazioni di salto. La struttura della control word è illustrata di seguito(figura 8.4).

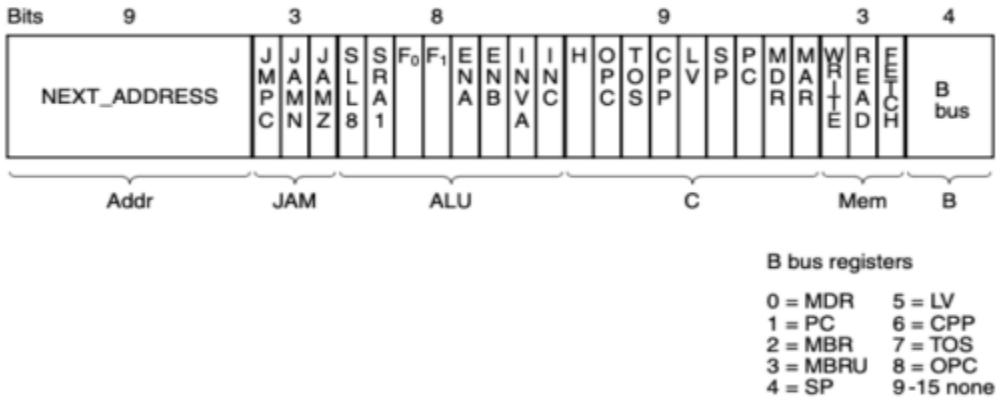


Figura 8.4: Struttura control word

- Addr: è costituito da 9 bit, concorre a stabilire il prossimo indirizzo da immettere nel MPC
- Jam: i tre bit di jam, specificano eventuali salti eccezionali, che alterano il flusso lineare dell'indirizzamento in memoria
- ALU: i bit di controllo rivolti alla ALU, con una combinazione di questi è possibile specificare l'operazione da fare effettuare alla ALU del datapath.
- C: i 9 bit consentono di abilitare i registri in lettura dal bus C
- Mem: specificano l'operazione da effettuare verso la memoria
- B: 4 bit che codificano l'abilitazione in mutua esclusione rivolta verso uno dei registri del datapath e scrivere sul bus B

Il segnale di controllo per il bus B, è codificato su 4 bit, è necessario aggiungere un decoder al fine di far arrivare l'informazione su 9 bit al bus B, con il vantaggio di non far viaggiare 9 fili lungo tutta la memoria, ma solo dal decoder in poi. E' possibile codificare l'informazione del segnale di controllo B, poiché i registri sul bus B sono selezionati in modo mutuamente esclusivo, mentre i registri sul bus C non accedono in modo necessariamente mutuamente esclusivo. Il campo next-address indica l'indirizzo della prossima micro-operazione, in particolare l'indirizzo della prima micro-operazione da eseguire per un determinato codice operativo è indicato dal codice operativo stesso, il quale è codificato con l'indirizzo della microistruzione della micro-ROM da cui partire, le successive

microistruzioni saranno calcolate a partire da questa. Quando invece è necessario effettuare un salto, il valore di next-address viene opportunamente modificato, prima di essere inserito nel PC. Per fare ciò si utilizzano i bit del campo JAM e i bit N e Z, proveniente dall'ALU, a seconda di tali valori copieremo nel MPC il next-address diversamente alterato:

- Se il campo JAM vale 000, allora next-address viene copiato senza modifiche
- Se JAMN è alto, si calcola l'OR tra il bit più significativo ed il flag N, se ne pone il risultato nel bit più significativo di MPC
- Se JAMZ è alto, si calcola l'OR con il flag Z e si pone il risultato nel bit più significativo di MPC
- Se JAMN e JAMZ sono entrambi alti, si calcola l'OR rispetto ad entrambi i flag.
- Se è alto il bit JMPC si effettua l'OR tra gli 8 bit di MBR e gli 8 bit meno significativi di next-address e il risultato viene posto in MPC

## 8.6 Flusso di controllo

### IADD:

IADD = 0x65:

```
MAR = SP = SP - 1; rd
H = TOS
MDR = TOS = MDR + H; wr; goto main
```

La prima microistruzione prevede il decremento dello stack pointer per poter leggere il secondo operando dell'operazione, in quanto verrà preso dalla posizione immediatamente inferiore alla testa dello stack), il secondo operando è memorizzato in MDR, in seguito al comando rd; nella prossima microistruzione, si memorizza il valore contenuto in TOS (primo operando) nel registro H, poiché TOS contiene il valore della precedente testa dello stack. Nell'ultima microistruzione, abbiamo il secondo operando memorizzato in MDR ed il primo memorizzato in H, sommiamo i due operandi ed inseriamo il risultato in MDR e TOS, specifichiamo l'operazione di write per memorizzare il risultato presente in MDR, nell'indirizzo puntato dal MAR. Riportiamo per le 3 microistruzioni appena descritte i bit specificati:

- 1)
  - ALU: 110110 (decremento operando B)
  - C: 000001001 (bus C scrive su SP e MAR)
  - Mem: 010 (lettura dati)
  - B: 0100 (SP controlla il bus B)
- 2)
  - ALU: 010100 (l'operando B passa invariato)
  - C: 100000000 (bus C scrive su H)
  - Mem: 000 (nessuna operazione di memoria)
  - B: 0111 (TOS controlla il bus B)
- 3)
  - ALU: 111100 (somma degli operandi A e B)
  - C: 001000010 (bus C scrive su MDR e TOS)
  - Mem: 100 (scrittura dati)
  - B: 0000 (MDR controlla il bus B)

**ISTORE:**

ISTORE = 0x36:

```
H = LV  
MAR = MBRU + H
```

ISTORE\_CONT:

```
MDR = TOS; wr  
SP = MAR = SP - 1; rd  
PC = PC + 1; fetch  
TOS = MDR; goto main
```

La prima microistruzione serve per inserire il contenuto di LV in H; la seconda microistruzione permette di aggiungere l'offset necessario a posizionarsi alla posizione corretta, dato che l'MBRU contiene il byte corrispondente all'indirizzo del PC; la terza microistruzione inserisce il valore del TOS in MDR e da il segnale di wr, quindi il valore presente nell'MDR sarà scritto all'indirizzo contenuto nel MAR; la quarta serve per eliminare il valore contenuto nel top stack, andando a decrementare lo stack pointer, inoltre dando il segnale di rd, troveremo in MDR il valore puntato dal MAR cioè la nuova testa dello stack; si effettua poi in anticipo il fetch, e viene aggiornato successivamente il TOS. Di seguito riportiamo i bit specificati per le microistruzioni della micro-procedura ISTORE:

1)

- ALU: 00010100
- C: 100000000
- Mem: 000
- B: 0101

2)

- ALU: 00111100
- C: 000000001
- Mem: 000
- B: 0011

3)

- ALU: 00010100
- C: 000000010
- Mem: 100
- B: 0111

4)

- ALU: 00110110
- C: 000001001
- Mem: 010
- B: 0100

5)

- ALU: 00110101
- C: 000000100
- Mem: 001
- B: 0001

6)

- ALU: 00010100
- C: 001000000
- Mem: 000
- B: 0000

## Modifica della microistruzione IADD

La modifica effettuata manipolando la microprocedura di IADD, fa in modo che la microprocedura IADD sia in grado di effettuare la somma di 3 operandi

IADD = 0x65:

```
MAR = SP = SP - 1; rd
H = TOS
H = MDR + H
MAR = SP = SP - 1; rd
empty
MDR = TOS = MDR + H; wr; goto main
```

La prima microistruzione è rimasta invariata, quindi come nell'istruzione IADD normale, viene decrementato lo stack pointer ed inserito nel MAR, in MDR troveremo il valore del secondo operando, in seguito al segnale rd; nella seconda microistruzione, si memorizza il valore contenuto in TOS (primo operando) nel registro H, poiché TOS contiene il valore della precedente testa dello stack; nella terza microistruzione si memorizza la somma del primo operando contenuto in H ed il secondo operando contenuto in MDR, quindi possiamo dire che in H viene memorizzata la somma parziale; la quarta microistruzione serve per decrementare lo stack pointer, in modo da puntare al terzo operando, viene dato il segnale di rd in modo da trovare il valore del terzo operando nel registro MDR; abbiamo inserito la microistruzione empty(equivalente a nop), per aspettare un ciclo di clock in più prima di procedere, dato che l'operazione di rd impiega due colpi di clock per presentare il valore in MDR; infine, l'ultima microistruzione fa la somma tra il terzo operando contenuto in MDR e la somma parziale (op1+op2) fatta in precedenza e memorizzata nel registro H, il segnale di wr serve per memorizzare la somma totale degli operandi nella locazione puntata dal MAR.

Consideriamo il seguente programma assembler, vengono caricate in memoria i tre operandi di cui si vuole effettuare la somma, viene effettuata la somma richiamando IADD ed infine viene memorizzato il risultato nella variabile a

```
.main
.var
    a
.endvar
    BIPUSH 0xA
    BIPUSH 0xE
    BIPUSH 0x1
    IADD
    ISTORE a
    HALT
.endmethod
```

La simulazione produce il seguente output:

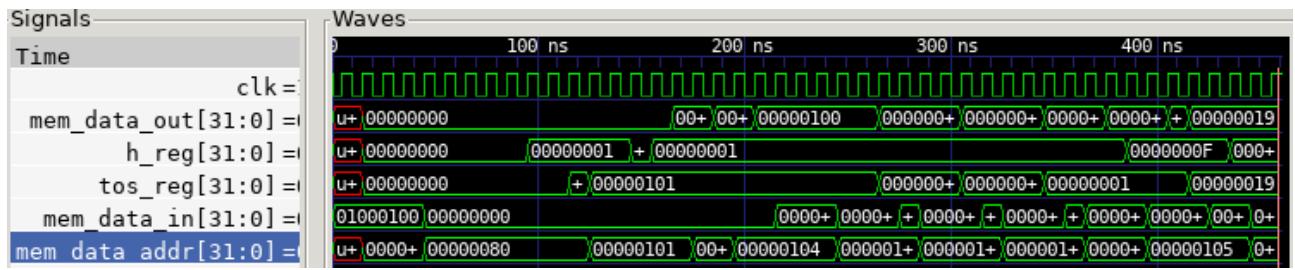


Figura 8.5: Simulazione del programma

# 9. Interfaccia Seriale

## 9.1 Traccia

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

### 9.1.1 Approfondimento sull'UART

#### 9.1.1.1 Comunicazione seriale

La comunicazione in parallelo seppur più semplice non sempre è possibile utilizzarla, questo perché essa necessita di un numero di fili pari al numero di dati da trasferire, di conseguenza sorge la necessità di fare trasmissioni seriali. Il problema principale della trasmissione seriale è la sincronizzazione, senza quest'ultima si rischia di non perdere dati. In particolare, tale ostacolo non interessa il trasmettitore, ma il ricevitore, che non sa quando inizia la trasmissione e non sa se un certo byte ha tutti i valori pari a 0 o tutti pari ad 1, non sapendo quindi cosa si sta trasmettendo sulla linea. Ci sono due tipologie di comunicazioni: asincrono e sincrona. Nell'asincrona, ci sincronizziamo per ogni carattere. La linea è in uno stato di riposo (attiva alta), prima di poter fare la comunicazione si abbassa per due quanti di tempo. È chiaro che il trasmettitore debba generare una forma d'onda che sia stata definita con il ricevitore. La linea è attiva alta poter distinguere un bit inviato pari ad 1 da uno pari a 0, successivamente viene dato un segnale di start abbassando la linea trasferendo quindi due bit pari a 0; creata la sequenza di bit opportuna, comincia la trasmissione dei dati, successivamente potrebbe esserci un bit di parità e poi uno di stop. Il ricevitore per assicurarsi di prendere tutti i bit deve sovra-campionare la linea, deve avere quindi una frequenza multipla rispetto al trasmettitore; ciò è dovuto ad eventuali accumuli di ritardi causati dallo sfasamento dei segnali di clock dei dispositivi interessati alla comunicazione; infatti, nonostante fossimo sicuro che i segnali di clock dei 2 sistemi viaggino alla stessa frequenza o a frequenze multiple (isocronia), non è detto che abbiano la stessa fase.

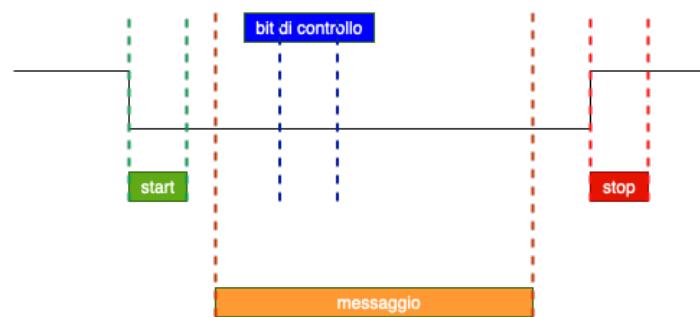


Figura 9.1: Sequenza della comunicazione asincrona

Gli errori possibili nella comunicazione seriale sono dovuti a vari fattori. A seguito di un trasferimento o di una completa ricezione, può accadere che il trasmettitore tenti di effettuare una successiva trasmissione prima che qualcuno abbia letto il contenuto del registro di recezione, ciò comporta quindi la cancellazione dell'attuale carattere e ovviamente la sua perdita, tale errore è detto **errore di overrun**. Un altro errore è quello di **parità**: se, esso avviene quando il numero di '1' presenti nel carattere, incluso il bit di parità, non è pari. Ci potrebbe essere l'errore anche se il bit di parità è corretto, cioè quando l'errore è su due bit o se proprio il bit di parità è stato alterato ad esempio

a causa di rumore sul canale. Un ulteriore errore è quello di **framing**: stabilita la dimensione del messaggio e il baud rate, abbiamo automaticamente anche stabilito il formato di messaggio che stiamo andando a trasmettere. Dati questi presupposti, è normale che ad un certo punto il ricevitore si aspetterà di ricevere un segnale alto di stop, che terminerà la ricezione. Può però capitare che così non sia, e dunque, qualora il frame ricevuto non fosse congruo alle specifiche stabilite a monte, si verificherà l'errore di framing.

Nel caso della comunicazione sincrona, la trasmissione funziona con il concetto di messaggio, non di carattere, per cui la sincronizzazione avviene ad ogni messaggio viene mandato prima un segnale di sync, poi un altro e poi una serie di caratteri fino a quando si hanno nuovamente due segnali di sync. Ci chiediamo il motivo per cui ci siano due protocolli e anche quali dei due sia più efficiente. C'è un problema di efficienza della trasmissione e di sincronismo. Quello asincrono è più semplice perché la sincronizzazione avviene per ogni carattere, mentre nel secondo caso avviene ad ogni messaggio, quindi, è un po' più complesso mantenere la sincronizzazione. Se introduciamo un fattore di efficienza, definito come il rapporto tra utile e spesa totale, se dobbiamo trasmettere 8 caratteri dovremmo inviarne 1 per lo stop, 1 per lo start, 8 bit di segnale e 1 per la parità, si ha quindi un rapporto di 8/11. Se invece calcoliamo l'efficienza della trasmissione sincrona, mandiamo 1000 byte, quindi  $100 \times 8$  bit, a fronte di questi mandiamo  $(100+4) \times 8$ , quindi l'efficienza è  $100/104$ . La comunicazione sincrona è più efficiente di quella asincrona per messaggi lunghi.

La comunicazione utilizzata nell'esercizio è asincrona.

### 9.1.1.2 Comunicazione tra dispositivi tramite interfaccia UART

La periferica UART prende informazioni in parallelo e le trasmette in seriale (trasmettitore), dall'altro lato il ricevitore prende le informazioni in seriale e mediante un registro a scorrimento le memorizza in parallelo. Le modalità di trasmissione possono essere di tre tipologie:

- Simplex: la comunicazione è unidirezionale, per cui solo il trasmettitore comunica con il ricevitore;
- Half-Duplex: trasmettitore e ricevitore comunicano sullo stesso canale, ma solo uno alla volta può parlare.
- Full-Duplex: trasmettitore e ricevitore comunicano sullo stesso canale e possono parlare contemporaneamente.

La nostra board monta la periferica UART in modalità full-duplex: in questa modalità di trasmissione, avendo la necessità di trasmettere e ricevere contemporaneamente, le strutture dati vengono duplicate. Il frame dati usato per la trasmissione tramite UART è quello mostrato in figura 9.2. È chiaro che i dispositivi devono essere d'accordo sulla struttura del pacchetto trasmesso. Nel nostro caso il pacchetto dati è caratterizzato da 10 bit, di cui uno è lo start bit, uno è lo stop bit e gli altri 8 sono i bit dati. I livelli tra start e stop devono essere diversi altrimenti non ci sarebbe alcuna transizione di stato mediante cui possiamo identificare la trasmissione di un bit diverso da quello trasmesso.

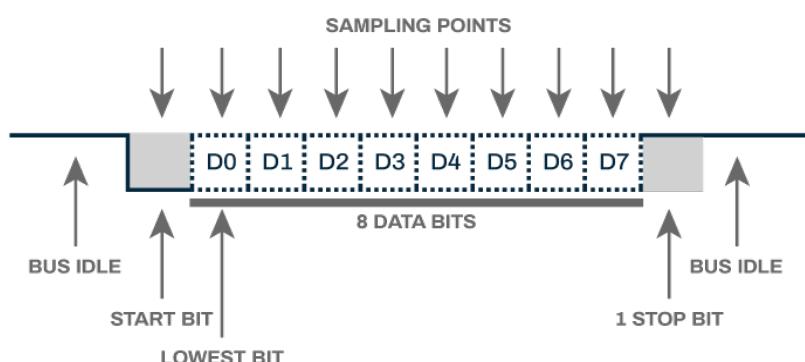


Figura 9.2: Frame dati usato per la trasmissione UART

Siccome UART comunicanti non hanno un clock condiviso, esse si risincronizzano ad ogni carattere trasmesso. Nonostante tra i due dispositivi venga definito un baud rate, numero di transizioni che

avvengono sulla linea (anche detto tasso di simbolo), il ricevitore lavora con una frequenza multipla (fattore 8 o 16) di quella del trasmettitore, per poter sovra-campionare la linea. Quando la comunicazione termina il trasmettitore alza il flag Text Buffer Empty, per indicare che il buffer è stato svuotato, mentre il ricevitore alza il flag Read Data Available, cioè la disponibilità in lettura delle informazioni ricevute.

Lo schema del componente UART utilizzato in questo esercizio è mostrato in figura 9.3 in cui i segnali RXD e TXD rispettivamente di input e output, rappresentano il canale di trasmissione. TXD è inizializzato ad 1, Questo accade perché non si vuole iniziare una ricezione spuria: affinché la linea sia sempre in stato inattiva alta, a meno che non si voglia iniziare una trasmissione, l'uscita TXD è sempre costante e pari ad 1, in modo che il ricevitore non possa vedere un valore diverso e non possa così iniziare una ricezione non prevista. Altri segnali di ingresso sono il reset RST, affinché la macchina possa partire da uno stato noto, il segnale di WR che serve per far partire il trasmettitore, i flag di uscita RDA, TBE, PE, FE, OE. Questi ultimi tre rappresentano proprio l'errore di parità, l'errore di framing e l'errore di overrun. Infine, si nota la presenza di un segnale di ingresso DBIN ad 8 bit ed un segnale di uscita DBOUT, anch'esso ad 8 bit. Il ricevitore riceve un byte di informazioni seriali attraverso la porta RXD di ingresso e converte queste in parallelo. Il byte convertito viene mostrato sull'uscita DBOUT. Dall'altro lato il trasmettitore prende un byte di informazioni parallele DBIN (quando si alza il bit di write) e lo trasmette in serie sulla porta TXD. Questo spiega ancora meglio perché RXD è un pin di ingresso e TXD di uscita.

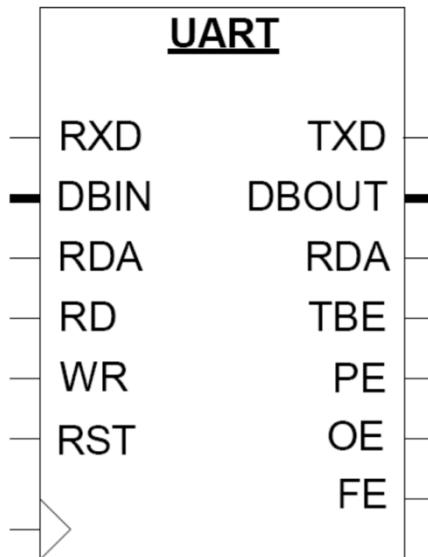


Figura 9.3: Blocco UART con pin di I/O

### 9.1.2 Schematici

Il sistema è stato realizzato tramite approccio strutturale, sfruttando l'UART dataci in dotazione. In particolare, si sono utilizzati due componenti UART: un componente è stato utilizzato nel sistema A sfruttando il trasmettitore al suo interno; il secondo invece è stato utilizzato nel sistema B sfruttando il ricevitore interno all'UART. Lo schema dei collegamenti è presentato in *figura 9.4*.

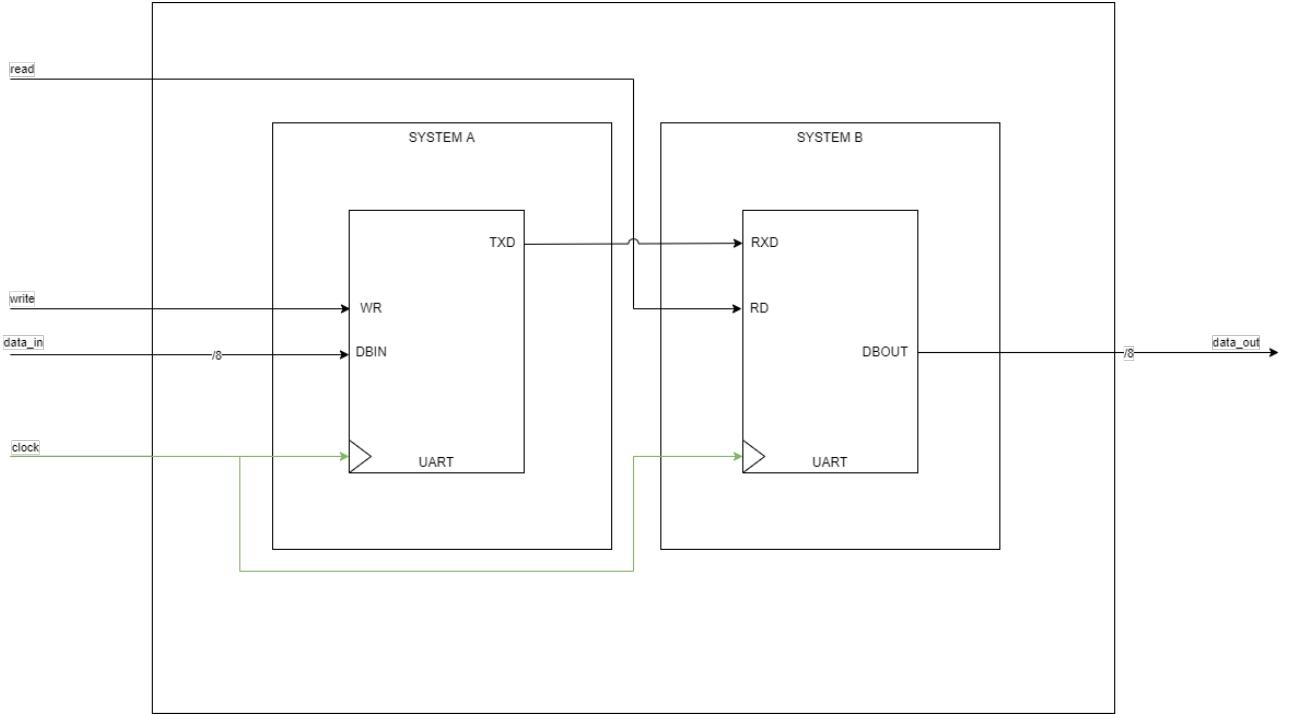


Figura 9.4: Schema Trasmissione seriale con UART

### 9.1.3 Codice

Come descritto precedentemente entrambi i sistemi utilizzano l'UART come si può vedere dal codice in basso

```

34  entity SystemA is
35    Port (
36      clock: in std_logic;
37      reset: in std_logic;
38      input: in std_logic_vector(7 downto 0);
39      write: in std_logic;
40      --read: in std_logic;
41      output: out std_logic
42    );
43  end SystemA;
44
45  architecture Structural of SystemA is
46
47  component UARTcomponent is
48    Generic (
49      --48MHz
50      -- BAUD_DIVIDE_G : integer := 26;  --115200 baud
51      -- BAUD_RATE_G   : integer := 417
52
53      --@26.6MHz
54      BAUD_DIVIDE_G : integer := 14;  --115200 baud
55      BAUD_RATE_G   : integer := 231
56    );
57    Port (
58      TXD    : out std_logic  := '1';
59      RXD    : in std_logic;
60      CLK    : in std_logic;
61      DBIN   : in std_logic_vector (7 downto 0);
62      DBOUT  : out std_logic_vector (7 downto 0);
63      RDA   : inout std_logic;
64      TBE   : out std_logic  := '1';
65      RD    : in std_logic;
66      WR    : in std_logic;
67      PE    : out std_logic;
68      FE    : out std_logic;
69      OE    : out std_logic;
70      RST   : in std_logic  := '0';
71  end component;
72
73  signal rdat: std_logic;
74  signal peT: std_logic;
75  signal feT: std_logic;
76  signal oeT: std_logic;
77  signal tbet: std_logic;
78  signal dbout: std_logic_vector(7 downto 0) := "00000000";
79  signal rxdt: std_logic;
80  signal rdT: std_logic;
81
82  begin
83    begin
84      uart: UARTcomponent
85        generic map(
86          BAUD_DIVIDE_G => 14,    --115200 baud
87          BAUD_RATE_G => 231
88        )
89        port map(
90          TXD => output,
91          RXD => rxdt,
92          CLK => clock,
93          DBIN => input,
94          DBOUT => dboutT,
95          RDA => rdat,
96          TBE => tbet,
97          RD => rdT,
98          WR => write,
99          PE => peT,
100         FE => feT,
101         OE => oeT,
102         RST => reset
103       );
104
105

```

Figura 9.4: Sistema A

Nell'unità principale il dato parallelo in ingresso è stato collegato a degli switch, mentre le uscite a dei led. Inoltre, vengono utilizzati due switch per assegnare un valore ai flag di write e read.

```

34  entity SystemB is
35  Port (
36    clk: in std_logic;
37    rst: in std_logic;
38    rxDR: in std_logic;
39    dboutR: out std_logic_vector(7 downto 0);
40    rdR: in std_logic
41  );
42 end SystemB;
43
44 architecture Structural of SystemB is
45
46 component UARTcomponent is
47   Generic (
48     --@48MHz
49     BAUD_DIVIDE_G : integer := 26; --115200 baud
50     BAUD_RATE_G : integer := 417
51
52     --@26.6MHz
53     BAUD_DIVIDE_G : integer := 14; --115200 baud
54     BAUD_RATE_G : integer := 231
55   );
56   Port (
57     TXD : out std_logic := '1';
58     RXD : in std_logic;
59     CLK : in std_logic;
60     DBIN : in std_logic_vector (7 downto 0);
61     DBOUT : out std_logic_vector (7 downto 0);
62     RDA : inout std_logic;
63     TBE : out std_logic := '1';
64     RD : in std_logic;
65     WR : in std_logic;
66     PE : out std_logic;
67     FE : out std_logic;
68     OE : out std_logic;
69     RST : in std_logic := '0';
70   end component;
71   signal rdaR: std_logic;
72   signal peR: std_logic;
73   signal feR: std_logic;
74   signal oeR: std_logic;
75   signal tbeR: std_logic;
76   signal txDR: std_logic;
77   signal dbinR: std_logic_vector(7 downto 0);
78   signal wr: std_logic;
79
80
81 begin
82   uart: UARTcomponent
83   generic map(
84     BAUD_DIVIDE_G => 14, --115200 baud
85     BAUD_RATE_G => 231
86   )
87   port map(
88     TXD => txDR,
89     RXD => rxDR,
90     CLK => clk,
91     DBIN => dbinR,
92     DBOUT => dboutR,
93
94     RDA => rdaR,
95     TBE => tbeR,
96     RD => rdR,
97     WR => wr,
98     PE => peR,
99     FE => feR,
100    OE => oeR,
101    RST => rst
102  );
103
104 end Structural;

```

Figura 9.5: Sistema B

## 9.2 Traccia

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

### 9.2.1 Soluzione

La traccia è stata svolta utilizzando un approccio strutturale. In primis sono stati realizzati mediante uno sviluppo comportamentale le entità Trasmettitore e Ricevitore, in quanto quest'ultime compongono i due sistemi principali, ovvero A e B, insieme all'interfaccia UART dataci in dotazione. Infine, è stato realizzato un unico sistema contenente i due sottosistemi A e B.

### 9.2.2 Schematici

#### Trasmettitore

Il componente Trasmettitore è l'entità che utilizza l'UART, gestendo i dati da inviare e i segnali di ingresso dell'UART per il trasferimento. Tale unità è stata descritta tramite l'utilizzo di un ASF (Automa a Stati Finiti) composto da 5 stati (*figura 9.4*).

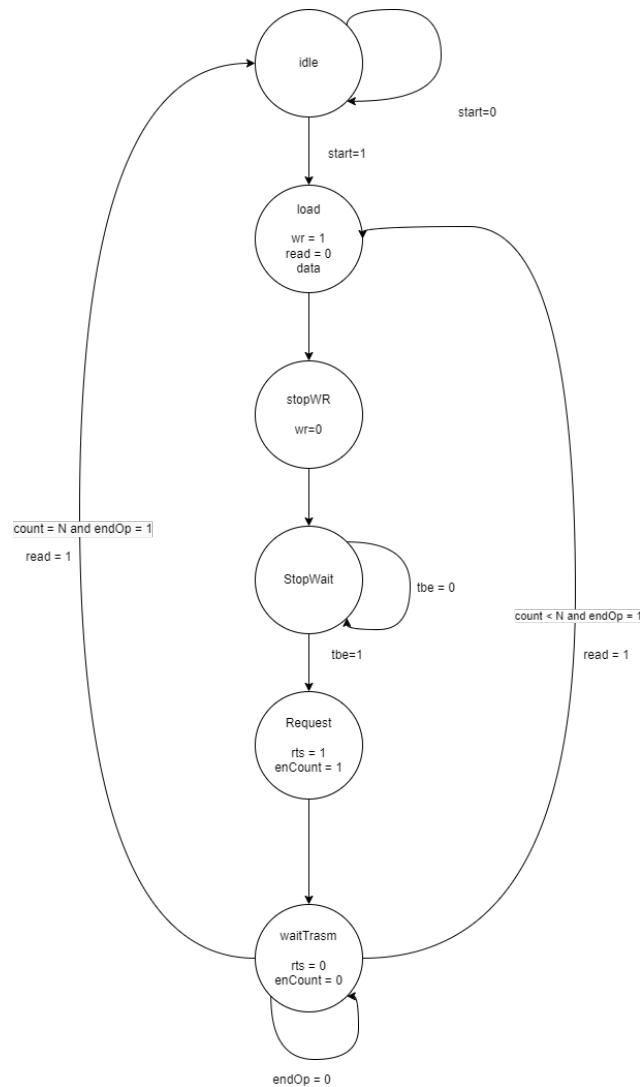


Figura 9.5: Automa del Trasmettitore

Vediamo ora in dettaglio le funzionalità dei diversi stati:

- **Idle**

Nello stato **idle** l'automa pone tutte le abilitazioni a 0 e resta in attesa del segnale di start di inizio trasmissione. Ricevuto tale segnale transita nello stato **load**.

- **Load**

Nello stato **load** il trasmettitore tiene tutte le abilitazioni basse ad esclusione del segnale di write dell'unità UART, ponendolo ad 1. Preleva i dati dalla ROM e li pone in ingresso al pin DBIN dell'UART, dopodiché transita nello stato **stopWR**.

- **stopWR**

Lo stato **stopWR** è uno stato di attesa in cui il segnale di write viene abbassato, il successivo stato è **StopWait**.

- **StopWait**

Lo stato **StopWait** è un ulteriore stato di attesa, in cui il trasmettitore attende che il buffer di trasferimento dell'UART sia di nuovo disponibile. Tale controllo viene effettuata tramite il flag TBE in uscita dall'UART. Nel momento in cui TBE ritorna ad essere 1 transita nello stato **Request**.

- **Request**

In questo stato il trasmettitore invia un segnale di richiesta di trasmissione (rts) al systemB. Abilita inoltre il contatore per predisporsi alla successiva trasmissione. Transita infine nello stato **waitTrasm**.

- **waitTrasm**

Nello stato **waitTrasm** il Trasmettitore attende un segnale di fine operazione dal systemB. Ricevuto tale segnale esso verifica se il valore del contatore è N-1 (in quanto il contatore parte da 0), in altre parole verifica se la trasmissione di tutte le stringhe di bit è stata effettuata, se così fosse esso torna allo stato **idle** attendendo un nuovo segnale di start, in caso contrario ritorna allo stato **Load** per il trasferimento della prossima stringa di bit. Infine, alza il segnale di read per indicare l'avvenuta lettura da parte del System B del dato.

## System A

Il sistema A (figura 9.5) è composto da 3 unità: l'unità trasmettitore, un contatore modulo N e l'UART. Il sistema A ha 3 ingressi: clock, segnale di start ovvero il segnale di inizio trasmissione e il segnale di fine trasmissione ricevuto dal sistema B. In uscita invece il sistema A presenta il dato serializzato, il segnale di richiesta di trasmissione e un segnale di read.

All'interno del sistema il trasmettitore presenta in output il dato prelevato dalla ROM, trasmettendolo in parallelo in ingresso all'UART, più precisamente esso viene presentato sul pin DBIN. Inoltre, il trasmettitore gestisce anche il segnale di write (wr) in ingresso all'UART. Il contatore modulo N viene abilitato tramite un segnale proveniente dal trasmettitore; invece, l'output del contatore rappresenta la posizione corrente nella memoria del trasmettitore, detto in altre parole, esso è indicata quale locazione di memoria deve essere trasmessa al ricevitore.

L'UART riceve in parallelo il dato da trasmettere, quest'ultimo viene presentato al suo componente trasmettitore (separato dal componente ricevitore) il quale dopo aver ricevuto il segnale di write carica il registro a scorrimento per il trasferimento e abbassa il segnale TBE per indicare la non disponibilità del bus di

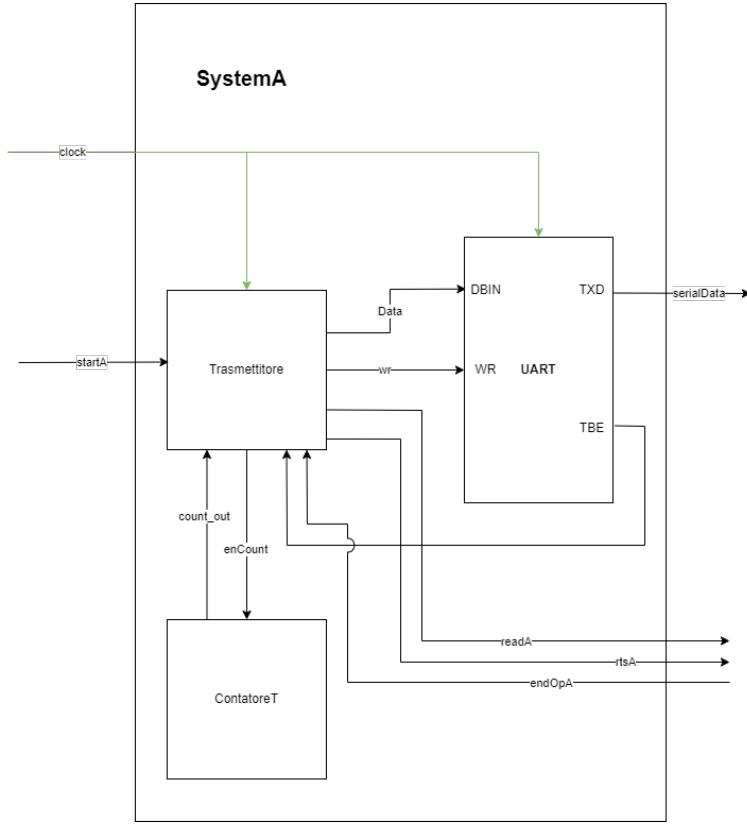


Figura 9.5: Schema sistema A

trasferimento. Possiamo notare come il segnale di Transfer va in ingresso al trasmettitore, in quanto quest'ultimo aspetta nello stato q2 che la trasmissione sia conclusa e che quindi il buffer torni disponibile prima di proseguire.

### Ricevitore

Il Ricevitore è l'unità del sistema B che si occupa di memorizzare i dati trasmessi dal System A e presentarli in output. Esso è stato descritto mediante un automa a stati finiti in 3 stati (*figura 9.6*).

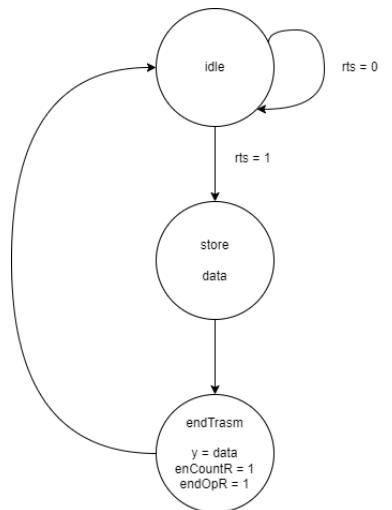


Figura 9.6: Automa Ricevitore

Vediamo ora in dettaglio ogni singolo stato:

- **idle**

Nello stato **idle** il sistema abbassa tutti i segnali e attende il segnale rts dal sistema A. Ricevuto tale segnale transita nello stato **store**.

- **store**

In questo stato il Ricevitore memorizza il dato ricevuto dall'unità UART. Transita nello stato **endTrasm**.

- **endTrasm**

Nello stato corrente il dato memorizzato viene presentato in uscita, viene alzata l'abilitazione del contatore e il flag di fine operazione (rispettivamente enCountR e endOpR). Infine transita nello stato **idle** attendendo il segnale rts.

## System B

Il sistema B (figura 9.7) è composto da 4 unità: l'unità ricevitore, un contatore modulo N, una memoria e l'UART. Il sistema B ha 5 ingressi: clock, segnale di richiesta di trasmissione, il dato serializzato, il segnale di read e un segnale utilizzato per scorrere la memoria ricevuto tramite un pulsante. In uscita invece il sistema presenta il dato parallelo e il segnale di fine operazione.

Due dei 5 ingressi sono collegati all'UART, quest'ultima riceve il segnale serializzato nella porta RXD e il segnale di read (RD) dal sistema A. Nel momento in cui l'UART riceve il segnale di start inizia a catturare i bit trasmessi dal sistema A sulla porta RXD, a fine trasmissione, a patto che non ci siano errori, il dato ormai parallelizzato viene memorizzato in un registro attivo a contenere una stringa di 8 bit, assegnato pari all'uscita DBOUT collegata all'unità Ricevitore. Il Ricevitore oltre a ricevere il dato ricevuto in ingresso dall'UART anche il segnale RDA, utilizzato per verificare la disponibilità o meno del dato in parallelo.

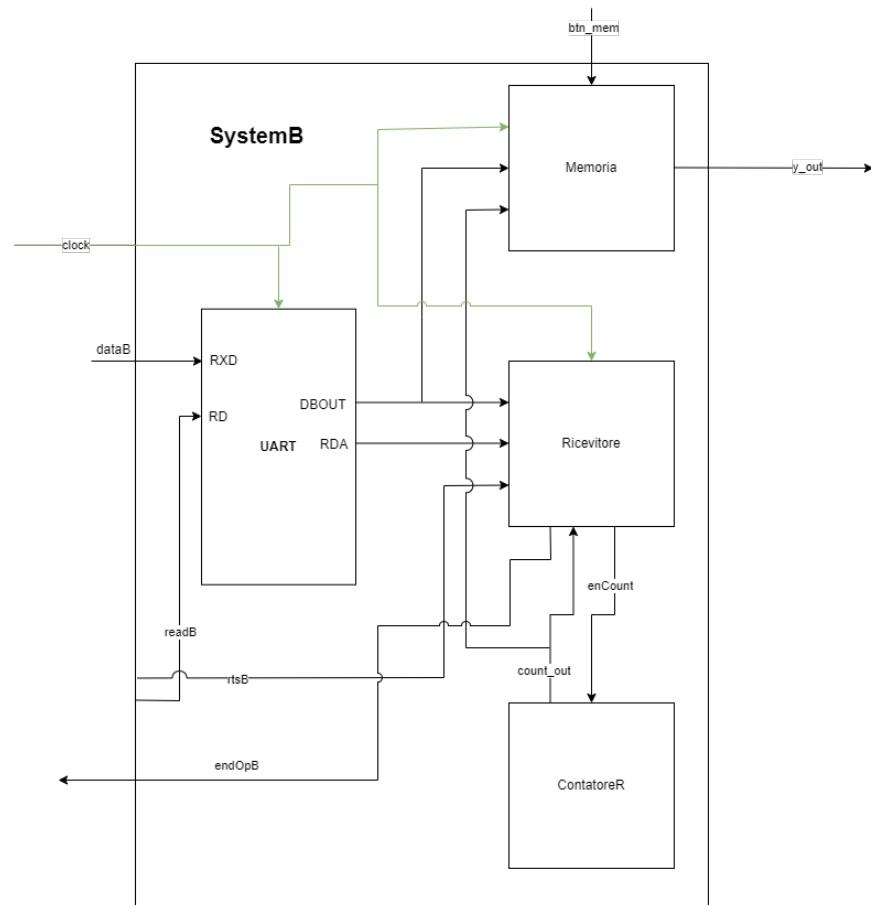


Figura 9.7: Schema sistema B

Il Ricevitore salva il dato ricevuto dall'UART nella memoria, ad ogni memorizzazione il contatore viene incrementato tramite l'abilitazione data dal Ricevitore e tale valore incrementato viene utilizzato per accedere alla corretta locazione di memoria in cui verrà memorizzato il dato.

Infine, tramite il pulsante il dato viene prelevato dalla memoria e presentato in uscita, ad ogni pressione viene visualizzato il valore contenuto nella locazione di memoria successiva a quella corrente.

### 9.2.3 Codice

#### Trasmettitore

Nella sezione “schematici” abbiamo mostrato lo schema dell'automa che descrive il componente, tale automa è stato implementato in VHDL nel seguente modo:

Dapprima abbiamo definito l'entità Trasmettitore, definendo i rispettivi segnali di ingresso e uscita (*figura 9.8*)

```

34 entity Trasmettitore is
35   generic (
36     N: integer := 10
37   );
38   Port (
39     clock: in std_logic;
40     start: in std_logic;
41     wr: out std_logic;
42     tbe: in std_logic;
43     read: out std_logic := '1';
44     endOp: in std_logic := '0';
45     rtsT: out std_logic;
46     enCount: out std_logic;
47     countT: in integer;
48     data: out std_logic_vector(7 downto 0)
49   );
50 end Trasmettitore;
51
52 architecture Behavioral of Trasmettitore is
53
54   type rom_type is array (N-1 downto 0) of std_logic_vector(7 downto 0);
55   signal ROM : rom_type := (
56     "01101011",
57     "01101010",
58     "01101001",
59     "01101110",
60     "01101111",
61     "01101000",
62     "01101111",
63     "01000000",
64     "00010101",
65     "00000001");
66
67   type stato is (idle,load,stopWR,StopWait,Request, waitTrasm);
68   signal stato_corrente: stato := idle;
69
70   signal last_start: std_logic := '0';

```

*Figura 9.8: VHDL Trasmettitore*

Il componente possiede una ROM interna, definita come vettore di segnali generico, contenente le stringhe di bit da trasmettere. È stato definito un nuovo tipo, che assume specifici valori e rappresenta gli stati della macchina. Successivamente si è proceduti con la descrizione dell'ASF.

Ogni stato tiene traccia del valore delle abilitazioni, tenendole basse o alte. Nell'ultimo stato viene effettuato un controllo sul valore del contatore.

```

72 begin
73 process(clock)
74 begin
75 if rising_edge(clock) then
76 case stato_corrente is
77 when idle =>
78 wr <= '0';
79 read <= '0';
80 rtsT <= '0';
81 enCount <= '0';
82 if(start = '1' and last_start = '0') then
83 last_start <= '1';
84 stato_corrente <= load;
85 else
86 last_start <= '0';
87 stato_corrente <= idle;
88 end if;
89 when load =>
90 wr <= '1';
91 read <= '0';
92 rtsT <= '0';
93 enCount <= '0';
94 data <= ROM(countT);
95 stato_corrente <= stopWR;
96 when stopWR =>
97 read <= '0';
98 rtsT <= '0';
99 enCount <= '0';
100 wr <= '0';
101 stato_corrente <= StopWait;
102 when StopWait =>
103 read <= '0';
104 rtsT <= '0';
105 enCount <= '0';
106 wr <= '0';
107
100 enCount <= '0';
101 wr <= '0';
102 stato_corrente <= StopWait;
103 when StopWait =>
104 read <= '0';
105 rtsT <= '0';
106 enCount <= '0';
107 wr <= '0';
108 if(tbe = '1') then
109 stato_corrente <= Request;
110 else
111 stato_corrente <= StopWait;
112 end if;
113 when Request =>
114 rtsT <= '1';
115 enCount <= '1';
116 wr <= '0';
117 read <= '0';
118 stato_corrente <= waitTrasm;
119 when waitTrasm =>
120 rtsT <= '0';
121 enCount <= '0';
122 wr <= '0';
123 if(countT < N-1 and endOp = '1') then
124 read <= '1';
125 stato_corrente <= load;
126 elsif(countT = N-1 and endOp = '1') then
127 read <= '1';
128 stato_corrente <= idle;
129 else
130 stato_corrente <= waitTrasm;
131 end if;
132 end case;
133 end if;
134 end process;
135 end Behavioral;

```

## Contatore

Il contatore, utilizzato sia dal Trasmettitore che dal Ricevitore è stato realizzato mediante un approccio comportamentale. Al fronte di salita del clock il contatore controlla se l'abilitazione è alta, se così fosse allora incrementa il valore di 1 altrimenti lo lascia invariato. Inoltre, esso effettua un controllo sul valore ogni volta l'abilitazione diviene alta (sempre sul fronte di salita del clock): nel caso in cui il valore ‘count’, ovvero, il valore presentato in uscita raggiunge N-1 (conteggio massimo, notiamo che count parte da 0) esso setta la variabile a 0. Il componente è stato reso generico, di conseguenza il suo conteggio massimo viene deciso al momento della sua istanziazione.

```

34 entity contatore is
35 generic(
36 N: integer := 2
37 );
38 Port (
39 clk: in std_logic;
40 rst: in std_logic;
41 enable: in std_logic;
42 count_out: out integer
43 );
44 end contatore;
45
46 architecture Behavioral of contatore is
47
48 signal ty: std_logic;
49 signal last_ensem: std_logic;
50 signal last_clk: std_logic;
51
52 begin
53 process(clk,rst,enable)
54 variable count: integer := 0;
55 begin
56 if(rst = '1') then
57 count := 0;
58 end if;
59 if falling_edge(clk) then
60 if(enable = '1') then
61 if(count = N-1) then
62 count := 0;
63 else
64 count := count +1;
65 end if;
66 end if;
67 end if;
68 count_out <= count;
69 end process;
70 end Behavioral;

```

## System A

Il sistema A è stato realizzato tramite approccio strutturale. Dapprima si è descritta l'entità **System A**.

```

34  entity SystemA is
35      Port(
36          clockA: in std_logic;
37          resetA: in std_logic;
38          readA: out std_logic;
39          endOpA: in std_logic;
40          startA: in std_logic;
41          rqtst: out std_logic;
42          serialData: out std_logic
43      );
44  end SystemA;

```

Successivamente sono stati definiti i componenti ad esso appartenenti.

```

46  architecture Structural of SystemA is
47
48  component UARTcomponent is
49      Generic (
50          --44MHz
51          BAUD_DIVIDE_G : integer := 26; --115200 baud
52          BAUD_RATE_G : integer := 417
53      );
54      --26.6MHz
55      BAUD_DIVIDE_G : integer := 14; --115200 baud
56      BAUD_RATE_G : integer := 231
57  );
58  Port (
59      TXD : out std_logic := '1'; -- Transmitted serial data output
60      RXD : in std_logic; -- Received serial data input
61      CLK : in std_logic; -- Clock signal
62      DBIN : in std_logic_vector (7 downto 0); -- Input parallel data to be transmitted
63      DBOUT : out std_logic_vector (7 downto 0); -- Received parallel data output
64      RDA : inout std_logic; -- Read Data Available
65      TBE : out std_logic := '1'; -- Transfer Buffer Empty
66      RD : in std_logic; -- Read Strobe
67      WR : in std_logic; -- Write Strobe
68      PE : out std_logic; -- Parity error
69      FE : out std_logic; -- Frame error
70      OE : out std_logic; -- Overwrite error
71      RST : in std_logic := "0"; -- Reset signal
72  end component;
73
139      OE => oeT,
140      RST => resetA
141  );
142
143  T: Trasmettitore
144      generic map (
145          N => 10
146      )
147  port map (
148      clock => clockA,
149      start => startA,
150      wr => wr_temp,
151      the => the_temp,
152      read => readA,
153      endOp => endOpA,
154      rtsT => rqtst,
155      enCount => en_temp,
156      countT => count_temp,
157      data => data_temp
158  );
159
160
161  CountA: contatore
162      generic map (
163          N => 10
164      )
165  port map (
166      clk => clockA,
167      rst => resetA,
168      enable => en_temp,
169      count_out => count_temp
170  );
171
172
173 end Structural;

```

E infine sono stati effettuati i rispettivi collegamenti come mostrati nella sezione “*Schematici*”

```

105  signal data_temp: std_logic_vector(7 downto 0);
106  signal tbe_temp: std_logic;
107  signal wr_temp: std_logic;
108
109  signal rxdt: std_logic;
110  signal rdःt: std_logic;
111  signal rdःt: std_logic;
112  signal peT: std_logic;
113  signal feT: std_logic;
114  signal oeT: std_logic;
115  signal dboutT: std_logic_vector(7 downto 0);
116
117  signal en_temp: std_logic;
118  signal count_temp: integer;
119
120 begin
121
122  UART: UARTcomponent
123      generic map (
124          BAUD_DIVIDE_G => 14, --115200 baud
125          BAUD_RATE_G => 231
126      )
127      port map (
128          TXD => serialData,
129          RXD => rxdt,
130          CLK => clockA,
131          DBIN => data_temp,
132          DBOUT => dboutT,
133          RDA => rdःt,
134          TBE => tbe_temp,
135          RD => rdःt,
136          WR => wr_temp,
137          PE => peT,
138          FE => feT,
139          OE => oeT,
140          RST => resetA
141  );
142
143  T: Trasmettitore
144      generic map (
145          N => 10
146      )
147  port map (
148      clock => clockA,
149      start => startA,
150      wr => wr_temp,
151      the => the_temp,
152      read => readA,
153      endOp => endOpA,
154      rtsT => rqtst,
155      enCount => en_temp,
156      countT => count_temp,
157      data => data_temp
158  );
159
160
161  CountA: contatore
162      generic map (
163          N => 10
164      )
165  port map (
166      clk => clockA,
167      rst => resetA,
168      enable => en_temp,
169      count_out => count_temp
170  );
171
172
173 end Structural;

```

## Ricevitore

L'entità ricevitore presenta 5 segnali di ingresso e 3 di uscita, ovvero, come segnale di input è presente: il clock, il segnale di RDA dell'UART, il dato parallelo in uscita al pin DBOUT dell'UART, il segnale di Request To Send (rts) del sistema A e il valore del conteggio. In uscita invece: il dato e il flag di fine operazione.

Il Ricevitore possiede una memoria interna definita come vettore di vettore di segnali, nella quale vengono memorizzati i dati nella corrispondente locazione puntata dal contatore.

```
34 entity Ricevitore is
35     generic (
36         N: integer := 10
37     );
38     Port (
39         clk: in std_logic;
40         rda: in std_logic;
41         dato_in: in std_logic_vector(7 downto 0);
42         endOpR: out std_logic;
43         rts: in std_logic;
44         enCountR: out std_logic;
45         countR: in integer;
46         y0: out std_logic_vector(7 downto 0)
47     );
48 end Ricevitore;
49
50 architecture Behavioral of Ricevitore is
51
52 type rom_type is array (N-1 downto 0) of std_logic_vector(7 downto 0);
53 signal ROM_2 : rom_type := (
54     "00000000",
55     "00000000",
56     "00000000",
57     "00000000",
58     "00000000",
59     "00000000",
60     "00000000",
61     "00000000",
62     "00000000",
63     "00000000");
64
65
66 type stato is (idle,store,endTrasm);
67 signal stato_corrente : stato := idle;
68
69 begin
70 process(clk)
71 begin
72 begin
73
74 if rising_edge(clk) then
75 case stato_corrente is
76 when idle =>
77     endOpR <= '0';
78     if(rts = '1') then
79         stato_corrente <= store;
80     else
81         stato_corrente <= idle;
82     end if;
83
84 when store =>
85     endOpR <= '0';
86     ROM_2(countR) <= dato_in;
87     stato_corrente <= endTrasm;
88
89 when endTrasm =>
90     y0 <= ROM_2(countR);
91     enCountR <= '1';
92     endOpR <= '1';
93     stato_corrente <= idle;
94
95 end case;
96 end if;
97 end process;
98 end Behavioral;
```

L'automa è stato già descritto nella sezione “*Schematici*”, possiamo notare come a differenza del Trasmettitore qui non viene effettuato il controllo sul valore di conteggio, in quanto il ricevitore nello stato **idle** è in attesa di un segnale di RTS da parte del Trasmettitore nel sistema A, di conseguenza al termine della trasmissione degli N elementi se il Trasmettitore non riceve ulteriori segnali di start non procederà mai più con l'invio di tale segnale.

## Memoria

La memoria è stata realizzata mediante un approccio comportamentale. Sul fronte di salita del clock attende il segnale di memorizzazione da parte del ricevitore, ricevuto tale segnale memorizza il dato. La locazione di memoria in cui memorizzare il dato viene indicata dallo stesso contatore utilizzato per il Ricevitore.

Inoltre, per dare la possibilità all'utente di visualizzare i valori sulla board, ad ogni fronte di salita del clock viene effettuato un controllo su un ulteriore segnale, il quale rappresenta (per una nostra scelta di progettazione) la pressione di un pulsante. Alla pressione di tale pulsante la memoria incrementa il valore di una variabile e mostra in output i bit memorizzati nella locazione di memoria corrispondente al valore di tale variabile. Il codice VHDL della memoria è mostrato in *figura 9.9*.

## System B

Il sistema B è stato realizzato tramite un approccio strutturale. E' stata definita l'entity del sistema, definendone gli opportuni segnali di ingresso e uscita. Poi abbiamo proceduto con l'istanziazione dei componenti e il collegamento tra di essi rispettando lo schema mostrato al paragrafo precedente in *figura 9.7*.

```

34  entity Memoria is
35      generic(
36          N: integer := 10
37      );
38      Port (
39          clk: in std_logic;
40          rst: in std_logic;
41          enSet: in std_logic;
42          head: in integer;
43          TxData: in std_logic_vector(7 downto 0);
44          mem_out: out std_logic_vector(7 downto 0)
45      );
46  end Memoria;
47
48  architecture Behavioral of Memoria is
49
50      signal last_enset: std_logic := '0';
51
52      type rom_type is array (N-1 downto 0) of std_logic_vector(7 downto 0);
53      signal ROM_3 : rom_type := (
54          "00000000",
55          "00000000",
56          "00000000", --
57          "00000000", --
58          "00000000", --
59          "00000000",
60          "00000000", --
61          "00000000", --
62          "00000000",
63          "00000000");
64
65      begin
66          process(clk)
67              variable tail: integer := 0;
68          begin
69              if rising_edge(clk) then
70                  ROM_3(head) <= TxData;
71                  if(enSet = '1' and last_enset = '0') then
72                      last_enset <= '1';
73                      mem_out <= ROM_3(tail);
74                  if(tail = N-1) then
75                      tail := 0;
76                  else tail := tail +1;
77              end if;
78              else last_enset <= '0';
79          end if;
80      --end if;
81  end process;
82
83
84  end Behavioral;

```

Figura 9.9: VHDL Memoria

Di seguito viene riportato il codice del System B.

```

34  entity SystemB is
35      Port(
36          clkM: in std_logic;
37          resetM: in std_logic;
38          dataM: in std_logic;
39          endOpM: out std_logic;
40          readR: in std_logic;
41          rqtS: in std_logic;
42          btn_rom: std_logic;
43          y: out std_logic_vector(7 downto 0)
44      );
45  end SystemB;
46
47
48  architecture Structural of SystemB is
49
50      component UARTcomponent is
51          Generic (
52              --@40MHz
53              BAUD_DIVIDE_G : integer := 26; --115200 baud
54              BAUD_RATE_G : integer := 417
55              );
56          End component;
57
58          BAUD_DIVIDE_G : integer := 14; --115200 baud
59          BAUD_RATE_G : integer := 231
60      );
61
62      Port (
63          TXD : out std_logic := '1'; -- Transmitted serial data output
64          RXD : in std_logic; -- Received serial data input
65          CLK : in std_logic; -- Clock signal
66          DBIN : in std_logic_vector (7 downto 0); -- Input parallel data to be transmitted
67          DBOUT : out std_logic_vector (7 downto 0); -- Received data output
68          RD : out std_logic; -- Read Strobe
69          WR : in std_logic; -- Write Strobe
70          PE : out std_logic; -- Parity error
71          FE : out std_logic; -- Frame error
72          OE : out std_logic; -- Overwrite error
73          RST : in std_logic := '0'); -- Reset signal
74  end component;
75
76  component Ricevitore is
77      generic (
78          N: integer := 10
79      );
79  end component;
80
81  Port (
82      clk: in std_logic;
83      rda: in std_logic;
84      data_in: in std_logic_vector(7 downto 0);
85      endOpR: out std_logic;
86      rta: in std_logic;
87      enCountR: out std_logic;
88      countR: in integer;
89      y0: out std_logic_vector(7 downto 0)
90  );
91
92  component contatore is
93      generic(
94          N: integer := 2
95      );
96  end component;
97
98  Port (
99      clk: in std_logic;
100     rst: in std_logic;
101     enable: in std_logic;
102     count_out: out integer
103  );
104
105 component Memoria is
106     generic(
107         N: integer := 10
108     );
109     Port (
110         clk: in std_logic;
111         rst: in std_logic;
112         enSet: in std_logic;
113         head: in integer;
114         TxData: in std_logic_vector(7 downto 0);
115         mem_out: out std_logic_vector(7 downto 0)
116     );
117 end component;
118
119
120 signal data_temp: std_logic_vector(7 downto 0);
121 signal rda_temp: std_logic := '0';
122 signal y_temp: std_logic_vector(7 downto 0);
123
124 signal dbinR: std_logic_vector(7 downto 0);
125 signal tbeR: std_logic;
126 signal wrR: std_logic;
127 signal peR: std_logic;
128 signal feR: std_logic;
129 signal oeR: std_logic;
130 signal txdR: std_logic;
131
132 signal count_temp: integer;
133 signal enCount_temp: std_logic;
134
135
136 begin
137
138
139 uart: UARTcomponent
140     generic map (
141         BAUD_DIVIDE_G => 14, --115200 baud
142         BAUD_RATE_G => 231
143     )
144     port map (
145         TXD => txdR,
146         RXD => dataB,
147         CLK => clockB,
148         DBIN => dbinR,
149         DBOUT => data_temp,
150         RDA => rda_temp,
151         TBE => tbeR,
152         RD => readB,
153         WR => wrR,
154         PE => peR,
155         FE => feR,
156         OE => oeR,
157         RST => resetB
158     );
159
160 R: Ricevitore
161     generic map (
162         N => 10
163     )
164     port map (
165         clk => clockB,
166         rda => rda_temp,
167         data_in => data_temp,
168         endOpR => endOpB,
169         rts => rqtS,
170         enCountR => enCount_temp,
171         countR => count_temp,
172         y0 => y_temp
173 );
174
175
176 cont: contatore
177     generic map (
178         N => 10
179     )
180     port map (
181         clk => clockB,
182         rst => resetB,
183         enable => enCount_temp,
184         count_out => count_temp
185 );
186
187 Mem: Memoria
188     generic map (
189         N => 10
190     )
191     port map (
192         clk => clockB,
193         rst => resetB,
194         enSet => btn_rom,
195         head => count_temp,
196         TxData => y_temp,
197         mem_out => y
198 );
199
200 end Structural;

```

## SerialSystem

Il top-module è stato realizzato tramite approccio strutturale, esso presenta in ingresso il clock, il reset, due bottoni (btn e btn2) ovvero rispettivamente il pulsante per lo start e il pulsante per scorrere la memoria. In uscita invece presenta un segnale *std\_logic* da 8 bit. Esso è composto dal Systema A e il System B opportunamente collegati. Di seguito il codice VHDL.

```
34 entity SerialSystem is
35     Port (
36         clock_in: in std_logic;
37         reset_in: in std_logic;
38         btn: in std_logic;
39         btn2: in std_logic;
40         y_out: out std_logic_vector(7 downto 0)
41     );
42 end SerialSystem;
43
44 architecture Structural of SerialSystem is
45
46     component SystemA is
47         Port (
48             clockA: in std_logic;
49             resetA: in std_logic;
50             readA: out std_logic;
51             endOpA: in std_logic;
52             startA: in std_logic;
53             rqtst: out std_logic;
54             serialData: out std_logic
55         );
56     end component;
57
58     component SystemB is
59         Port (
60             clockB: in std_logic;
61             resetB: in std_logic;
62             dataB: in std_logic;
63             endOpB: out std_logic;
64             readB: in std_logic;
65             rqtst: in std_logic;
66             btn_rom: std_logic;
67             y: out std_logic_vector(7 downto 0)
68         );
69     end component;
70
71     component ButtonDebouncer is
72         generic (
73             CLK_period: integer := 10; -- period
74             btn_noise_time: integer := 65000000; --assu
75             --assu
76         );
77         Port (
78             RST : in STD_LOGIC;
79             CLK : in STD_LOGIC;
80             BTN : in STD_LOGIC;
81             CLEARED_BTN : out STD_LOGIC);
82     end component;
83
84     begin
85         reset_n <= not reset_in;
86
87         A: SystemA
88             port map (
89                 clockA => clock_in,
90                 resetA => reset_n,
91                 readA => read_temp,
92                 endOpA => endOp_temp,
93                 startA => btn_temp,
94                 rqtst => rts_temp,
95                 serialData => data_temp
96     );
97
98         B: SystemB
99             port map (
100                clockB => clock_in,
101                resetB => reset_n,
102                dataB => data_temp,
103                endOpB => endOp_temp,
104                readB => read_temp,
105                rqtst => rts_temp,
106                btn_rom => btn_temp2,
107                y => y_out
108            );
109
110            b1: ButtonDebouncer
111                generic map (
112                    CLK_period => 10,
113                    btn_noise_time => 65000000
114                )
115                port map (
116                    RST => reset_n,
117                    CLK => clock_in,
118                    BTN => btn,
119                    CLEARED_BTN => btn_temp
120            );
121
122            b2: ButtonDebouncer
123                generic map (
124                    CLK_period => 10,
125                    btn_noise_time => 65000000
126                )
127                port map (
128                    RST => reset_n,
129                    CLK => clock_in,
130                    BTN => btn2,
131                    CLEARED_BTN => btn_temp2
132            );
133
134
135
136
137
138
139
140
141
142     end Structural;
```

# Capitolo 10

## 10. Switch e Omega Network

### 10.1 Traccia 1

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).

#### 10.1.1 Soluzione

In questo caso, si è deciso di suddividere la macchina in unità operativa e unità di controllo, l'unità operativa realizza la rete di switch, con il modello Omega Network, mentre l'unità di controllo realizza la rete a priorità fissa mediante un componente arbitro. Nel nostro progetto abbiamo utilizzato uno schema a priorità fissa, evita che più nodi possano comunicare nello stesso momento, facendo trasmettere solo il nodo a priorità maggiore che decide di trasmettere.

##### 10.1.1.1 Schematici

L'elemento base con cui è realizzata l'Omega Network è lo switch (figura 10.1.1), esso è stato realizzato tramite approccio strutturale componendo un multiplexer 2:1 ed un demultiplexer 1:2, è azionato utilizzando 2 bit (src, dst) il primo seleziona la sorgente mentre il secondo la destinazione, ad esempio nel caso  $\text{src}=0$  e  $\text{dst}=1$ , lo switch inoltra i bit presenti sulla linea A0 di ingresso sulla linea B1 in uscita, l'Omega Network è realizzata componendo più switch secondo la tecnica del perfect shuffling, questa tecnica 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, nel senso di sovrapporre in modo alternato le carte dei due mazzi, dopo  $\log_2(n)$  volte si ritorna all'ordine del mazzo di carte iniziale, questa tecnica ci permette di ottimizzare i percorsi da seguire e quindi come collegare i diversi stadi della rete. Nel nostro progetto abbiamo utilizzato uno schema a priorità fissa, evita che più nodi possano comunicare nello stesso momento, facendo trasmettere solo il nodo a priorità maggiore che decide di trasmettere.

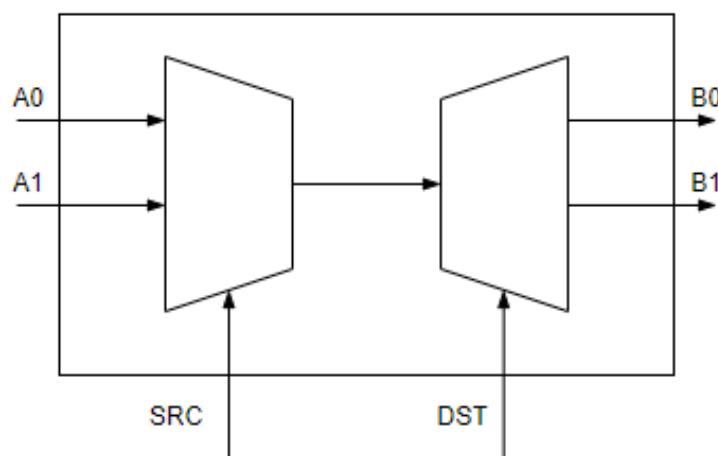


Figura 10.1.1: Switch

L'**unità operativa** (figura 10.1.2) è realizzata con approccio strutturale, seguendo la topologia del perfect shuffling, infatti per collegare 4 nodi sono necessari 2 stadi con 2 switch ciascuno. Le stringhe SRC e DST sono lunghe 2 bit, dato che ci sono 4 possibili indirizzi, dato che la comunicazione avviene da sinistra verso destra, i bit di sorgente sono dati dal bit meno significativo SRC(1) al più significativo SRC(0), i bit di destinazione invece sono dati al contrario. Nell'unità operativa viaggiano solo i bit di dato, infatti i segnali  $X_i$  ed  $Y_i$  sono di tipo std\_logic\_vector(1 downto 0)

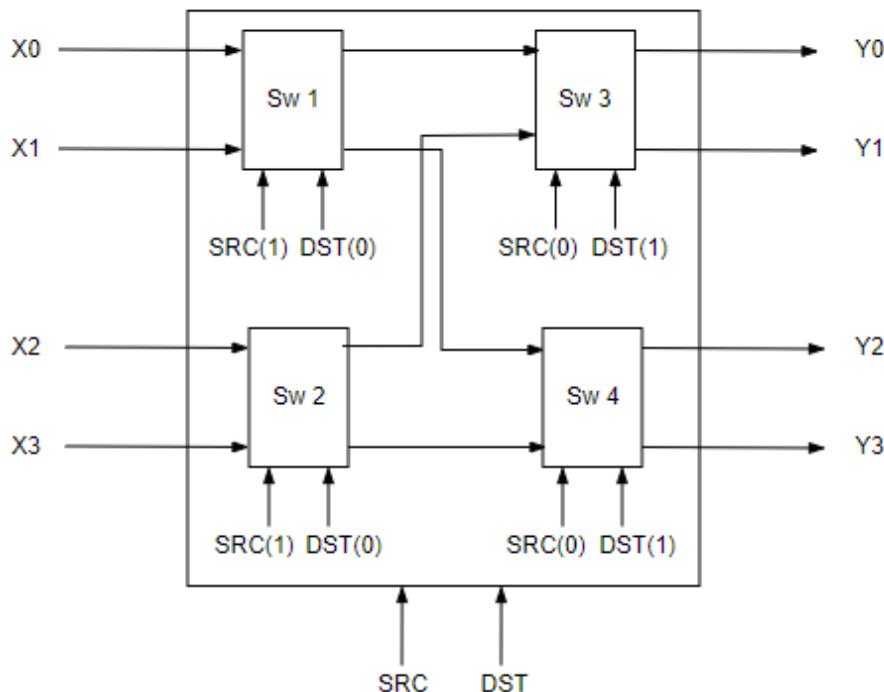


Figura 10.1.2: Unità operativa realizzata con perfect shuffling

L'**unità di controllo** (figura 10.1.3) invece, realizza la rete a priorità, decidendo tramite un componente arbitro quale dei nodi può trasmettere, in base alle priorità fisse. L'unità di controllo riceve in ingresso 6 bit da ciascun nodo che vuole trasmettere, i primi 2 bit sono per il SRC, altri 2 per DST e gli ultimi 2 sono il dato vero e proprio il PAYLOAD. Ogni nodo deve comunicare all'unità operativa la volontà di trasmettere, tramite il segnale di enable, utilizzati successivamente dall'arbitro per decidere chi far trasmettere. E' presente una coppia di mux/demux 4:1, dove in ingresso abbiamo i 4 PAYLOAD da 2 bit dei nodi, mentre le 4 uscite del demultiplexer sono collegate direttamente all'unità operativa, l'arbitro in base ai segnali di enable, decide quale nodo far trasmettere e di conseguenza produce 2 bit di selezione che vengono utilizzati sia per il multiplexer sia per il demultiplexer, in modo da selezionare il PAYLOAD del nodo che si vuole far trasmettere ed inoltrarlo sulla linea giusta in ingresso all'unità operativa, infine l'unità di controllo in base al nodo selezionato dall'arbitro, produce le informazioni su sorgente e destinatario prelevando i bit dalla stringa fornita dal nodo, inoltrate poi all'unità operativa tramite i segnali di SRC e DST. L'Omega Network si ottiene componendo l'unità di controllo e l'unità operativa, come illustrato in figura 10.1.4

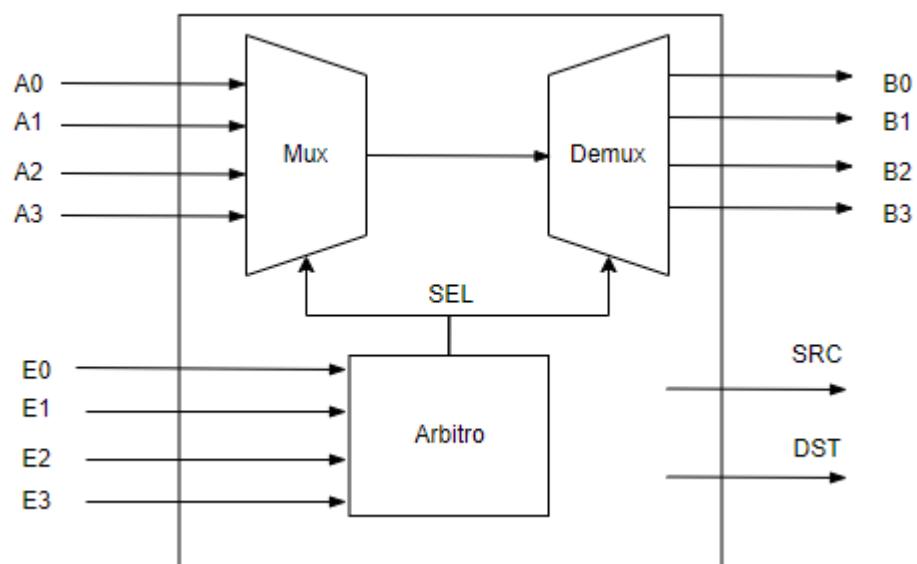


Figura 10.1.3: Unità di controllo

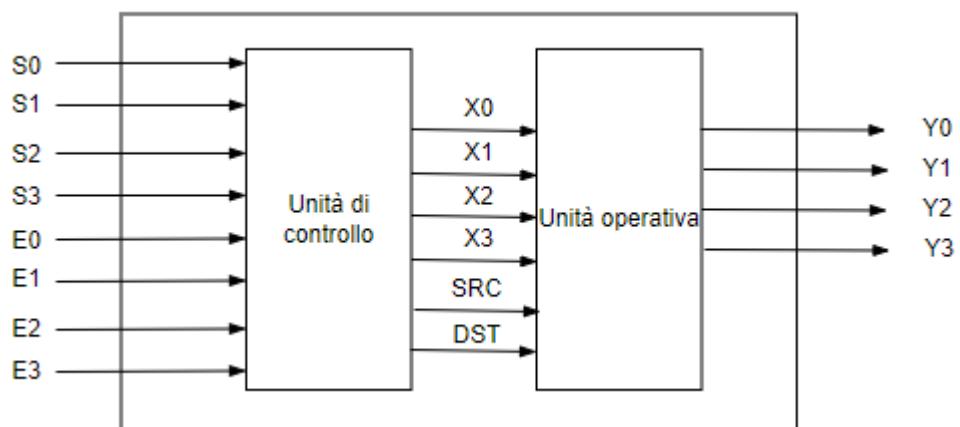


Figura 10.1.4: Omega Network

## 10.1.2 Codice

### Switch

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity switch is
```

```

    port(X0, X1: in std_logic_vector(1 downto 0);
          ss, sd: in std_logic;
          Y0, Y1: out std_logic_vector(1 downto 0));
end switch;

architecture structural of switch is
  component mux_21 is
    port(X0, X1: in std_logic_vector(1 downto 0);
          s: in std_logic;
          Y: out std_logic_vector(1 downto 0));
  end component;
  component dem_12 is
    port(X: in std_logic_vector(1 downto 0);
          s: in std_logic;
          Y0, Y1: out std_logic_vector(1 downto 0));
  end component;
  signal temp_y: std_logic_vector(1 downto 0);
begin

  mux: mux_21
    port map(X0, X1, ss, temp_y);

  dem: dem_12
    port map(temp_y, sd, Y0, Y1);

end structural;

```

Questa macchina è estremamente semplice, è realizzata collegando un mux 2:1 ed un demux 1:2.

## Unità Operativa

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OP is
  port (X0,X1,X2,X3: in std_logic_vector(1 downto 0);
        src, dst: in std_logic_vector(1 downto 0);
        Y0, Y1, Y2, Y3: out std_logic_vector(1 downto 0));
end OP;

architecture structural of OP is
component switch is
port(X0 , X1: in std_logic_vector (1 downto 0);
      ss, sd: in std_logic;
      Y0, Y1: out std_logic_vector (1 downto 0));
end component;
signal t10 , t11 , t20 , t21: std_logic_vector (1 downto 0);
begin
  s1: switch
    port map(X0 ,X1 ,src(0),dst(1),t10 ,t11);
  s2: switch

```

```

port map(X2 ,X3 ,src(0),dst(1),t20 ,t21);
s3: switch
port map(t10 ,t20 ,src(1),dst(0),Y0,Y1);
s4: switch
port map(t11 ,t21 ,src(1),dst(0),Y2,Y3);

end structural;

```

Questa macchina è realizzata componendo 4 switch, come mostrato in figura 10.1.2

## Unità di Controllo

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity CU is
    Port (
        A0, A1, A2, A3: in std_logic_vector(5 downto 0);
        en0, en1, en2, en3: in std_logic;
        B0,B1,B2,B3: out std_logic_vector(1 downto 0);
        s,d: out std_logic_vector(1 downto 0) );
end CU;

architecture structural of CU is
    component mux_41 is
        port(X0, X1, X2, X3: in std_logic_vector(1 downto 0);
              s: in std_logic_vector(1 downto 0);
              Y: out std_logic_vector(1 downto 0));
    end component;
    component dem_14 is
        port(X: in std_logic_vector(1 downto 0);
              s: in std_logic_vector(1 downto 0);
              Y0, Y1, Y2, Y3: out std_logic_vector(1 downto 0));
    end component;
    component priority is
        Port (en0, en1, en2, en3: in std_logic;
              y: out std_logic_vector(1 downto 0) );
    end component;

    signal t_sel: std_logic_vector(1 downto 0);
    signal t_y: std_logic_vector(1 downto 0);
begin
    mux: mux_41  --A0(1 downto 0) DATO input
        port map(A0(1 downto 0), A1(1 downto 0), A2(1 downto 0), A3(1
downto 0),t_sel, t_y);

    demux: dem_14 --B DATO output
        port map(t_y, t_sel, B0,B1,B2,B3);

    arb: priority
        port map(en0, en1, en2, en3, t_sel);

--bit selezione Source della rete Omega

```

```

s <= A0(5 downto 4) when t_sel = "00" else
A1(5 downto 4) when t_sel = "01" else
A2(5 downto 4) when t_sel = "10" else
A3(5 downto 4) when t_sel = "11" else
"--";
--bit selezione Dest della rete Omega
d <= A0(3 downto 2) when t_sel = "00" else
A1(3 downto 2) when t_sel = "01" else
A2(3 downto 2) when t_sel = "10" else
A3(3 downto 2) when t_sel = "11" else
"--";

```

**end structural;**

L'unità di controllo è composta da un arbitro, una coppia mux 4:1 e demux 1:4. L'arbitro realizza una rete a priorità fissa, ed il suo output viene usato per pilotare il mux\demux, l'arbitro in base ai segnali di enable decide quale nodo far trasmettere, ad esempio nel caso en0=1, en1=1, en2=0, en3=1, l'arbitro decide di far trasmettere il nodo 0 poiché è il nodo con priorità più alta tra quelli che hanno richiesto di trasmettere alzando il segnale di abilitazione, il componente allora immetterà in uscita la stringa “00”, essa andrà in ingresso sia al mux 4:1 sia al demux 1:4, nel nostro esempio sarà selezionata la linea A0 dal mux e la linea B0 dal demux, in ingresso alla coppia mux\demux è presente solo il PAYLOAD(2 bit) per ogni nodo. Infine, l'unità di controllo produce in uscita i due segnali s e d, utilizzati successivamente dall'unità operativa per instradare il messaggio, ciò è stato realizzato tramite un approccio dataflow, dalla stringa del nodo selezionato dall'arbitro vengono prelevati 2 bit per il sorgente e 2 per il destinatario.

## Arbitro

L'arbitro è realizzato semplicemente con un approccio dataflow, sfruttando il costrutto when\else

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity priority is
  Port (en0, en1, en2, en3: in std_logic;
        y: out std_logic_vector(1 downto 0) );
end priority;
architecture dataflow of priority is
begin
  y <= "00" when en0 = '1' else
    "01" when en1='1' else
    "10" when en2 = '1' else
    "11" when en3 = '1' else
    "--";
end dataflow;

```

### 10.1.3 Simulazione

Tramite il testbench riportato di seguito (figura 10.1.5) , abbiamo simulato la trasmissione di un messaggio dal nodo X0 al nodo Y2, il segnale di abilitazione E vale “1001”, quindi sia il nodo X3 sia il nodo X0 vogliono trasmettere, l'arbitro farà trasmettere solo il nodo X0 in quanto a priorità maggiore, la stringa S0 contiene i bit di informazione sulla trasmissione che il nodo X0 vuole

effettuare, i primi 2 bit sono i bit che codificano il nodo sorgente, i successivi 2 riguardano il destinatario, mentre gli ultimi 2 riguardano il messaggio vero e proprio, nel nostro caso S0 vale "001011", quindi i bit sorgente sono 00, i bit destinazione 10, mentre i bit del messaggio 11

```
wait for 10 ns;
-----s0,s1,d0,d1
S0 <= "001011"; --11 al nodo Y2
S1 <= "011010"; --10 al nodo Y2
S2 <= "101001"; --01 al nodo Y2
S3 <= "111101"; --01 al nodo Y3
E <= "1001"; --da SX-DX il X3,X2,X1,X0
```

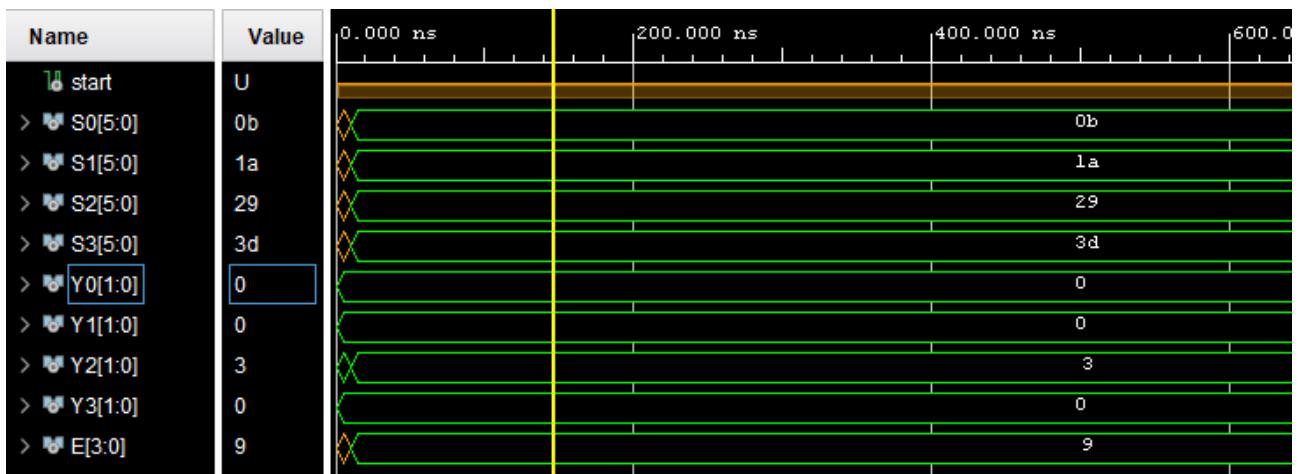


Figura 10.1.5: Simulazione Omega Network

# 11. Moltiplicatore di Robertson

## 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;

## 11.2 Cenni teorici

Il moltiplicatore di Robertson è un moltiplicatore sequenziale, per capirne il funzionamento, analizziamo come si effettua la moltiplicazione manualmente, passo per passo, cominciando dall'algoritmo manuale di moltiplicazione binaria. Se dobbiamo moltiplicare 1010 per 1101, prendiamo il moltiplicando, lo moltiplichiamo per la cifra i-esima del moltiplicatore e poi dobbiamo sommare delle righe che non hanno sempre lo stesso peso ma sono ogni volta shiftate di una posizione a sinistra. Dunque, prima generiamo le righe e poi ne facciamo la somma, se entrambi gli operandi hanno dimensione N allora il prodotto ha dimensione  $2^N$ . Ad ogni passo prendiamo la cifra i-esima, la moltiplichiamo per il moltiplicando, poi ogni volta moltiplichiamo per un fattore  $2^i$ , ovvero facciamo uno shift a sinistra di i posizioni.

Vediamo cosa possiamo fare per modificare la procedura manuale in qualcosa di iterativo. Non conserviamo tutte le righe e poi facciamo la somma, come visto nelle architetture parallele, ma a ogni passo cominciamo a calcolare la somma intermedia: consideriamo come somma parziale iniziale tutti 0 e poi sommiamo il prodotto  $2^{j^*}x_j$ , il prodotto ogni volta è shiftato di una posizione a sinistra. La somma non la facciamo alla fine per n righe ma iterativamente per ogni riga. Se dobbiamo trasformarlo in un algoritmo, al passo i-esimo il numero di shift dipende da i, non è fisso. Possiamo fare ancora meglio, invertiamo l'ordine della somma e degli shift, facciamo prima la somma parziale e poi lo shift a destra di una posizione:

$$P_i = P_{i-1} + x_i * Y$$

$$P_{i+1} = P_i * 2^{-1}$$

Vediamo ora un esempio di moltiplicazione:

<b>Y</b>	<b>1010</b>
<b>X</b>	<b>1101</b>
0000 0000	P0
<b>1010</b>	
0000 1010	ADD;shift
000 0101 0	P1
<b>0000</b>	
000 0101 0	ADD;shift
00 0010 10	P2
<b>1010</b>	
00 1100 10	ADD;shift
0 0110 010	P3
<b>1010</b>	
1 0000 010	ADD;shift
<b>1000 0010</b>	

Fig. 11.1 Esempio di moltiplicazione

Cerchiamo di immaginare il datapath necessario a una macchina che opera in tal modo. Mettiamo il moltiplicando in un registro da N bit ed il moltiplicatore in un registro che ha già lunghezza doppia. Se pensiamo ai bit in blu in figura 11.1, è un registro in tutto grande  $2^*N$  bit che si riempie piano piano, con A(primi 4 bit blu) e Q(secondi 4 bit blu). All'inizio esso potrebbe contenere la somma parziale che sono 8 zeri, poi ad ogni iterazione l'ultimo bit, il meno significativo, viene buttato, dunque man mano delle 8 cifre ne servono meno. Anzichè usare un registro a parte per mantenere gli altri dati che ci servono, sfruttiamo il registro in cui le cifre a ogni iterazione si liberano. La porzione meno significativa del registro, Q, all'inizio è vuota e viene riempita man mano che avvengono gli shift, alla fine avremo sugli 8 bit che prima contenevano tutti 0 (prodotto parziale iniziale, P0) un pezzo di A e di Q che infine fanno il risultato.

Per il calcolo del prodotto di due interi relativi non è possibile applicare l'algoritmo derivato dalla procedura manuale, che fornisce un risultato errato. Se i numeri sono codificati in modulo e segno l'unica possibilità è quella di calcolare separatamente il prodotto dei moduli e dei segni. Se i numeri sono codificati in complementi a due esiste una soluzione alternativa che si basa su alcune proprietà della rappresentazione in complementi.

In complementi a due, se dobbiamo fare  $-X$  dobbiamo complementare tutti i bit e aggiungere 1. Il singolo bit del numero, in complementi, è 1 meno la cifra  $i$ -esima, ovvero il complemento:

$$-X = x_{n-1}x_{n-2} \dots x_1x_0 + 00000001$$

Se  $X$  è positivo, dunque la cifra di peso più significativo vale 0, possiamo scrivere  $x$  tramite la seguente sommatoria:

$$X = \sum_{i=0}^{n-2} 2^i x_i$$

Se invece è negativo non possiamo fare così, il primo bit del segno sappiamo sarà 1 ma non sappiamo il suo peso, lo dobbiamo ricavare. Abbiamo determinato  $-X$ , dunque prima lo consideriamo con lo 0 davanti e poi sommiamo 10...0, perché sappiamo che quella cifra vale 1 (la più significativa, che determina il segno)

$$\begin{aligned} -X &= 111\dots11 - (0x_{n-2}\dots x_1x_0 + 100\dots00) + 000\dots01 = \\ &= (111\dots11 - 100\dots00 + 000\dots01) - x_{n-2}\dots x_1x_0 = \\ &= 100\dots00 - x_{n-2}\dots x_1x_0 = \\ &= 2^{n-1} - x_{n-2}\dots x_1x_0 \end{aligned}$$

Per avere  $X$  in complementi moltiplichiamo per il segno meno, ottenendo:

$$X = -2^{n-1} + x_{n-1} \dots x_1x_0 = -2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

In complementi, un numero positivo si scrive solo con la sommatoria, un numero negativo invece la cifra più significativa ha peso -1. Abbiamo dovuto fare ciò perché l'algoritmo sequenziale guarda una cifra per volta per cui dobbiamo tener conto del peso negativo quando arriviamo alla cifra più significativa. o. In particolare, abbiamo 4 possibili casi da dover gestire:

- Se  $X$  e  $Y$  sono entrambi positivi e facciamo una moltiplicazione le somme sono sempre positive, opportunamente shiftate, non dobbiamo correggere niente.
- Se  $X$  è positivo ma  $Y$  (moltiplicando) è negativo, lo sappiamo subito, ogni volta che prendiamo la  $Y$  e la moltiplichiamo per una cifra non nulla il prodotto parziale è negativo, scriviamo 1 in un flip flop (F visto nell'unità operativa) e questo rimane il valore del segno, fino alla fine.
- Se  $Y$  è positivo e  $X$  è negativo fino all'ultima cifra non ce ne accorgiamo, facciamo le somme, all'ultima cifra ci accorgiamo che ha un peso negativo, invece di fare la somma facciamo la sottrazione (passo di correzione).

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

Tutto l'algoritmo prevede di fare moltiplicazione, somma e shift ad ogni passo, solo all'ultimo dobbiamo controllare se fare o meno la correzione.

## 11.3 Soluzione

Abbiamo scomposto l'architettura del moltiplicatore di Robertson in parte operativa e parte di controllo. In particolare, la parte operativa è stata realizzata con un approccio strutturale, mettendo insieme i componenti che realizzano la moltiplicazione, la parte di controllo invece è stata realizzata, in logica cablata, come un automa a stati finiti, che in ogni stato compie determinati passi, abilitando i pezzi della parte operativa per ottenere il risultato finale.

### 11.3.1 Schematici

Lo schema seguito per il progetto del moltiplicatore di Robertson `e quello presente in figura 11.2, in particolare, abbiamo l'unità di controllo e poi tutti gli altri blocchi che formano l'unità operativa. A destra c'è Y che mettiamo nel registro M, ad ogni iterazione dobbiamo fare il prodotto di Y con il bit i-esimo di X, che può essere un 1 o uno 0, piuttosto che fare la and bit a bit usiamo un multiplexer.

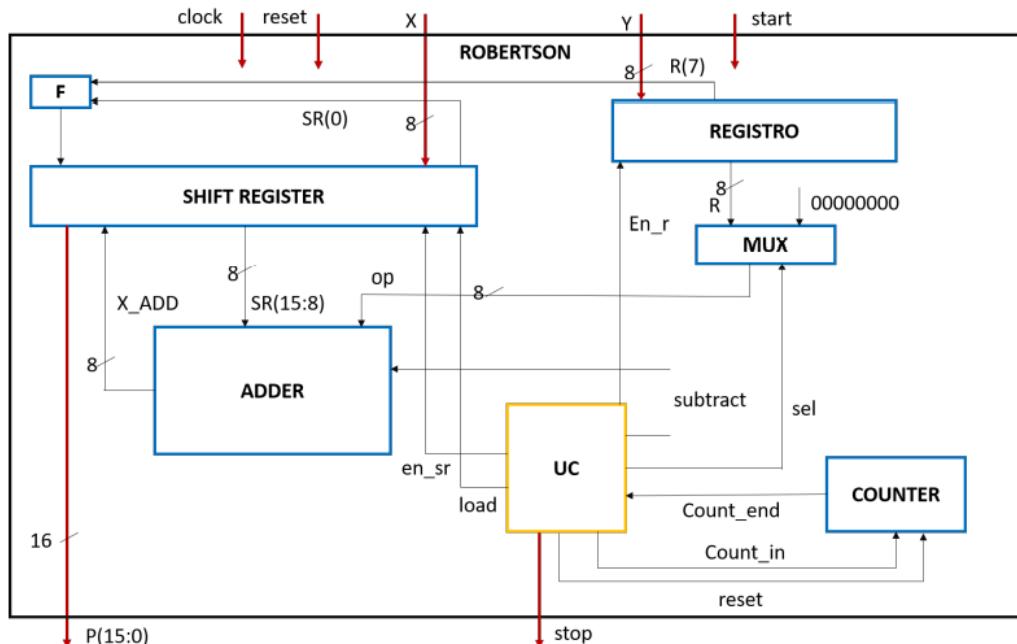


Fig. 11.2 Schema moltiplicatore di Robertson

Il moltiplicatore, invece, è memorizzato in uno shift-register che inizialmente contiene una stringa di 8 zeri e X, poi ad ogni passo eseguo uno shift così da sfruttare le posizioni liberate dal moltiplicando per memorizzare le cifre della somma parziale. In testa allo shift register c'è l'uscita del flip flop. L'uscita dell'addizionatore va direttamente nello shift-register, in ingresso all'addizionatore va o M o 0, in un operando, e la stessa metà dello shift-register, poi l'adder ha il carry in che `e collegato al segnale subtract della UC. Con la XOR, l'adder e il segnale di sub siamo in grado di realizzare un sommatore/sottrattore (figura 11.3). Il contatore va in ingresso alla UC, perchè disciplina il passo di moltiplicazione a cui siamo, all'ultimo va fatta la correzione: `e la UC a dare il segnale subtract. La UC dà anche delle uscite verso il counter perchè gli deve dare il conteggio.

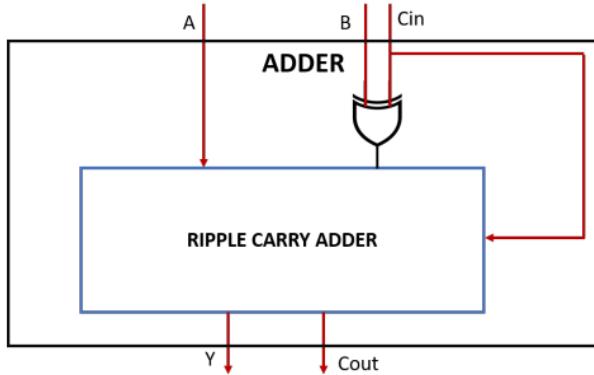


Fig. 11.3 Schema sommatore/sottrattore

### 11.3.2 Codice

Analizziamo il codice dei singoli blocchi che caratterizzano l'**unità operativa**.

- Shift-register

```
entity ShiftRegister is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    en: in std_logic;
    data: in std_logic_vector(15 downto 0);
    y: out std_logic_vector(15 downto 0);
    v_add: in std_logic_vector(7 downto 0);
    shift: in std_logic;
    f_shift: in std_logic;
    load: in std_logic;
    load_sp: in std_logic;
    F: in std_logic
  );
end ShiftRegister;

architecture Behavioral of ShiftRegister is

  signal value_temp: std_logic_vector(15 downto 0);

begin
  process(clk, rst)
  begin
```

Il blocco shift-register è stato descritto a livello comportamentale con un process caratterizzato dalla presenza dei segnali di clock e reset nella sensitivity lists. Esso è

```
if(en = '1') then
  if(rst = '1') then
    value_temp <= "0000000000000000";
  elsif rising_edge(clk) then
    if(load = '1') then
      value_temp <= data;
    elsif(load_sp = '1') then
      value_temp(15 downto 8) <= v_add;
    elsif(shift = '1') then
      value_temp(14 downto 0) <= value_temp(15 downto 1);
      value_temp(15) <= F;
    elsif(f_shift = '1') then
      value_temp(14 downto 0) <= value_temp(15 downto 1);
    end if;
  end if;
end if;

y <= value_temp;
end process;
end Behavioral;
```

caratterizzato da un reset sincrono, per cui se è alto, l'uscita presenta tutti 0. Quando non c'è il reset, sul fronte di salita del clock, se load è alto, viene caricato l'ingresso. Se invece è alto il segnale load\_sp, viene caricato nei primi 8 bit il risultato della somma parziale. Se il segnale di shift è alto si effettua il right shift, ponendo in testa il valore F, ricevuto dal flip flop D; infine, se è alto il segnale di f\_shift, effettua lo shift finale, dunque shifta a destra mantenendo invariato il bit più significativo.

- **Adder**

```

entity Adder is
  port(
    M: in std_logic_vector(7 downto 0);
    X: in std_logic_vector(7 downto 0);
    Y: out std_logic_vector(7 downto 0);
    carry_in: in std_logic;
    carry_out: out std_logic
  );
end Adder;

architecture Structural of Adder is

component RippleCarryAdder is
  Port (
    A: std_logic_vector(7 downto 0);
    B: std_logic_vector(7 downto 0);
    c_in: in std_logic;
    c_out: out std_logic;
    s: out std_logic_vector(7 downto 0)
  );
end component;

signal value1_temp: std_logic_vector(7 downto 0);

begin

  fortemp: for I in 0 to 7 generate
    value1_temp(I) <= M(I) xor carry_in;
  end generate fortemp;

  Rip: RippleCarryAdder
    port map (
      A => value1_temp,
      B => X,
      c_in => carry_in,
      c_out => carry_out,
      s => Y
    );

end Structural;

```

Il blocco adder è stato descritto a livello strutturale, tramite la composizione di due componenti: il ripple carry adder, ottenuto per composizione di full adder, e un blocco che effettua il complemento delle cifre, facendo la xor bit a bit tra le cifre del secondo operando e il riporto entrante.

- **Contatore**

```

entity contatore is
    generic(
        N: integer := 8
    );
    Port (
        clk: in std_logic;
        rst: in std_logic;
        enable: in std_logic;
        count_end: out std_logic;
        count_out: out integer
    );
end contatore;

architecture Behavioral of contatore is

begin
process(clk,rst,enable)
variable count: integer  := 0;
begin
    begin
        if(rst = '1') then
            count := 0;
        end if;
        if rising_edge(clk) then
            if(enable = '1') then
                if(count = N-2) then
                    count_end <= '1';
                    count := 0;
                else
                    count_end <= '0';
                    count := count +1;
                end if;
            end if;
            count_out <= count;
        end process;
    end Behavioral;

```

Tale blocco è un contatore modulo 8, descritto a livello comportamentale, necessario a mantenere il conteggio dei passi eseguiti dalla UC. Esso presenta un reset sincrono, per cui se è alto, pone la variabile di conteggio a 0. Una volta arrivati al settimo passo (count = N-2), il contatore si resetta e notifica l'UC, alzando il segnale count\_end, in modo tale che essa possa a effettuare il passo di correzione e lo shift finale.

- **Registro**

```

entity Registro is
    Port (
        Input: in std_logic_vector(7 downto 0);
        Output: out std_logic_vector(7 downto 0);
        enable: in std_logic;
        rst: in std_logic;
        clk: in std_logic
    );
end Registro;

architecture Behavioral of Registro is

begin
process(clk, rst)
begin

```

```

        if(enable = '1') then
            if(rst = '1') then
                Output <= "00000000";
            elsif rising_edge(clk) then
                Output <= Input;
            end if;
        end if;

    end process;
end Behavioral;

```

Tale blocco è un registro parallelo-parallelo, descritto a livello comportamentale, dove viene memorizzato il valore del moltiplicando. Esso è caratterizzato da un reset sincrono, per cui se è alto, l'uscita presenta tutti 0. Quando non c'è il reset, sul fronte di salita del clock, pone l'uscita pari all'ingresso.

- **Multiplexer 2:1**

Per brevità non andiamo ad inserire il codice del mux 2:1; esso serve per effettuare la moltiplicazione delle cifre del moltiplicatore per il moltiplicando, se la cifra è 0, il risultato sarà la stringa di 0, se la cifra è 1, il risultato è il moltiplicando stesso. Dunque, usiamo un multiplexer e poniamo come ingressi il moltiplicando e la stringa di 0, poi la cifra q0 del moltiplicatore come ingresso di selezione.

- **Flip-Flop**

```

entity FlipFlop is
    Port (
        R: in std_logic;
        Clk: in std_logic;
        D: in std_logic;
        E: in std_logic;
        Q: out std_logic
    );
end FlipFlop;

architecture Behavioral of FlipFlop is

begin
process(clk, R)
begin

    if(E = '1') then
        if(R = '1') then
            Q <= '0';
        elsif rising_edge(Clk) then
            Q <= D;
        end if;
    end if;
end process;

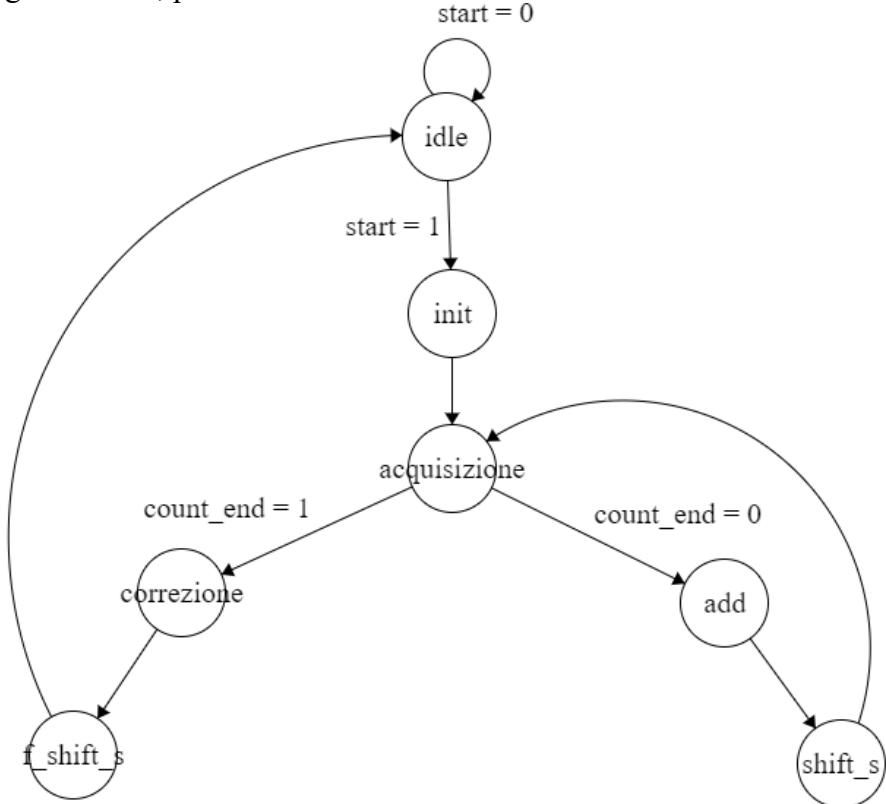
end Behavioral;

```

Il flip-flop è descritto a livello comportamentale. . Esso è caratterizzato da un reset sincrono, per cui se è alto, l'uscita presenta tutti 0. Quando non c'è il reset, sul fronte di salita del clock, pone l'uscita pari all'ingresso. È necessario per mantenere il segno, ovvero per capire se il risultato dev'essere positivo o negativo. Il segno viene calcolato come ( $Y(7)$  and  $Q(0)$ )or F, è

alto se il prodotto tra la cifra del moltiplicatore e l'ultima cifra del moltiplicando è alto oppure se era già alto, dunque una volta capito che il segno è 1, rimane tale.

Analizziamo ora il funzionamento ed il codice relativo all'**unità di controllo**. Dato che è stata realizzata in logica cablata, partiamo dall'automa:



Per comprendere al meglio l'automa effettuiamo l'analisi stato per stato.

- **Idle**: In questo stato, l'unità di controllo resetta tutti i componenti dell'UO, alzando reset e abilitazioni in uscita in quanto, per come abbiamo progettato i componenti, questi si resettano se l'abilitazione è alta. Una volta che il segnale di start si alza si passa allo stato init.
- **Init**: Giunti in questo stato, andremo a predisporre l'acquisizione dei dati, che avverrà al colpo di clock successivo. Quindi, l'unità di controllo, abilita il flip-flop, il registro e lo shift register con il segnale di load alto. La selezione del multiplexer è posta pari a q0, che è la cifra considerata del moltiplicatore.
- **Acquisizione**: In questo stato, vengono acquisiti gli operandi e predisposta l'addizione, ovvero la somma parziale; infatti, abilitiamo lo shift register alzando il segnale load\_sp, in modo tale da caricare nei primi 8 bit la somma parziale. Inoltre, alziamo il segnale di abilitazione del contatore. Se il conteggio è pari a 7 e quindi count\_end è alto, si alza il segnale di sub (per effettuare la sottrazione al passo successivo) e si passa allo stato correzione. In caso contrario, si passa allo stato add.
- **Add**: Nello stato add, si esegue la somma e si predispone lo shift, dunque si abilita lo shift register con segnale di shift alto.
- **Shift\_s**: In questo stato, si effettua lo shift e si passa nuovamente allo stato acquisizione per effettuare poi il passo successivo della moltiplicazione.
- **Correzione**: Questo stato è del tutto analogo a quello di add, con la differenza che il segnale di sub è alto e quindi viene effettuata la sottrazione anziché l'addizione.
- **F\_shift\_s**: In questo stato, si effettua lo shift finale e si ritorna nello stato idle in cui la macchina aspetta nuovamente il segnale di start per la moltiplicazione successiva.

Vediamo il codice:

```

entity UnitaDiControllo is
  Port (
    clock: in std_logic;
    reset: in std_logic;
    q0: in std_logic;
    start: in std_logic;
    count_end: in std_logic;
    en_C: out std_logic;
    en_SR: out std_logic;
    en_R: out std_logic;
    en_FP: out std_logic;
    load_sp: out std_logic;
    load: out std_logic;
    sel: out std_logic;
    reset_out: out std_logic;
    sub: out std_logic;
    stop: out std_logic;
    shift: out std_logic;
    f_shift: out std_logic;
    stop_cu: out std_logic
  );
end UnitaDiControllo;

architecture Behavioral of UnitaDiControllo is
type state is (idle, init, acquisizione, add, shift_s, correzione, f_shift_s);
signal current_state: state := idle;
begin
process(clock, reset)
begin
  if rising_edge(clock) then
    if(reset = '1') then
      current_state <= init;
    end if;
    case current_state is
      when idle =>
        reset_out <= '1';
        en_SR <= '1';
        en_R <= '1';
        en_FP <= '1';
        en_C <= '0';
        sub <= '0';
        load <= '0';
        load_sp <= '0';
        stop <= '0';
        stop_cu <= '0';
        shift <= '0';
        f_shift <= '0';
        if(start = '1') then
          current_state <= init;
        else
          current_state <= idle;
        end if;
      when acquisizione =>
        en_SR <= '0';
        en_R <= '0';
        en_FP <= '0';
        en_C <= '1';
        sub <= '1';
        load <= '1';
        load_sp <= '1';
        stop <= '1';
        stop_cu <= '1';
        shift <= '1';
        f_shift <= '1';
        if(count_end = '1') then
          current_state <= add;
        else
          current_state <= acquisizione;
        end if;
      when add =>
        en_SR <= '0';
        en_R <= '0';
        en_FP <= '0';
        en_C <= '0';
        sub <= '0';
        load <= '0';
        load_sp <= '0';
        stop <= '0';
        stop_cu <= '0';
        shift <= '0';
        f_shift <= '0';
        if(count_end = '1') then
          current_state <= correzione;
        else
          current_state <= add;
        end if;
      when shift_s =>
        en_SR <= '0';
        en_R <= '0';
        en_FP <= '0';
        en_C <= '0';
        sub <= '0';
        load <= '0';
        load_sp <= '0';
        stop <= '0';
        stop_cu <= '0';
        shift <= '1';
        f_shift <= '1';
        if(count_end = '1') then
          current_state <= f_shift_s;
        else
          current_state <= shift_s;
        end if;
      when correzione =>
        en_SR <= '0';
        en_R <= '0';
        en_FP <= '0';
        en_C <= '0';
        sub <= '0';
        load <= '0';
        load_sp <= '0';
        stop <= '0';
        stop_cu <= '0';
        shift <= '0';
        f_shift <= '0';
        if(count_end = '1') then
          current_state <= f_shift_s;
        else
          current_state <= correzione;
        end if;
      when f_shift_s =>
        en_SR <= '0';
        en_R <= '0';
        en_FP <= '0';
        en_C <= '0';
        sub <= '0';
        load <= '0';
        load_sp <= '0';
        stop <= '0';
        stop_cu <= '0';
        shift <= '0';
        f_shift <= '0';
        current_state <= idle;
    end case;
  end if;
end process;
end;

```

```

when init =>
    reset_out <= '0';
    en_C <= '0';
    load_sp <= '0';
    en_R <= '1';
    en_FP <= '1';
    en_SR <= '1';
    load <= '1';
    sub <= '0';
    stop <= '0';
    sel <= q0;
    shift <= '0';
    f_shift <= '0';
    stop_cu <= '0';
    current_state <= acquisizione;

when acquisizione =>
    reset_out <= '0';
    en_SR <= '1';
    en_R <= '1';
    en_FP <= '1';
    load <= '0';
    load_sp <= '1';
    en_C <= '1';
    sub <= '0';
    sel <= q0;
    stop <= '0';
    stop_cu <= '0';
    shift <= '0';
    f_shift <= '0';
    if(count_end = '1') then
        sub <= '1';
        current_state <= correzione;
    else
        current_state <= add;
    end if;

when add =>
    reset_out <= '0';
    en_SR <= '1';
    en_FP <= '0';
    en_C <= '0';
    en_R <= '1';
    sel <= q0;
    sub <= '0';
    load_sp <= '0';
    load <= '0';
    shift <= '1';
    stop_cu <= '0';
    stop <= '0';
    f_shift <= '0';
    current_state <= shift_s;

when shift_s =>
    reset_out <= '0';
    en_SR <= '1';
    en_C <= '0';
    en_R <= '1';
    en_FP <= '0';
    sel <= q0;
    load <= '0';
    load_sp <= '0';
    sub <= '0';
    shift <= '0';
    f_shift <= '0';
    stop_cu <= '0';
    stop <= '0';
    current_state <= acquisizione;

```

```

when correzione =>
    reset_out <= '0';
    en_SR <= '1';
    en_C <= '0';
    en_FP <= '0';
    en_R <= '1';
    shift <= '0';
    f_shift <= '1';
    sub <= '0';
    load <= '0';
    load_sp <= '0';
    sel <= q0;
    stop <= '1';
    stop_cu <= '0';
    current_state <= f_shift_s;

when f_shift_s =>
    reset_out <= '0';
    en_SR <= '0';
    en_FP <= '0';
    en_R <= '1';
    en_C <= '0';
    shift <= '0';
    f_shift <= '0';
    sel <= q0;
    stop <= '0';
    stop_cu <= '1';
    load <= '0';
    load_sp <= '0';
    sub <= '0';
    current_state <= idle;
end case;
end if;
end process;
end Behavioral;

```

Infine, secondo un approccio strutturale abbiamo composto le due unità realizzate in modo tale da implementare il **moltiplicatore di Robertson**. Si può notare la presenza di segnali intermedi per modellare lo scambio di segnali tra unità operativa e unità di controllo.

```

entity Robertson is
  Port (
    clock_in: in std_logic;
    reset_in: in std_logic;
    Stop: out std_logic;
    Stop_cu: out std_logic;
    X: in std_logic_vector(7 downto 0);
    Y: in std_logic_vector(7 downto 0);
    start: in std_logic;
    result: out std_logic_vector(15 downto 0)
  );
end Robertson;

architecture Structural of Robertson is

```

```

component UnitaOperativa is
    Port (
        clk: in std_logic;
        rst: in std_logic;
        X: in std_logic_vector(7 downto 0);
        Y: in std_logic_vector(7 downto 0);
        shift: in std_logic;
        f_shift: in std_logic;
        load: in std_logic;
        load_sp: in std_logic;
        en_SR: in std_logic;
        en_FP: in std_logic;
        en_R: in std_logic;
        en_C: in std_logic;
        count: out integer;
        count_end: out std_logic;
        sub: in std_logic;
        sel: in std_logic;
        y_out: out std_logic_vector(15 downto 0)
    );
end component;

component UnitaDiControllo is
    Port (
        clock: in std_logic;
        reset: in std_logic;
        q0: in std_logic;
        start: in std_logic;
        count_end: in std_logic;
        en_C: out std_logic;
        en_SR: out std_logic;
        en_R: out std_logic;
        en_FP: out std_logic;
        load_sp: out std_logic;
        load: out std_logic;
        sel: out std_logic;
        reset_out: out std_logic;
        sub: out std_logic;
        stop: out std_logic;
        shift: out std_logic;
        f_shift: out std_logic;
        stop_cu: out std_logic
    );
end component;

signal q0_temp: std_logic;
signal count_end_temp: std_logic;
signal en_C_temp: std_logic;
signal en_SR_temp: std_logic;
signal en_R_temp: std_logic;
signal en_FP_temp: std_logic;
signal load_sp_temp: std_logic;
signal load_temp: std_logic;
signal sel_temp: std_logic;
signal reset_out_temp: std_logic;
signal sub_temp: std_logic;
signal shift_temp: std_logic;
signal f_shift_temp: std_logic;
signal count_temp: integer;
signal y_temp: std_logic_vector(15 downto 0);
begin

    UO: UnitaOperativa
        port map (
            clk => clock_in,
            rst => reset_out_temp,
            X => X,
            Y => Y,
            shift => shift_temp,
            f_shift => f_shift_temp,
            load => load_temp,
            load_sp => load_sp_temp,
            en_SR => en_SR_temp,
            en_FP => en_FP_temp,
            en_R => en_R_temp,
            en_C => en_C_temp,
            count => count_temp,
            count_end => count_end_temp,
            sub => sub_temp,
            sel => sel_temp,
            y_out => y_temp
        );

```

```

UC: UnitaDiControllo
port map (
    clock => clock_in,
    reset => reset_in,
    q0 => q0_temp,
    start => start,
    count_end => count_end_temp,
    en_C => en_C_temp,
    en_SR => en_SR_temp,
    en_R => en_R_temp,
    en_FP => en_FP_temp,
    load_sp => load_sp_temp,
    load => load_temp,
    sel => sel_temp,
    reset_out => reset_out_temp,
    sub => sub_temp,
    stop => Stop,
    shift => shift_temp,
    f_shift => f_shift_temp,
    stop_cu => Stop_cu
);
q0_temp <= y_temp(0);
result <= y_temp;

end Structural;

```

## 11.4 Simulazione

Per effettuare la simulazione è stato realizzato il testbench sottostante. Abbiamo fissato il periodo del clock con rispettivo process che definisce il comportamento del segnale stesso. Inoltre, sono stati determinati tutti i segnali responsabili della simulazione, comprese le due stringhe da moltiplicare. Dopo 200 ns, viene alzato e poi successivamente abbassato il segnale di start, dando il via alla moltiplicazione.

```

entity TB_robert is
end TB_robert;

architecture Behavioral of TB_robert is

component Robertson is
    port (
        clock_in: in std_logic;
        reset_in: in std_logic;
        Stop: out std_logic;
        Stop_cu: out std_logic;
        X: in std_logic_vector(7 downto 0);
        Y: in std_logic_vector(7 downto 0);
        start: in std_logic;
        result: out std_logic_vector(15 downto 0)
    );
end component;

signal clk: std_logic;
signal rst: std_logic;
signal st: std_logic;
signal st_cu: std_logic;
signal sta: std_logic := '0';
signal x1: std_logic_vector(7 downto 0) := "01100000";
signal x2: std_logic_vector(7 downto 0) := "00000011";
signal output: std_logic_vector(15 downto 0);
constant CLK_period : time := 10 ns;

begin

```

```

uut: Robertson
port map(
clock_in => clk,
reset_in => rst,
Stop => st,
Stop_cu => st_cu,
X => x1,
Y => x2,
start => sta,
result => output
);

CLK_process :process
begin
clk <= '0';
wait for CLK_period/2;
clk <= '1';
wait for CLK_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
wait for 100 ns;

wait for CLK_period*10;

sta <= '1';

wait for 50 ns;
sta <= '0';

wait;
end process;

end architecture Behavioral;

```

X è pari a 01100000 (96 in decimale) e Y è pari a 00000011 (3 in decimale). Il risultato deve essere pari a 288. Com'è possibile vedere dalla figura 11.4 l'output della macchina è coerente con il risultato della moltiplicazione.

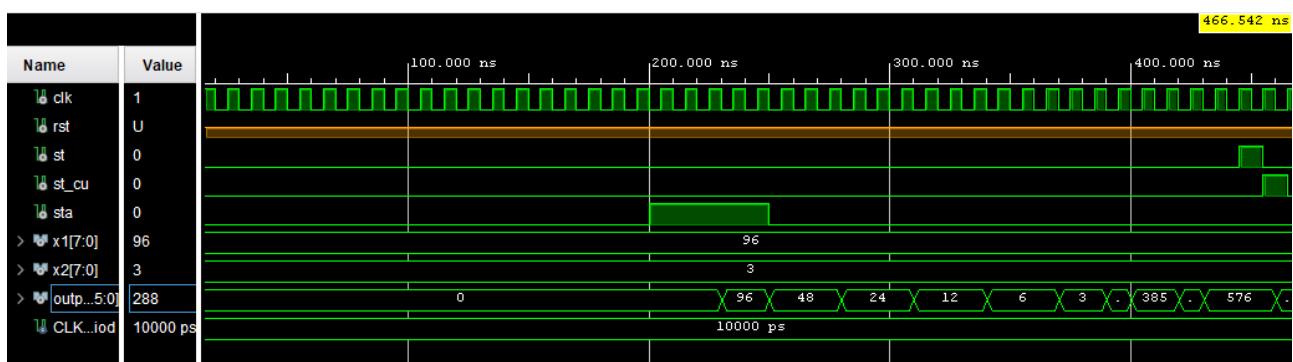


Fig. 11.4 Simulazione moltiplicazione

# 12. Guess The Sequence

## 12.1 Traccia

Progettare, implementare in VHDL e testare mediante simulazione (e, optionalmente, mediante sintesi su board), un sistema le cui specifiche siano definite dallo studente e rientrino in una delle seguenti tipologie:

- a. Modifica di esercizi già proposti (processore, rete di interconnessione o interfaccia seriale) mediante aggiunta/aggiornamento di funzionalità.
  - Esempio: si potrebbe pensare di modificare l'interfaccia seriale aggiungendo segnali specifici per l'handshaking fra due entità.
- b. Progetto di sistemi che assolvono a specifici compiti noti
  - Esempio: si potrebbe pensare di implementare una specifica macchina aritmetica non trattata a lezione, una funzione crittografica, una rete neurale, ecc.
- c. Progetto di sistemi che integrano opportunamente componenti visti a lezione (contatori, registri, macchine aritmetiche, ecc.).
  - A titolo di esempio, è possibile fare riferimento ai seguenti due esercizi:
    - i. Progettare un sommatore di byte seriale (le cifre degli addendi devono essere fornite serialmente a coppie alla macchina) a partire da un sommatore di bit. Il sommatore deve terminare le sue operazioni appena il valore temporaneo della somma diventa maggiore di un valore M fornito in input.
    - ii. Si consideri un nodo A che contiene una memoria ROM di N ( $N \geq 4$ ) locazioni da 8 bit ciascuna. Progettare un sistema in grado di trasmettere mediante handshaking completo tutti i valori strettamente positivi contenuti nella memoria di A ad un nodo B. Il nodo B, ricevuti i valori da A, li trasmetterà ad un nodo C mediante una comunicazione parallela con handshaking.

## 12.2 Soluzione

Come esercizio libero è stato scelto di ricreare un vecchio gioco anni 90 utilizzando gli strumenti a nostra disposizione, il gioco in questione è Simon Says. Tale gioco consiste nel riprodurre nello stesso ordine, tramite la pressione di pulsanti, la sequenza mostrata. È stato utilizzato il display seven segment, per mostrare le varie sequenze, con lo scopo di cimentarci anche nella modifica del cathodes manager, di conseguenza i valori possibili saranno destra, sinistra, sopra e sotto; inoltre quattro dei 5 pulsanti presenti sulla board vengono utilizzati dall'utente per riprodurre la sequenze e il bottone centrale per avviare la partita.

Le caratteristiche del sistema implementato sono:

- Aumento della difficoltà tramite l'incremento della velocità di comparsa delle sequenze
- Azzeramento del punteggio in caso di errori
- Generazione pseudo-casuale delle sequenze mostrate nei vari livelli
- Tempo massimo per la riproduzione della sequenza
- Dieci livelli disponibili
- Visualizzazione del punteggio sul display
- Visualizzazione del risultato del singolo match sui led RGB e display
- Numero fisso di valori mostrati per ogni sequenza, ovvero 4

Per la realizzazione di questo sistema sono state implementati due componenti fondamentali: unità operativa e unità di controllo. L'unità operativa a sua volta è composta da due grandi sub-unità: Visual e Player che rispettivamente contengono i componenti necessari alla corretta visualizzazione delle sequenze e al corretto funzionamento della parte utente. Entrambe le unità sono state implementate tramite approccio strutturale, di

conseguenza contengono al loro interno componenti più piccoli (che vedremo nel dettaglio nei paragrafi successivi), tale approccio ci ha permesso di avere una maggiore modularità e simmetria dell'intero sistema, permettendoci di simulare i vari componenti come se fossero sistemi isolati, così da prevenire eventuali errori che sarebbero potuti sorgere in fase di test quando il sistema era ormai completo.

### 12.3 Schematici

Negli schemi riportati in questo paragrafo non è stato inserito il reset né le uscite cathodes e anodes per una questione di visibilità. Ma ogni componente, dove è necessario ha un reset.

#### Visual

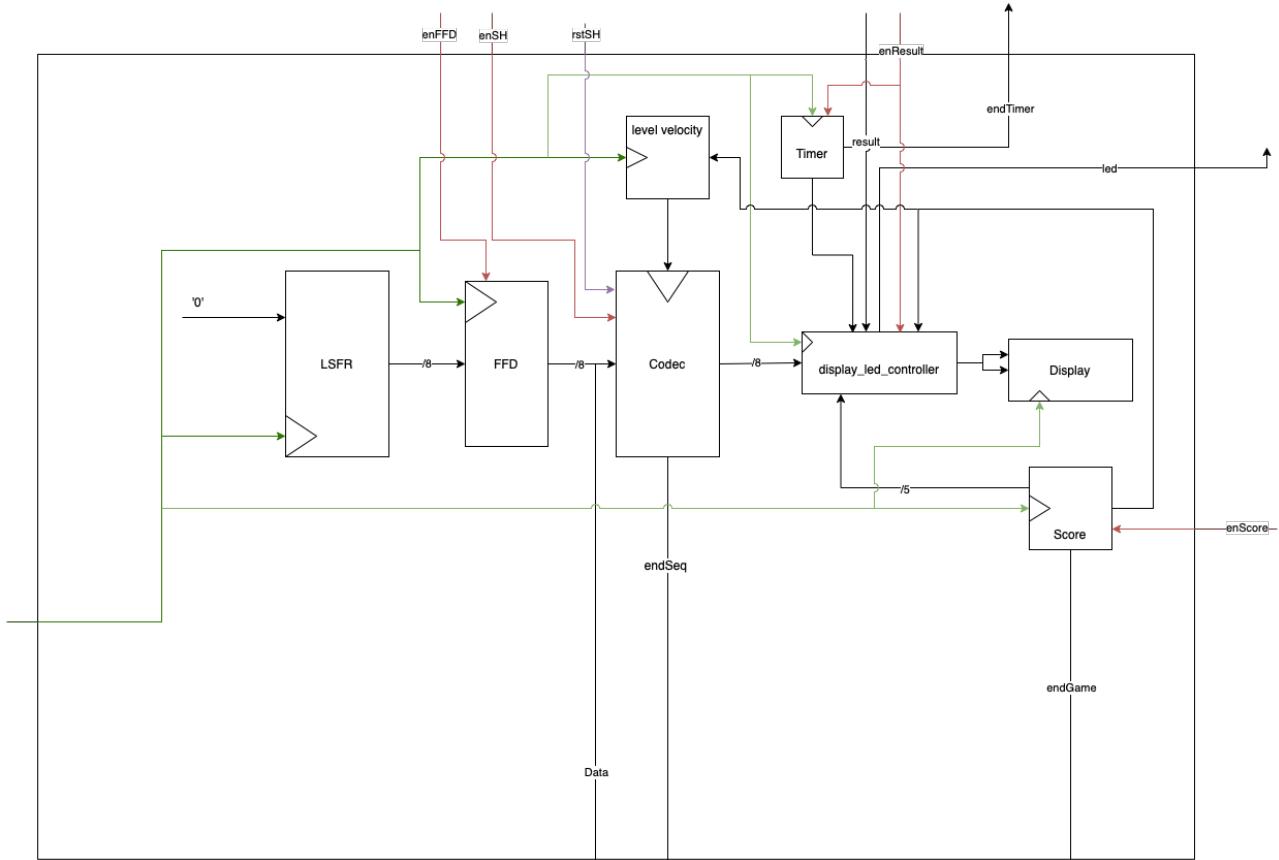


Figura 12. 1: Schema unità Visual

Iniziamo col descrivere i componenti che formano l'unità Visual.

Prima di descrivere ogni componente possiamo notare che in rosso sono state indicate le abilitazioni dei singoli pezzi che la richiedono, gestite dall'unità di controllo. In verde invece è indicato il clock entrante nell'unità Visual ed in ingresso a ogni componente.

Partendo da sinistra verso destra procediamo alla descrizione dei vari componenti.

**LSFR** è stato realizzato mediante un approccio comportamentale, la sua funzione è generare delle sequenze pseudo-casuali di 8 bit ad ogni colpo di clock. Vengono generati 8 bit in quanto sono necessari 2 bit per codificare 4 direzioni ed essendo il numero di direzioni visualizzabili per ogni sequenza fissato a 4 sono necessari 2x4 bit ovvero 8 bit.

Il **flip flop** posto alla destra dell'LSFR è di tipo D, il suo compito è quello di memorizzare una delle sequenze generate, nel momento in cui la sua abilitazione è alta. L'obbiettivo è memorizzare per un tempo sufficiente a terminare il match (il quale non è necessariamente costante, in quanto dipende anche dalla velocità del giocatore) la sequenza catturata, in maniera tale, da aumentare la casualità delle sequenze generate e separare

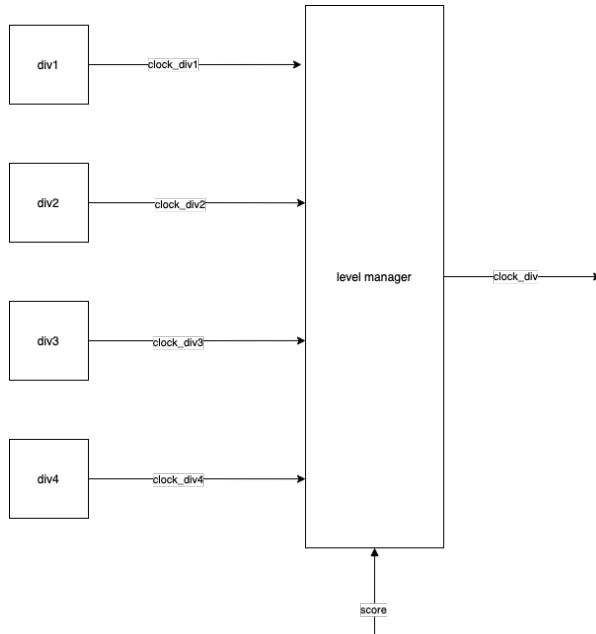
la parte di generazione da quella di visualizzazione. L'uscita del flip flop è collegata ad un codificatore *Codec* ed è posta in output all'unità Visual.

Il **Codec** non è altro che un codificatore, esso preleva sequenza di 8 bit catturata dal flip flop e la codifica in una sequenza di 5 bit. Tale codifica viene effettuata estraendo dalla stringa di 8 bit due bit per volta (in quanto sono sufficienti 2 bit per coprire 4 direzioni) traducendoli nel seguente modo:

- 00 → 01100 (Sinistra)
- 01 → 01101 (Destra)
- 11 → 01111 (Sopra)
- 10 → 01110 (Sotto)

I due bit dalla stringa di 8 vengono estratti e codificati con una frequenza diversa in base al punteggio dell'utente, tale velocità viene stabilita dal blocco *level velocity*. Vengono utilizzati 5 bit in quanto esso rappresenta il numero di bit necessari a codificare i simboli che dovranno attivarsi su un singolo display. Nel paragrafo successivo approfondiremo questo aspetto mediante la visualizzazione del file *cathodes\_manager*. Il Codec è inoltre dotato anche di un reset diverso (filo in viola) dal reset collegato agli altri componenti, tale reset viene utilizzato per fissare in output la stringa “01010”, ovvero la stringa codificata che permette di spegnere l'intero display.

Il blocco **level velocity** gestisce le differenti frequenze di clock in ingresso al *Codec*. Esso è composto da quattro divisori di frequenza e un blocco che ne gestisce l'output in base al punteggio attuale del giocatore, come mostrato in *figura 12.2*.



*Figura 12. 2: Blocco level velocity*

La stringa di 5 bit in uscita al *Codec* è stata usata come ingresso per un ulteriore blocco, ovvero il **display led controller**. Questo blocco dispone di 4 ingressi (oltre il clock): i 5 bit di output del *Codec*, 5 bit di Output del blocco *score*, il risultato del singolo match inviato dall'unità di controllo e un'abilitazione; e presenta tre uscite: 40 bit che rappresentano i valori per l'intero display, 8 bit che rappresentano le cifre del display da abilitare e 3 bit per i led RGB.

Tale componente si occupa di gestire ciò che deve o non deve essere visualizzato sul display a sette segmenti. In particolare, attraverso l'abilitazione data dall'unità di controllo e il risultato della singola partita mostra tramite il display la dicitura “PASS” o “FAIL” accendendo rispettivamente un led verde o rosso. Il tempo per

cui tali parole devono essere visualizzate viene gestito dal blocco *Timer*, finito questo tempo il blocco in questione mostra di nuovo punteggio e una nuova sequenza sul display.

Il **timer** non è altro che un componente formato da un contatore e un divisore di frequenza, al raggiungimento del suo conteggio massimo (che nel nostro caso corrisponde ad 1 secondo) viene inviato un flag all'unità di controllo.

Il **display seven segment** è un componente datoci in dotazione, sono state però effettuate delle modifiche al cathodes\_manager in maniera tale da adattarlo alle esigenze del progetto; infatti, esso presenta un ingresso non più di 32 bit ma di dimensione 40 bit. In seguito, visioneremo più in dettaglio queste modifiche.

Infine, è presente nella struttura un componente denominato **score**, questo blocco non è altro che un contatore. Esso tiene traccia del punteggio ed invia un segnale (topscore) all'unità di controllo nel momento in cui il giocatore raggiunge il punteggio massimo. Inoltre, il punteggio viene inviato tramite una stringa di 5 bit al *display led controller*.

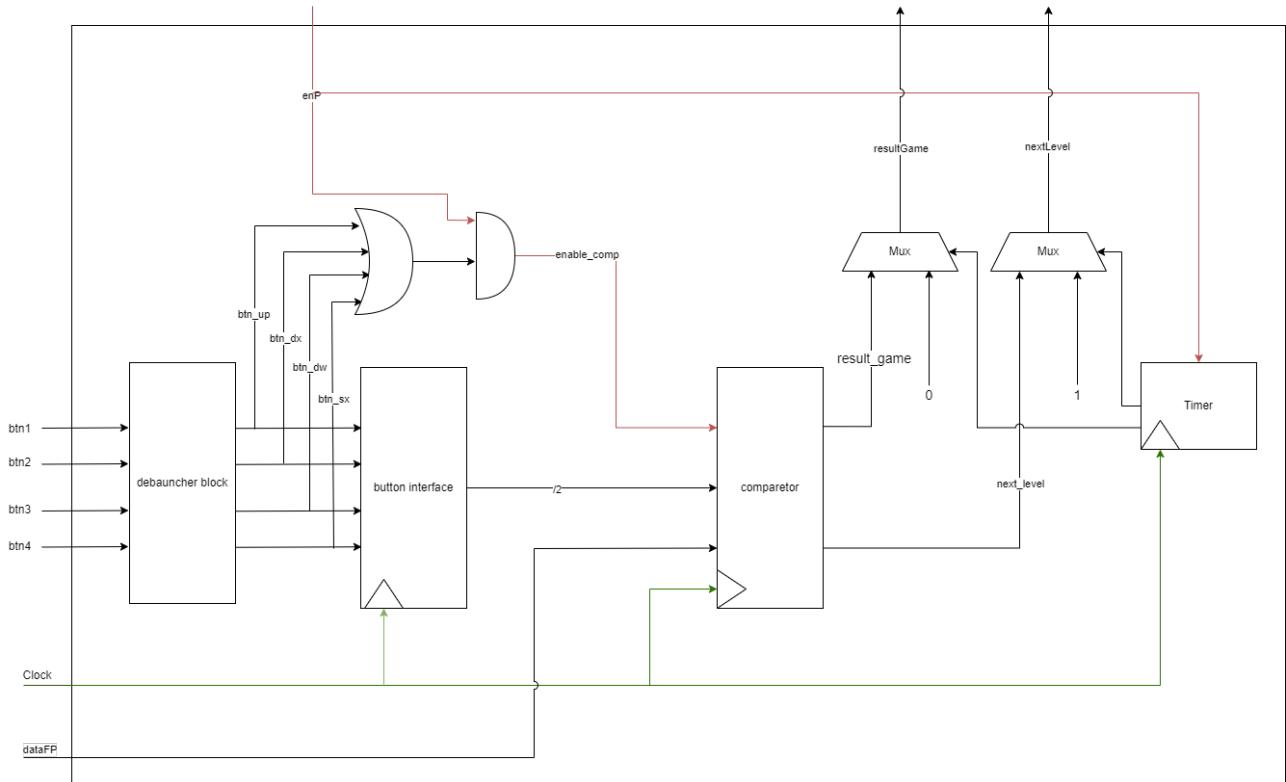


Figura 12. 3: Schema unità Player

## Player

L'unità player si occupa della gestione della parte di interfacciamento con l'utente. Essa prende in ingresso 4 pulsanti (gestiti opportunamente tramite dei debaucher), il dato catturato precedentemente dal flip flop nell'unità Visual e un segnale di abilitazione derivante dall'unità di controllo. In uscita presenta 2 linee, utilizzate dall'UC.

Da sinistra verso destra descriviamo i vari componenti dell'unità.

Il **debaucher block** non è altro che un componente strutturale composto da 4 button debaucher per i rispettivi bottoni in ingresso; quindi, come ci si può aspettare presenta in uscita 4 segnali, ovvero i segnali puliti da oscillazione dei pulsanti. Queste uscite vengono sia utilizzate da un ulteriore componente, cioè il *button interface* e sia per generare l'abilitazione al blocco *comparator* tramite la porta AND.

Il blocco **button interface** ha il compito di codificare la pressione di un pulsante su 2 bit. In particolare, riceve in ingresso 4 segnali associati alla pressione dei 4 pulsanti e produce in output una stringa di 2 bit.

La codifica avviene nel seguente modo:

- btn\_sx → 00
- btn\_dx → 01
- btn\_dw → 10
- btn\_up → 11

Si può notare come ogni associazione pulsante → bit, viene fatta in base alla rispettiva posizione del pulsante.

Il blocco **comparetor** effettua il confronto tra i pulsanti premuti e la sequenza mostrata. Riceve in ingresso la stringa da 8 bit catturata precedentemente dal flip flop nell'unità Visual ed effettua un confronto, estraendo due bit per volta, con la codifica del pulsante utilizzato dall'utente, ricevuta in ingresso dal *button interface*. Se la sequenza dei pulsanti premuti corrisponde alla sequenza mostrata sul display precedentemente, il segnale *result\_game* diviene 1 altrimenti resta 0. Il valore del segnale *next\_level* diventa alto nel momento in cui la partita termina, indipendente dal risultato di quest'ultima e quindi dal valore di *result\_game*.

Inoltre, l'esito della partita viene calcolato in maniera dinamica, ovvero il **comparetor** non attende che l'utente completi la sequenza prima di poterla valutare, ma effettua il confronto pulsante dopo pulsante, nel momento in cui il giocatore sbaglia, la partita viene interrotta e i valori delle uscite vengono modificate opportunamente. Per evitare che il confronto venga effettuato in istanti di tempo errati, il componente è dotato di abilitazione, quest'ultima è calcolata come la AND tra i segnali dei pulsanti e l'abilitazione *enP* proveniente dall'unità di controllo, così facendo il **comparetor** è attivo solo se l'unità di controllo lo richiede e solo nel momento in cui i pulsanti vengono premuti.

Viene riportato in seguito un esempio del funzionamento di questo componente:

- **Sequenza:**

01010011 (La sequenza visualizzata sul display sarà: Sopra-Sinistra-Destra-Destra)

- **Giocatore:**

1° pulsante: btn\_up →

il *button interface* lo codifica con 11 →

entra nel *comparetor* →

il comparetor estrae i primi 2 bit della sequenza (da destra a sinistra) →

confronta i 2 bit estratti con i due bit in input → coincidono quindi continua

2° pulsante: btn\_sx →

il *button interface* lo codifica con 00 →

entra nel *comparetor* →

il comparetor estrae altri 2 bit dalla sequenza (da destra a sinistra) →

confronta i 2 bit estratti con i due bit in input → coincidono quindi continua

3° pulsante: btn\_up →

il *button interface* lo codifica con 11 →

entra nel *comparetor* →

il comparetor estrae altri 2 bit dalla sequenza (da destra a sinistra) →

confronta i 2 bit estratti con i due bit in input → non coincidono →

alza *next\_level* → tiene basso *result\_game*

In uno scenario differente in cui sia il 3° che il 4° pulsante fossero stati giusti il **comparetor** avrebbe alzato sia *next\_level* che *result\_game*.

Nell'unità Player è presente anche un **Timer**. Il timer inizia il conteggio nel momento in cui la sequenza mostrata sul display termina e quindi nel momento in cui il giocatore deve riprodurla. Viene attivato tramite il segnale di abilitazione *enP*.

Esso viene utilizzato per fissare un tempo limite per indovinare la sequenza. Abbiamo settato questo tempo a 7 secondi, dopo i quali il timer alza un flag di *time\_out*. Questo segnale è utilizzato come selezione dei due multiplexer: se fosse 0 allora il multiplexer presenta in uscita i valori di *result\_game* e *next\_level* prodotti dal comparatore; se fosse 1 allora il multiplexer assegna a *resulGame* e *nextLevel* rispettivamente il valore 0 ed 1 per indicare la fine del tempo limite e quindi la sconfitta del giocatore.

## Unità Operativa

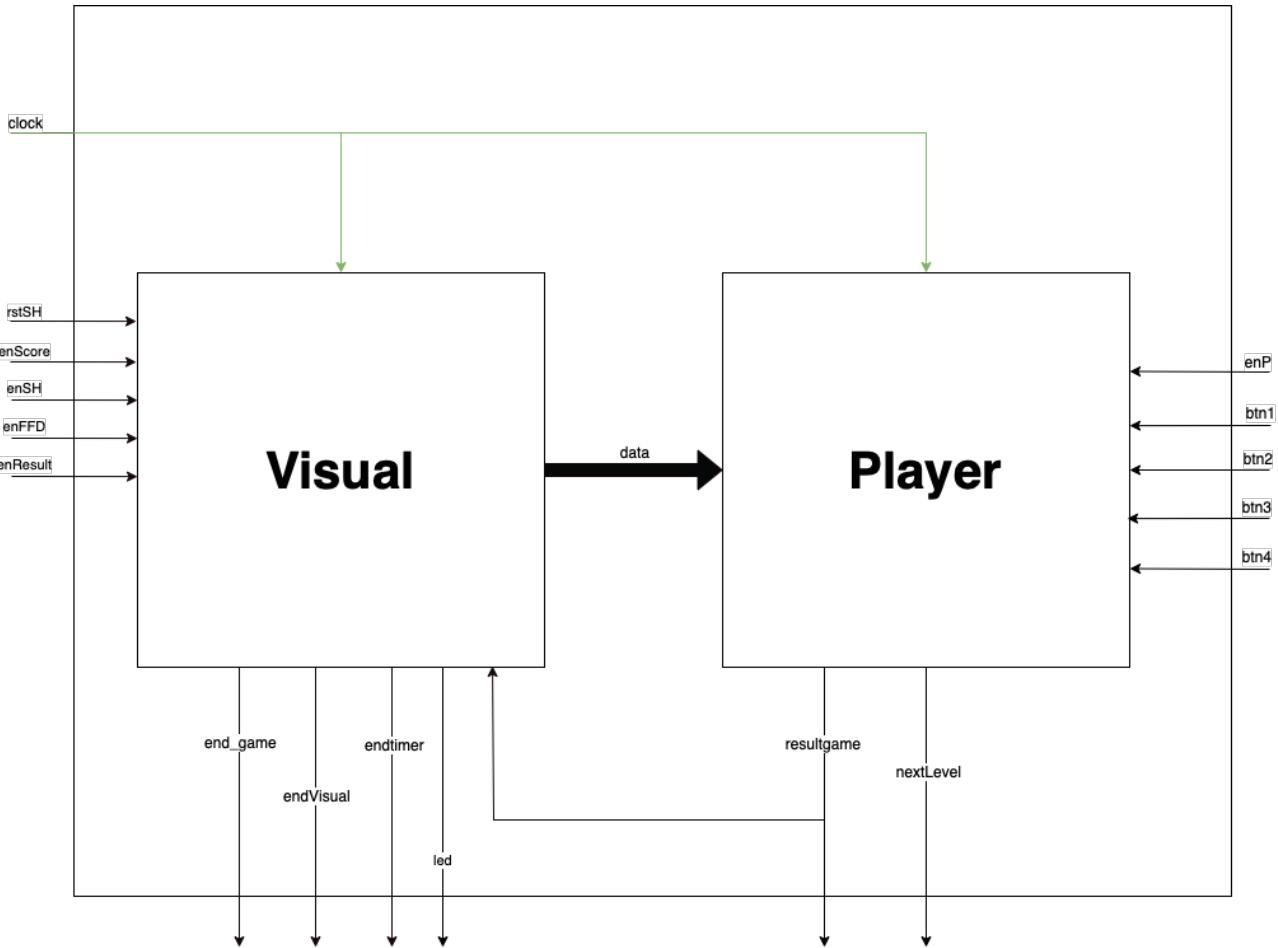


Figura 12. 4: Schema unità operativa

La *figura 12.4* mostra lo schema a blocchi dell’unità operativa: complessamente è dotata di 10 ingressi più il reset e 5 uscite. L’unità Visual trasmette in parallelo la sequenza di bit catturata dal flip flop al blocco Player; invece, il blocco Player invia il segnale *resultGame* sia in uscita all’UO sia al blocco Visual.

## Unità di Controllo

L’unità di controllo si occupa di tempificare l’elaborazione degli input dell’UO tramite l’utilizzo di segnali di abilitazione. Abbiamo scelto di realizzare l’UC tramite logica cablata e quindi descrivendola mediante un automa a stati finiti (ASF). In *figura 12.5* è riportato l’automa.

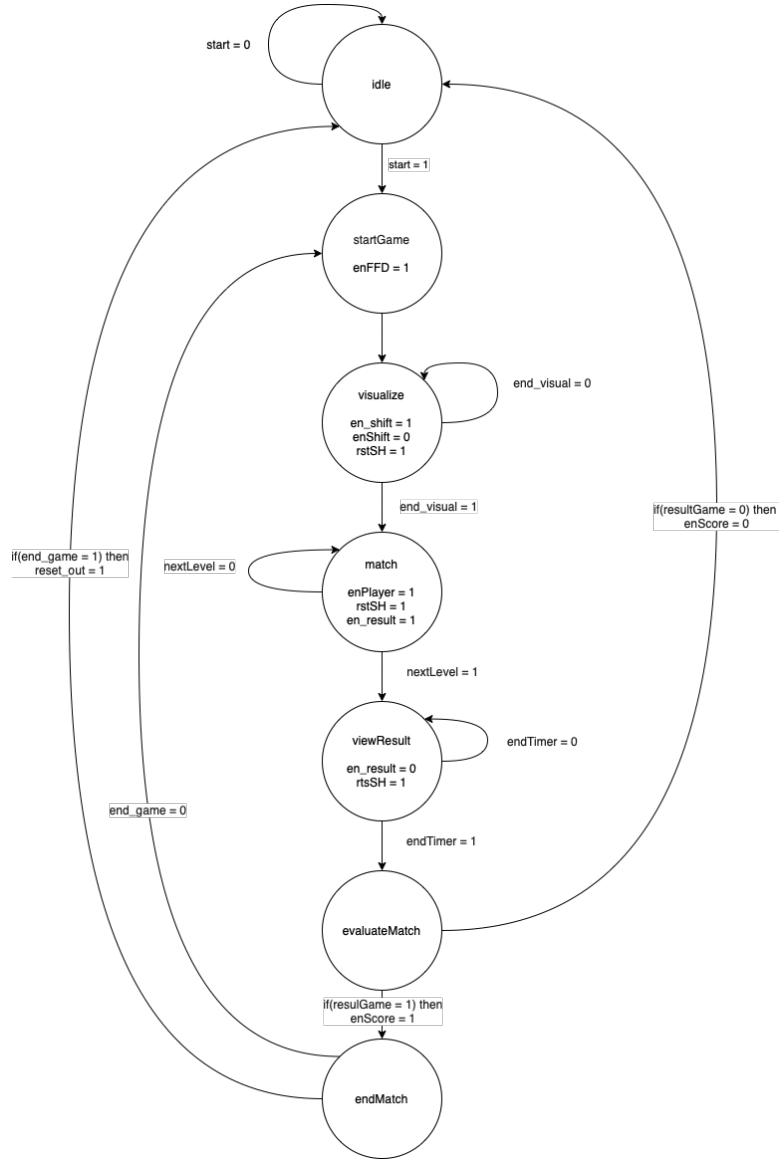


Figura 12. 5: ASF Unità di controllo

Descriviamo quindi in dettaglio gli stati:

- **idle**

Lo stato *idle* è lo stato di partenza di questo automa. In questo stato tutte le abilitazioni sono basse e l'UO viene resettata così da poter partire da uno stato noto. L'unità di controllo resta nello stato *idle* finché non riceve il segnale di *start*.

- **startGame**

In questo stato l'UC abilita il flip flop per permettergli di catturare una sequenza casuale, successivamente transita nello stato *visualize*.

- **visualize**

Inizia quindi la fase di visualizzazione in cui l'abilitazione del flip flop torna bassa così da poter conservare la sequenza catturata. Viene abilitato invece il componente codec.

Al termine della visualizzazione, ovvero nel momento in cui il codec invia il segnale di *end\_visual*, l'unità di controllo abbassa l'abilitazione al codec e ne effettua il reset per spegnere la cifra del display

su cui vengono visualizzate le sequenze, il reset da questo momento in poi resterà alto fino al termine della partita corrente. Infine, transita nello stato *match*.

- **match**

In questo stato viene abilitata l'unità player. L'abilitazione *en\_Player* diventa 1 così da attivare il timer e il comparatore. L'unità di controllo attende che la partita termini e che quindi il segnale *nextLevel* sia alto. Al termine del match l'UC alza il flag *en\_result* così che il componente *display\_led\_controller* possa mostrare sul display il risultato della partita, in fine transita nello stato *viewResult*.

- **viewResult**

Lo stato *viewResult* è uno stato di attesa in cui l'unità di controllo viene ritardata fino al termine della visualizzazione del risultato (PASS o FAIL). Transita nello stato successivo nel momento in cui *end\_timer* è 1.

- **evaluateMatch**

La funzione di questo stato è valutare il risultato della partita in base al valore passato dall'UO. Se il risultato è positivo (*resultGame* = 1) il contatore che tiene traccia del punteggio del giocatore viene incrementato e quindi l'abilitazione di conteggio viene posta a 1, l'UC transita nello stato successivo. Se il risultato è negativo l'abilitazione di conteggio resta 0 e l'unità di controllo transita nello stato *idle* così da poter iniziare una nuova partita.

- **endMatch**

Lo stato *endMatch* valuta se il gioco è terminato o meno. In altre parole, se il giocatore ha raggiunto il punteggio massimo (*end\_game* = 1) l'UO viene resettata e l'UC transita nello stato *idle*, altrimenti l'UC transita in *startGame*.

## Guess The Sequence

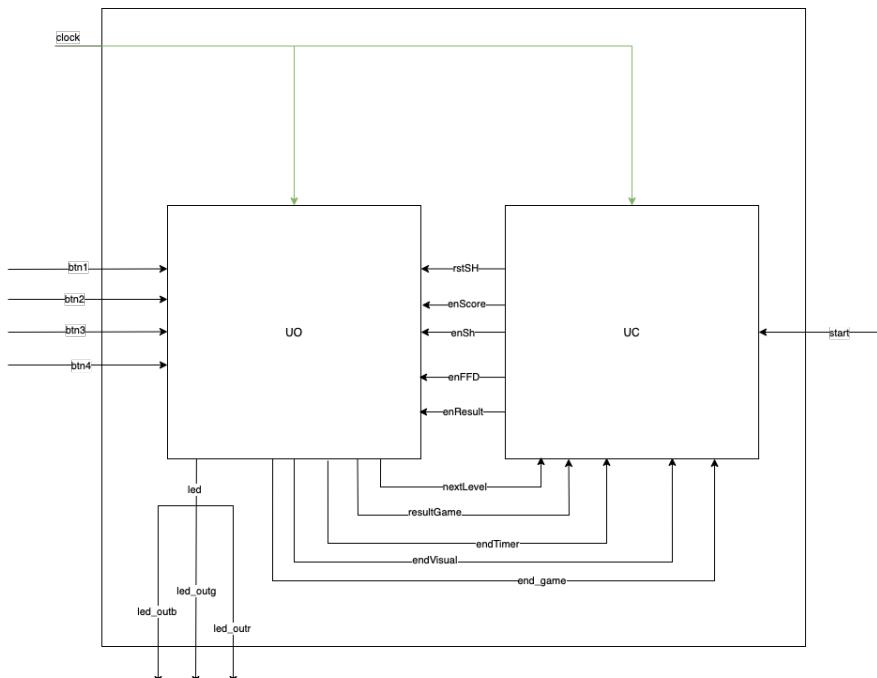


Figura 12. 6: Schema sistema completo

In figura 12.6 è rappresentato lo schema del sistema complessivo, ovvero composto da unità operativa e unità di controllo. Il sistema *Guess The Sequence* è caratterizzato da 6 ingressi: 4 pulsanti e il pulsante di start; e 3 uscite ovvero i valori rosso, verde e blu del led RGB. Nello schema non sono state rappresentate le opportune uscite per il display (cathodes e anodes)

## 12.4 Codice

Analizziamo in dettaglio il codice di ogni componente, iniziando con i componenti che strutturano l'unità Visual.

### LSFR

```

33  entity lsfr2 is
34      Port ( clock : in  STD_LOGIC;
35          reset : in  STD_LOGIC;
36          lsfr_out : out  STD_LOGIC_VECTOR (7 downto 0));
37 end lsfr2;
38
39 architecture Behavioral of lsfr2 is
40
41
42
43
44 signal lsfr_new: std_logic_vector (7 downto 0):="11010010";
45 signal lsfr: std_logic_vector ( 7 downto 0):="11010011";
46 --4 downto 0
47 begin
48
49 lsfr_update: process (reset,lsfr)
50
51 begin
52     lsfr_new (7 downto 1) <= lsfr (6 downto 0);
53     lsfr_new (0) <= not(lsfr(1) xor lsfr(7));
54
55 end process lsfr_update;
56
57
58 lsfr8: process(reset,clock)
59 begin
60     if (rising_edge (clock)) then
61         if (reset='1') then
62             lsfr <= "00000001";
63         else
64             lsfr <= lsfr_new;
65         end if;
66     end if;
67 end process lsfr8;
68 lsfr_out <= lsfr;
69 end Behavioral;

```

Figura 12. 7: Codice LSFR

Possiamo notare come l'entity non ha ingressi (oltre clock e reset) in quanto questo componente ha il solo compito di generare delle sequenze pseudo-casuali. LSFR è composto da due process: il primo sensibile al segnale lsfr, l'altro sensibile al clock (entrambi sensibili al reset). Questo permette di aumentarne la casualità in quanto i processi vengono eseguiti in istanti di tempo diversi. Il primo process non fa altro che aggiornare il segnale lsfr\_new tramite i valori del segnale lsfr e mediante le proprietà della xor; il secondo invece assegna per intero il segnale lsfr\_new a lsfr. Per aumentarne ulteriormente la casualità, a differenza degli altri componenti i quali possiedono un'abilitazione controllata dall'UC, questo componente resta sempre attivo e genera sempre sequenze, di conseguenza la sequenza catturata dipende anche dal tempo impiegato per

visualizzare la sequenza (ricordiamo che la velocità cambia) e dal tempo impiegato dal giocatore per indovinarla.

## FFD

Il flip flop è stato realizzato mediante un approccio comportamentale. Nel momento in cui l'abilitazione è alta cattura il dato in ingresso e lo presenta in uscita.

```

34  entity ffd is
35      Port ( clock : in STD_LOGIC;
36          reset : in STD_LOGIC;
37          input : in STD_LOGIC_VECTOR (7 downto 0);
38          data : out STD_LOGIC_VECTOR (7 downto 0);
39          enable: in STD_LOGIC);
40  end ffd;
41
42  architecture Behavioral of ffd is
43
44  begin
45  process(clock, reset)
46  begin
47  if reset = '1' then
48      data <= "00000000";
49  elsif rising_edge(clock) then
50  if enable = '1' then
51      data <= input;
52  end if;
53  end if;
54
55 end process;
56
57 end Behavioral;

```

*Figura 12. 8: VHDL flip flop*

## CODEC

Come già detto precedentemente questo componente codifica a due a due la sequenza di 8 bit in ingresso. Viene utilizzato sia il reset (utile all'UC) sia un flag sincrono (utile per simulare l'effetto spento-accesso nella visualizzazione della sequenza) per spegnere la cifra su cui viene visualizzata la sequenza. Possiede un contatore interno per tenere conto del numero di bit codificati, prendendo una coppia di bit a ogni colpo di clock (ricordiamo che il clock in ingresso a tale componente è più lento in quanto corrisponde all'uscita di un divisore di frequenza) il contatore viene incrementato di due ad ogni ciclo. Il codice è mostrato in *figura 12.9*.

## LEVEL VELOCITY

Il componente level velocity è stato realizzato mediante un approccio strutturale. I componenti che ne fanno parte sono 4 divisorie di frequenza, i quali regolano la velocità con cui il codec effettua la codifica e un level manager, il cui compito è di valutare il punteggio attuale del giocatore per poi presentare in uscita la frequenza di clock corretta. Quest'ultimo è stato realizzato mediante un approccio comportamentale e il codice è mostrato in *figura 12.11*.

## DISPLAY LED CONTROLLER

Il display led controller si occupa di smistare le varie informazioni sul display. In base alle abilitazioni ricevute dell'unità di controllo invia informazioni differenti al display. Di default ad ogni colpo di clock mostra il punteggio e l'eventuale sequenza, nel momento in cui però il flag *enable\_result* inviato dall'UC si alza esso non mostra più il punteggio o la sequenza ma valuta il risultato della partita tramite il segnale *result\_game* e in base al valore di quest'ultimo mostra sul display PASS o FAIL selezionando le opportune cifre da abilitare.

```

34 entity codec is
35     Port ( clock : in STD_LOGIC; --CLOCK DIVISORE 0.5 S
36         reset : in STD_LOGIC;
37         input : in STD_LOGIC_VECTOR (7 downto 0);
38         data : out STD_LOGIC_VECTOR (4 downto 0);
39         endSeq : out STD_LOGIC;
40         enable: in STD_LOGIC);
41 end codec;
42
43 architecture Behavioral of codec is
44     begin
45         process(clock, reset)
46             variable count: integer := 0;
47             variable flag: integer := 0;
48             begin
49                 if reset = '1' then
50                     count := 0;
51                     endSeq <= '0';
52                     data <= "01010"; --A, SPEGNI
53                 elsif rising_edge(clock) then
54                     if enable = '1' then
55                         if(flag = 1) then
56                             data <= "01010";
57                             flag := 0;
58                         else
59                             if(count > 7) then
60                                 endSeq <= '1';
61                                 count := 0;
62                             else endSeq <= '0';
63                             if(input(count)='0' and input(count+1)='0') then
64                                 data <= "01100"; --SX
65                                 flag := 1;
66                             elsif (input(count)='0' and input(count+1)='1') then
67                                 data <= "01101"; --DX
68                                 flag := 1;
69                             elsif (input(count)='1' and input(count+1)='0') then
70                                 data <= "01110"; --DW
71                                 flag := 1;
72                             elsif (input(count)='1' and input(count+1)='1') then
73                                 data <= "01111"; --UP
74                                 flag := 1;
75                         end if;
76                         count := count + 2;
77                     end if;
78                 end if;
79             end if;
80         end process;
81
82
83
84 end Behavioral;
85

```

Figura 12. 9: VHDL codec

```

34 entity level_velocity is
35     Port (
36         clock: in std_logic;
37         reset: in std_logic;
38         score: in std_logic_vector(4 downto 0);
39         clock_div: out std_logic
40     );
41 end level_velocity;
42
43 architecture Structural of level_velocity is
44     component div is
45         generic(
46             clock_frequency_in : integer := 50000000;
47             clock_frequency_out : integer := 5000000
48         );
49     Port ( clk_in : in STD_LOGIC;
50         rst_in : in STD_LOGIC;
51         clk_out : out STD_LOGIC
52     );
53 end component;
54
55     component level_manager is
56         Port (
57             clock_div1: in std_logic;
58             clock_div2: in std_logic;
59             clock_div3: in std_logic;
60             clock_div4: in std_logic;
61             score: in std_logic_vector(4 downto 0);
62             clock_out: out std_logic
63         );
64     end component;
65
66
67     signal clock_temp: std_logic_vector(3 downto 0);
68
69
74     div1: div
75         generic map(clock_frequency_in => 100000000,
76             clock_frequency_out => 2
77         )
78         port map(clk_in => clock,
79             rst_in => reset,
80             clk_out => clock_temp(0)
81         );
82
83
84
85     div2: div
86         generic map(clock_frequency_in => 100000000,
87             clock_frequency_out => 3
88         )
89         port map(clk_in => clock,
90             rst_in => reset,
91             clk_out => clock_temp(1)
92         );
93
94
95
96     div3: div
97         generic map(clock_frequency_in => 100000000,
98             clock_frequency_out => 5
99         )
100        port map(clk_in => clock,
101            rst_in => reset,
102            clk_out => clock_temp(2)
103        );
104
105
106
107     div4: div
108         generic map(clock_frequency_in => 100000000,
109             clock_frequency_out => 6
110         )
111         port map(clk_in => clock,
112             rst_in => reset,
113             clk_out => clock_temp(3)
114         );
115
116
117
118     lv_man: level_manager
119         port map(clock_div1 => clock_temp(0),
120             clock_div2 => clock_temp(1),
121             clock_div3 => clock_temp(2),
122             clock_div4 => clock_temp(3),
123             score => score,
124             clock_out => clock_div
125         );
126
127
128 end Structural;

```

Figura 12. 10: VHDL level velocity

```

34 entity level_manager is
35   Port (
36     clock_div1: in std_logic;
37     clock_div2: in std_logic;
38     clock_div3: in std_logic;
39     clock_div4: in std_logic;
40     score: in std_logic_vector(4 downto 0);
41     clock_out: out std_logic
42   );
43 end level_manager;
44
45 architecture Behavioral of level_manager is
46
47 begin
48 process(score)
49 variable count: integer := 0;
50 begin
51   count := to_integer(unsigned(score));
52   if(count >=0 and count < 3) then
53     clock_out <= clock_div1;
54   elsif(count >= 3 and count < 6) then
55     clock_out <= clock_div2;
56   elsif(count >= 6 and count < 8) then
57     clock_out <= clock_div3;
58   elsif(count >= 8 and count < 10) then
59     clock_out <= clock_div4;
60   end if;
61 end process;
62 end Behavioral;

```

*Figura 12. 11: VHDL level manager*

*Figura 12. 12: VHDL display led controller*

## TIMER

```

34  entity timer is
35      generic(
36          M: integer := 7
37      );
38      Port (
39          clock: in std_logic;
40          reset: in std_logic;
41          end_time: out std_logic;
42          enable: in std_logic
43      );
44  end timer;
45
46  architecture Structural of timer is
47
48  component contatore_timer is
49      generic(
50          N: integer := 7
51      );
52      Port ( clock: in std_logic;
53          enable : in std_logic;
54          reset: in std_logic;
55          timeout: out std_logic
56      );
57  end component;
58
59  component div is
60      generic(
61          clock_frequency_in : integer := 50000000;
62          clock_frequency_out : integer := 5000000
63      );
64      Port ( clk_in : in STD_LOGIC;
65          rst_in : in STD_LOGIC;
66          clk_out : out STD_LOGIC
67      );
68  end component;
69
70  signal clock_div: std_logic;
71
72  begin
73
74  divisore_fre: div
75      generic map(
76          clock_frequency_in => 100000000,
77          clock_frequency_out => 1
78      )
79      port map ( clk_in => clock,
80                 rst_in => reset,
81                 clk_out => clock_div
82     );
83
84  c_timer: contatore_timer
85      generic map(
86          N => M
87      )
88      Port map ( clock => clock_div,
89                 enable => enable,
90                 reset => reset,
91                 timeout => end_time
92     );
93
94  end Structural;

```

Figura 12. 13: VHDL timer

Il timer è stato realizzato utilizzando un contatore e un divisore di frequenza così che esso possa contare ogni secondo. Il conteggio massimo è stabilito all'istanziazione del componente mediante l'intero M in generic.

## SEVEN SEGMENT DISPLAY (CATHODES MANAGER)

E' stata riportata la parte di codice modificato del file cathodes\_manager.

```

32  entity cathodes_manager is
33      Port ( counter : in STD_LOGIC_VECTOR (2 downto 0);
34          value : in STD_LOGIC_VECTOR (39 downto 0); --dato da mostrare sugli 8 display
35          dots : in STD_LOGIC_VECTOR (7 downto 0); --configurazione punti da ascendere
36          cathodes : out STD_LOGIC_VECTOR (7 downto 0)); --sono i 7 catodi più il punto
37  end cathodes_manager;
38
39  architecture Behavioral of cathodes_manager is
40
41      constant zero : std_logic_vector(6 downto 0) := "1000000";
42      constant one : std_logic_vector(6 downto 0) := "1111001";
43      constant two : std_logic_vector(6 downto 0) := "0100100";
44      constant three : std_logic_vector(6 downto 0) := "0011001";
45      constant four : std_logic_vector(6 downto 0) := "0001001";
46      constant five : std_logic_vector(6 downto 0) := "0010010";
47      constant six : std_logic_vector(6 downto 0) := "0001010";
48      constant seven : std_logic_vector(6 downto 0) := "1111000";
49      constant eight : std_logic_vector(6 downto 0) := "0000000";
50      constant nine : std_logic_vector(6 downto 0) := "0010000";
51      constant a : std_logic_vector(6 downto 0) := "0001000";
52      constant b : std_logic_vector(6 downto 0) := "0000011";
53
54      constant off : std_logic_vector(6 downto 0) := "1111111";
55      constant sx : std_logic_vector(6 downto 0) := "1001111"; --SX
56      constant dx : std_logic_vector(6 downto 0) := "1110001"; --DX
57      constant dw : std_logic_vector(6 downto 0) := "1110111"; --DW
58      constant up : std_logic_vector(6 downto 0) := "1111110"; --UP
59
60      constant P : std_logic_vector(6 downto 0) := "0001100";
61      constant A : std_logic_vector(6 downto 0) := "0001000";
62      constant S : std_logic_vector(6 downto 0) := "0010010";
63      constant F : std_logic_vector(6 downto 0) := "0001110";
64      constant I : std_logic_vector(6 downto 0) := "1001111";
65      constant L : std_logic_vector(6 downto 0) := "1000111";
66
67      alias digit_0 is value (4 downto 0);
68      alias digit_1 is value (5 downto 5);
69      alias digit_2 is value (14 downto 10);
70      alias digit_3 is value (19 downto 15);
71      alias digit_4 is value (23 downto 20);
72      alias digit_5 is value (25 downto 19);
73      alias digit_6 is value (34 downto 30);
74      alias digit_7 is value (39 downto 35);
75
76  begin
77
78      seven_segment_decoder_process: process(nibble)
79      begin
80
81          case nibble is
82              when "00000" => cathodes_for_digit <= zero;
83              when "00001" => cathodes_for_digit <= one;
84              when "00010" => cathodes_for_digit <= two;
85              when "00011" => cathodes_for_digit <= three;
86              when "00100" => cathodes_for_digit <= four;
87              when "00101" => cathodes_for_digit <= five;
88              when "00110" => cathodes_for_digit <= six;
89              when "00111" => cathodes_for_digit <= seven;
90              when "01000" => cathodes_for_digit <= eight;
91              when "01001" => cathodes_for_digit <= nine;
92
93              when "01010" => cathodes_for_digit <= off;
94              when "01011" => cathodes_for_digit <= b;
95
96              when "01100" => cathodes_for_digit <= sx;
97              when "01101" => cathodes_for_digit <= dx;
98              when "01110" => cathodes_for_digit <= dw;
99              when "01111" => cathodes_for_digit <= up;
100
101              when "10000" => cathodes_for_digit <= P;
102              when "10001" => cathodes_for_digit <= A;
103              when "10010" => cathodes_for_digit <= S;
104              when "10011" => cathodes_for_digit <= F;
105              when "10100" => cathodes_for_digit <= I;
106              when "10101" => cathodes_for_digit <= L;
107
108              when others => cathodes_for_digit <= (others => '0');
109
110      end case;
111
112  end process seven_segment_decoder_process;

```

Figura 12. 14: VHDL cathodes manager

La prima modifica effettuata è stata quella di aumentare il numero di bit in ingresso passando da 36 a 40 (notiamo infatti che VALUE è un *std\_logic\_vector(39 downto 0)*). È stata effettuata questa modifica in quanto è sorta la necessità dell'aggiunta di nuovi simboli da visualizzare, era necessario quindi aumentare il numero di bit per ogni cifra:

- Numeri da 0 a 9 per tener conto del punteggio
- 4 simboli per indicare le direzioni di ogni sequenza
- 6 lettere per visualizzare la parola PASS o la parola FAIL

Per un totale di 20 simboli, codificabile con 5 bit ( $2^5 = 36$ ) avendo 8 display sono necessari 40 bit. I simboli aggiuntivi hanno comportato una ovvia modifica del resto del codice, ogni digit ora possiede non 4 bit ma 5 e il process è stato modificato aggiungendo i casi necessari ed effettuando le opportune modifiche a quelli già esistenti.

## VISUAL

L'unità visual è stata realizzata in maniera strutturale effettuando gli opportuni collegamenti come mostrato nel paragrafo precedente.

```

34  entity visual is
35    Port (
36      clock: in std_logic;
37      reset: in std_logic;
38      enSH: in std_logic;
39      enFFD: in std_logic;
40      rstSH: in std_logic;
41      enScore: in std_logic;
42      data_out: std_logic_vector(7 downto 0);
43      anodes_out: std_logic_vector(7 downto 0);
44      cathodes_out: std_logic_vector(7 downto 0);
45      end_game: out std_logic;
46      result: in std_logic;
47      en_result: in std_logic;
48      endtimer: out std_logic;
49      led: out std_logic_vector(2 downto 0);
50      fine: out std_logic
51    );
52  end visual;
53
54  architecture structural of visual is
55
56  component codec is
57    Port ( clock : in STD_LOGIC; --CLOCK DIVISORE 0.5 S
58        reset : in STD_LOGIC;
59        input : in STD_LOGIC_VECTOR (7 downto 0);
60        data : out STD_LOGIC_VECTOR (4 downto 0);
61        endSeq : out STD_LOGIC;
62        enable: in STD_LOGIC);
63  end component;
64
65  component counter is
66    Port ( clock: in std_logic;
67        enable : in std_logic;
68        reset: in std_logic;
69        topscore: out std_logic;
70        output: out std_logic_vector(4 downto 0));
71  end component;
72
73  component ffd is
74    Port ( clock : in STD_LOGIC;
75        reset : in STD_LOGIC;
76        input : in STD_LOGIC_VECTOR (7 downto 0);
77        data : out STD_LOGIC_VECTOR (7 downto 0);
78        enable: in STD_LOGIC);
79  end component;
80
81  component lsfr2 is
82    Port ( clock : in STD_LOGIC;
83        reset : in STD_LOGIC;
84        lsfr_out : out STD_LOGIC_VECTOR (7 downto 0));
85  end component;
86
87  component display_seven_segments is
88    Generic(
89      CLKIN_freq : integer := 100000000;
90      CLKOUT_freq : integer := 1000
91    );
92    Port ( CLK : in STD_LOGIC;
93        RST : in STD_LOGIC;
94        VALUE : in STD_LOGIC_VECTOR (39 downto 0);
95        ENABLE : in STD_LOGIC_VECTOR (7 downto 0); --
96        DOTS : in STD_LOGIC_VECTOR (7 downto 0); --
97        ANODES : out STD_LOGIC_VECTOR (7 downto 0);
98        CATHODES : out STD_LOGIC_VECTOR (7 downto 0));
99  end component;
100
101
102  component level_velocity is
103    Port (
104      clock: in std_logic;
105      reset: in std_logic;
106      score: in std_logic_vector(4 downto 0);
107      clock_div: out std_logic
108    );
109  end component;
110
111  component display_led_controller is
112    Port (
113      clock: in std_logic;
114      arrow: in std_logic_vector(4 downto 0);
115      score: in std_logic_vector(4 downto 0);
116      result_game: in std_logic;
117      enable_result: in std_logic;
118      enable_digit: out std_logic_vector(7 downto 0);
119      led: out std_logic_vector(2 downto 0);
120      output: out std_logic_vector(39 downto 0)
121    );
122  end component;
123
124  component timer is
125    generic(
126      M: integer := 1
127    );
128  end component;
129
130  signal random_temp, data_temp: std_logic_vector(7 downto 0);
131  signal arrow_temp : std_logic_vector(4 downto 0);
132  signal clock_lento: std_logic;
133  signal counter_out_temp: std_logic_vector(4 downto 0);
134  signal value_display_temp: std_logic_vector(39 downto 0);
135  signal enable_digit_temp: std_logic_vector(7 downto 0);
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189

```

Figura 12.16: VHDL Visual

```

147  random_data: lsfr2
148    port map(clock=>clock,
149              reset=>reset,
150              lsfr_out=>random_temp);
151  flipflopD: ffd
152    port map(
153      clock=>clock,
154      reset=>reset,
155      input=>random_temp,
156      data=>data_temp,
157      enable=>enFFD);
158
159
160  kodek: codec
161    port map ( clock => clock_lento,
162                reset => rstSH,
163                input => data_temp,
164                data => arrow_temp,
165                endSeq => fine,
166                enable => enSH);
167
168
169  visor: display_seven_segments
170    Generic map(
171      CLKIN_freq => 100000000,
172      CLKOUT_freq => 1000
173    );
174    Port map ( CLK => clock,
175                RST => reset,
176                VALUE => value_display_temp,
177                ENABLE => enable_digit_temp,
178                DOTS => "00000000",
179                ANODES => anodes_out,
180                CATHODES => cathodes_out
181    );
182
183  score: counter
184    Port map ( clock => clock,
185                enable => enScore,
186                reset => reset,
187                topscore => end_game,
188                output => counter_out_temp
189  );

```

Figura 12.15: Collegamenti VHDL Visual

Proseguiamo ora con la descrizione del codice dei componenti dell'unità Player.

## DEBAUNCER BLOCK

Il debauncer block è stato realizzato strutturalmente, esso non è altro che un insieme di debauncer per i 4 pulsanti utilizzati dal giocatore. I segnali in ingresso corrispondenti ai bottoni e la loro rispettiva uscita “pulita” sono stati dichiarati come std\_logic\_vector per permetterci di utilizzare un for generate per i collegamenti.

```

34  entity button_interface is
35    Port (
36      clock: in std_logic;
37      btn_up: in std_logic;
38      btn_dx: in std_logic;
39      btn_sx: in std_logic;
40      btn_dw: in std_logic;
41      out_btn: out std_logic_vector(1 downto 0)
42    );
43  end button_interface;
44
45  architecture Behavioral of button_interface is
46
47  signal last_btn_up: std_logic;
48  signal last_btn_dw: std_logic;
49  signal last_btn_sx: std_logic;
50  signal last_btn_dx: std_logic;
51
52  begin
53  process(clock)
54  begin
55    if rising_edge(clock) then
56      if(btn_up = '1' and last_btn_up = '0') then
57        last_btn_up <= '1';
58        out_btn <= "11";
59      else last_btn_up <= '0';
60      if(btn_dw = '1' and last_btn_dw = '0') then
61        last_btn_dw <= '1';
62        out_btn <= "10";
63      else last_btn_dw <= '0';
64      if(btn_dx = '1' and last_btn_dx = '0') then
65        last_btn_dx <= '1';
66        out_btn <= "01";
67      else last_btn_dx <= '0';
68      if(btn_sx = '1' and last_btn_sx = '0') then
69        last_btn_sx <= '1';
70        out_btn <= "00";
71      else last_btn_sx <= '0';
72      end if;
73      end if;
74    end if;
75  end if;
76  end process;
77 end Behavioral;
78

```

Figura 12. 17: VHDL Debauncer Block

## BUTTON INTERFACE

Il blocco che si occupa della codifica dei pulsanti premuti dal giocatore è il button interface. Quest’ultimo prende in ingresso il segnale corrispondente ad un pulsante e ne effettua la codifica in base alla direzione che il pulsante rappresenta. È stato realizzato mediante approccio comportamentale e mediante l’utilizzo di una serie di costrutti condizionali ovvero l’if then else. Il codice è riportato in *figura 12.18*.

## COMPARETOR

Il componente comparetor, realizzato mediante ASF, si occupa di confrontare la sequenza visualizzata con la sequenza di pulsanti premuta. Tale controllo viene effettuato ad ogni pressione, di conseguenza, se il risultato viene valutato anche prima che il giocatore termini di premere tutti i pulsanti. Possiede un contatore interno per tenere traccia del numero di bit confrontati, confrontando due bit alla volta il contatore viene incrementato di 2 ad ogni colpo di clock. È presente anche un segnale asincrono di timeout inviato dal timer, se il tempo scade il segnale si alza e il comparetor pone il risultato a 0.

```

34  entity button_interface is
35    Port (
36      clock: in std_logic;
37      btn_up: in std_logic;
38      btn_dx: in std_logic;
39      btn_sx: in std_logic;
40      btn_dw: in std_logic;
41      out_btn: out std_logic_vector(1 downto 0)
42    );
43  end button_interface;
44
45  architecture Behavioral of button_interface is
46
47  signal last_btn_up: std_logic;
48  signal last_btn_dw: std_logic;
49  signal last_btn_sx: std_logic;
50  signal last_btn_dx: std_logic;
51
52  begin
53    process(clock)
54    begin
55      if rising_edge(clock) then
56        if(btn_up = '1' and last_btn_up = '0') then
57          last_btn_up <= '1';
58          out_btn <= "11";
59        else last_btn_up <= '0';
60        if(btn_dw = '1' and last_btn_dw = '0') then
61          last_btn_dw <= '1';
62          out_btn <= "10";
63        else last_btn_dw <= '0';
64        if(btn_dx = '1' and last_btn_dx = '0') then
65          last_btn_dx <= '1';
66          out_btn <= "01";
67        else last_btn_dx <= '0';
68        if(btn_sx = '1' and last_btn_sx = '0') then
69          last_btn_sx <= '1';
70          out_btn <= "00";
71        else last_btn_sx <= '0';
72        end if;
73      end if;
74    end if;
75  end process;
76 end Behavioral.
77

```

Figura 12. 18: VHDL button interface

L'automa è composto complessivamente da 3 stati (figura 12.19):

- Nello stato **q0** il sistema attende l'arrivo dell'abilitazione, che ricordiamo è calcolata come la OR delle AND tra l'abilitazione data dall'UC e la pressione dei pulsanti in modo tale che il comparatore possa attivarsi solo quando è necessario.
- Nello stato **q1** il componente effettua il confronto tra i due bit che rappresentano la pressione di uno specifico pulsante e due bit della sequenza. Se corrispondono controlla il numero di bit che sono stati confrontati (notiamo che viene verificato che count sia 6 e non 8, in quanto essendo incrementato dopo il confronto il suo valore è 6 e non 8) se è pari a 6 allora alza next\_level e result\_game, ovvero termina la partita e invia un esito positivo. Se al contrario non fossero stati confrontati ancora tutti i bit incrementa di 2 il contatore. Se invece i bit confrontati non corrispondono alza next\_level e da esito negativo ponendo a 0 result\_game.
- Nello stato **q2** abbassa next\_level e torna ad attendere l'abilitazione in q0

## PLAYER

L'unità Player è stata realizzata strutturalmente e sono stati quindi realizzati i collegamenti mostrati precedentemente nello schema.

## UNITA' OPERATIVA

L'unità operativa è composta dal componente Visual e Player, questo ci ha permesso di separare le funzionalità dell'unità così da facilitare il testing e la modifica.

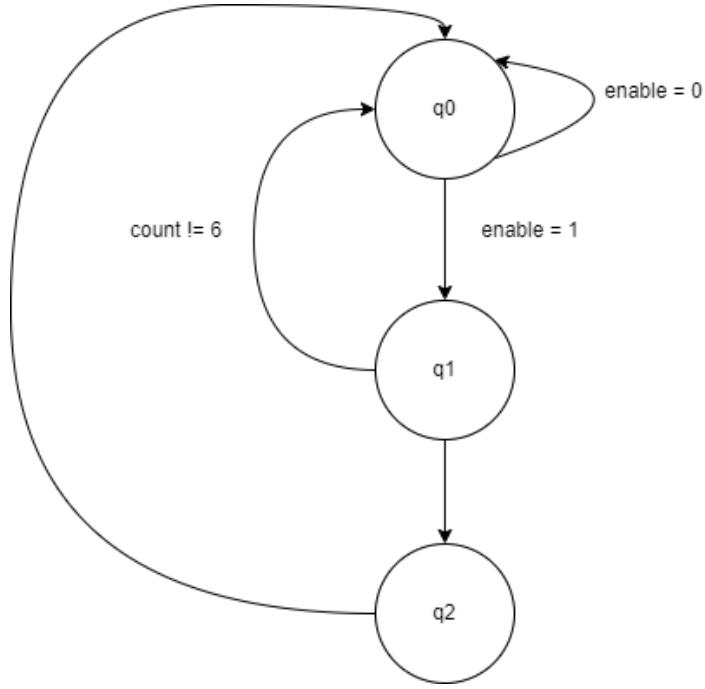


Figura 12. 19: ASF comparador

```

34 entity comparator is
35 :   Port (
36 :     clock: in std_logic;
37 :     reset: in std_logic;
38 :     enable: in std_logic;
39 :     data_ff: in std_logic_vector(7 downto 0);
40 :     data_btn: in std_logic_vector(1 downto 0);
41 :     timeout: in std_logic;
42 :     result_game: out std_logic := '0'; --pass o fail
43 :     next_level: out std_logic -- finito livello
44 :   );
45 end comparator;
46 .
47 architecture Behavioral of comparator is
48 .
49 .
50 :   type stato is (q0,q1,q2);
51 :   signal stato_corrente : stato := q0;
52 .
53 :   begin
54 :     process(clock,reset,timeout)
55 :     variable count: integer := 0;
56 :     begin
57 :       if(timeout = '1') then
58 :         result_game <= '0';
59 :       end if;
60 :       if(reset = '1') then
61 :         stato_corrente <= q0;
62 :       end if;
63 :       if rising_edge(clock) then
64 :         case stato_corrente is
65 :
66 :           when q0 =>
67 :             if(enable = '1') then
68 :               stato_corrente <= q1;
69 :             else stato_corrente <= q0;
70 :             end if;
71 :           when q1 =>
72 :             if(data_ff(count) = data_btn(0) and data_ff(count +1) = data_btn(1)) then
73 :               if(count = 6) then
74 :                 count := 0;
75 :                 result_game <= '1';
76 :                 next_level <= '1';
77 :                 stato_corrente <= q2;
78 :               else count := count +2;
79 :                 stato_corrente <= q0;
80 :               end if;
81 :             else count := 0;
82 :               result_game <= '0';
83 :               next_level <= '1';
84 :               stato_corrente <= q2;
85 :             end if;
86 :           when q2 =>
87 :             next_level <= '0';
88 :             stato_corrente <= q0;
89 :           when others => report "failure" severity failure;
90 :             end case;
91 :           end if;
92 :         end process;
93 :       end Behavioral;

```

Figura 12. 20: VHDL comparador

```

34 entity Player is
35   Port (
36     clock: in std_logic;
37     reset: in std_logic;
38     enP: in std_logic;
39     btn1: in std_logic; --up
40     btn2: in std_logic; --down
41     btn3: in std_logic; --right
42     btn4: in std_logic; --left
43     data_fp: in std_logic_vector(7 downto 0);
44     nextLevel: out std_logic;
45     resultGame: out std_logic
46   );
47 end Player;
48
49 architecture Structural of Player is
50
51 component button_interface is
52   Port (
53     clock: in std_logic;
54     btn_up: in std_logic;
55     btn_dx: in std_logic;
56     btn_sx: in std_logic;
57     btn_dw: in std_logic;
58     out_btn: out std_logic_vector(1 downto 0)
59   );
60 end component;
61
62 component comparator is
63   Port (
64     clock: in std_logic;
65     reset: in std_logic;
66     enable: in std_logic;
67     data_ff: in std_logic_vector(7 downto 0);
68     data_btn: in std_logic_vector(1 downto 0);
69     timeout: in std_logic;
70     result_game: out std_logic; --pass o fail
71     next_level: out std_logic -- finito livello
72   );
73 end component;
74
75 component timer is
76   generic(
77     M: integer := 7
78   );
79   Port (
80     clock: in std_logic;
81     reset: in std_logic;
82     end_time: out std_logic;
83     enable: in std_logic
84   );
85 end component;
86
87 component mux_21 is
88   port(X0: in std_logic;
89         X1: in std_logic;
90         s: in std_logic;
91         Y: out std_logic
92       );
93 end component;
94
95 component Debauncer_block is
96   Port (
97     reset: in std_logic;
98     clock: in std_logic;
99     btn: in std_logic_vector(3 downto 0);
100    btn_out: out std_logic_vector(3 downto 0)
101  );
102 end component;
103
104 signal out_btn_temp: std_logic_vector(1 downto 0);
105 signal timeout_temp: std_logic := '0';
106 signal result_game_temp: std_logic;
107 signal btn_out_temp: std_logic_vector(3 downto 0);
108 signal enable_comp: std_logic;
109 signal next_level_temp: std_logic;
110
111

```

Figura 12. 21: VHDL Player

```

113 begin
114
115 DB: Debauncer_block
116   port map (
117     reset => reset,
118     clock => clock,
119     btn(0) => btn1,
120     btn(1) => btn2,
121     btn(2) => btn3,
122     btn(3) => btn4,
123     btn_out => btn_out_temp
124 );
125
126
127 BI: button_interface
128   port map (
129     clock => clock,
130     btn_up => btn_out_temp(0),
131     btn_dx => btn_out_temp(1), --ci vorrebbe 2 ms mettiamo 1 per l'inversione dx dw
132     btn_sx => btn_out_temp(3),
133     btn_dw => btn_out_temp(2), --ci vorrebbe 1 ms mettiamo 2 per l'inversione dx dw
134     out_btn => out_btn_temp
135 );
136
137 enable_comp <= enP and (btn_out_temp(0) or btn_out_temp(1) or btn_out_temp(2) or btn_out_temp(3));
138
139 CMP: comparator
140   port map (
141     clock => clock,
142     reset => reset,
143     enable => enable_comp,
144     data_ff => data_fp,
145     data_btn => out_btn_temp,
146     timeout => timeout_temp,
147     result_game => result_game_temp,
148     next_level => next_level_temp
149 );
150
151 CMP: comparator
152   port map (
153     clock => clock,
154     reset => reset,
155     enable => enable_comp,
156     data_ff => data_fp,
157     data_btn => out_btn_temp,
158     timeout => timeout_temp,
159     result_game => result_game_temp,
160     next_level => next_level_temp
161 );
162
163 MUX: mux_21
164   port map (
165     X0 => result_game_temp,
166     X1 => '0',
167     s => timeout_temp,
168     Y => resultGame
169 );
170
171 MUX2: mux_21
172   port map (
173     X0 => next_level_temp,
174     X1 => '1',
175     s => timeout_temp,
176     Y => nextLevel
177 );
178
179 TM: timer
180   port map (
181     clock => clock,
182     reset => reset,
183     end_time => timeout_temp,
184     enable => enP
185 );
186
187
188 end Structural;
189

```

*Figura 12. 22: Collegamenti VHDL Player*

```

34  entity UnitaOperativa is
35      Port (
36          clock: in std_logic;
37          reset: in std_logic;
38          end_game: out std_logic;
39          en_Player: in std_logic;
40          end_visual: out std_logic;
41          en_shift: in std_logic;
42          rstSH: in std_logic;
43          enFFD: in std_logic;
44          enScore: in std_logic;
45          anodes_out: out std_logic_vector(7 downto 0);
46          cathodes_out: out std_logic_vector(7 downto 0);
47          btn1: in std_logic; --up
48          btn2: in std_logic; --down
49          btn3: in std_logic; --right
50          btn4: in std_logic; --left
51          nextLevel: out std_logic;
52          en_result: in std_logic;
53          end_timer: out std_logic;
54          led: out std_logic_vector(2 downto 0);
55          resultGame: out std_logic
56      );
57  end UnitaOperativa;
58
59  architecture Structural of UnitaOperativa is
60
61      component visual is
62          Port (
63              clock: in std_logic;
64              reset: in std_logic;
65              enSH: in std_logic;
66              enFFD: in std_logic;
67              rstSH: in std_logic;
68              enScore: in std_logic;
69              data: out std_logic_vector(7 downto 0);
70              anodes_out: out std_logic_vector(7 downto 0);
71              cathodes_out: out std_logic_vector(7 downto 0);
72              btn1: in std_logic; --up
73              btn2: in std_logic; --down
74              btn3: in std_logic; --right
75              btn4: in std_logic; --left
76              led: out std_logic_vector(2 downto 0);
77              fine: out std_logic
78          );
79      end component;
80
81      component Player is
82          Port (
83              clock: in std_logic;
84              reset: in std_logic;
85              enP: in std_logic;
86              btn1: in std_logic; --up
87              btn2: in std_logic; --down
88              btn3: in std_logic; --right
89              btn4: in std_logic; --left
90              data_fp: in std_logic_vector(7 downto 0);
91              nextLevel: out std_logic;
92              resultGame: out std_logic
93          );
94      end component;
95
96  begin
97      signal data_fp_temp: std_logic_vector(7 downto 0);
98      signal resultGame_temp: std_logic;
99
100     begin
101         begin
102             begin
103                 begin
104                     PLY: Player
105                         port map (
106                             clock => clock,
107                             reset => reset,
108                             enP => en_Player,
109                             btn1 => btn1,
110                             btn2 => btn2,
111                             btn3 => btn3,
112                             btn4 => btn4,
113                             data_fp => data_fp_temp,
114                             nextLevel => nextLevel,
115                             resultGame => resultGame_temp
116                         );
117
118                     VS: visual
119                         port map (
120                             clock => clock,
121                             reset => reset,
122                             enSH => en_Shift,
123                             enFFD => enFFD,
124                             rstSH => rstSH,
125                             enScore => enScore,
126                             data => data_fp_temp,
127                             anodes_out => anodes_out,
128                             cathodes_out => cathodes_out,
129                             end_game => end_game,
130                             result => resultGame_temp,
131                             en_result => en_result,
132                             endtimer => end_timer,
133                             led => led,
134                             fine => end_visual
135                         );
136
137
138         resultGame <= resultGame_temp;
139
140     end Structural;
141

```

Figura 12. 23: VHDL Unità Operativa

## UNITA' CONTROLLO

L’unità di controllo è stata realizzata mediante automa, descritto ampiamente nel paragrafo precedente. Per prima cosa è stata dichiarata l’entity e i segnali di input e output.

```

34  entity UnitaDiControllo is
35      Port (
36          clock: in std_logic;
37          reset: in std_logic;
38          reset_out: out std_logic;
39          end_game: in std_logic;
40          en_Player: out std_logic;
41          end_visual: in std_logic;
42          en_shift: out std_logic;
43          rstSH: out std_logic;
44          enFFD: out std_logic;
45          enScore: out std_logic;
46          nextLevel: in std_logic;
47          start: in std_logic;
48          en_result: out std_logic;
49          end_timer: in std_logic;
50          resultGame: in std_logic
51      );
52  end UnitaDiControllo;
53
54  architecture Behavioral of UnitaDiControllo is
55
56  type stato is (idle,startGame,visualize, match, evaluatetMatch, endMatch,viewResult);
57  signal stato_corrente : stato := idle;
58

```

Figura 12. 24: VHDL Unità di Controllo

Successivamente si è descritto l’automa in VHDL, non ci sono ulteriori osservazioni da fare oltre a quelle già fatte nel paragrafo “Schematici”, di conseguenza ci limitiamo a riportare il codice.

```

59 begin
60 process(clock, reset)
61 begin
62 if(reset = '1') then
63     stato_corrente <= idle;
64 end if;
65 if rising_edge(clock) then
66 case stato_corrente is
67 when idle =>
68     reset_out <= '1';
69     en_Player <= '0';
70     en_shift <= '0';
71     rstSH <= '0';
72     enFD <= '0';
73     enScore <= '0';
74     en_result <= '0';
75 if(start = '1') then
76     stato_corrente <= startGame;
77 else
78     stato_corrente <= idle;
79 end if;
80 when startGame =>
81     reset_out <= '0';
82     en_Player <= '0';
83     en_shift <= '0';
84     rstSH <= '0';
85     enFD <= '1';
86     enScore <= '0';
87     en_result <= '0';
88     stato_corrente <= visualize;
89 when visualize =>
90     reset_out <= '0';
91     en_Player <= '0';
92     en_shift <= '1';
93     enFD <= '0';
94     enScore <= '0';
95     en_result <= '0';
96 if(end_visual = '1') then
97     en_shift <= '0';
98     rstSH <= '1';
99     stato_corrente <= match;
100 else
101     stato_corrente <= visualize;
102 end if;
103 end if;
104 when match =>
105     reset_out <= '0';
106     en_Player <= '1';
107     en_shift <= '0';
108     rstSH <= '1';
109     enFD <= '0';
110     enScore <= '0';
111     en_result <= '1';
112     stato_corrente <= viewResult;
113 else
114     stato_corrente <= match;
115 end if;
116 when viewResult =>
117     reset_out <= '0';
118     en_Player <= '0';
119     en_shift <= '0';
120     rstSH <= '1';
121     enFD <= '0';
122     enScore <= '0';
123     if(end_timer = '1') then
124         en_result <= '0';
125         stato_corrente <= evaluatMatch;
126     else
127         stato_corrente <= viewResult;
128     end if;
129 when evaluatMatch =>
130     reset_out <= '0';
131     en_Player <= '0';
132     en_shift <= '0';
133     rstSH <= '1';
134     enFD <= '0';
135     en_result <= '0';
136     if(resultGame = '0') then
137         enScore <= '0';
138         stato_corrente <= idle;
139     else
140         enScore <= '1';
141         stato_corrente <= endMatch;
142     end if;
143 when endMatch =>
144     reset_out <= '0';
145     en_Player <= '0';
146     en_shift <= '0';
147     rstSH <= '0';
148     enFD <= '0';
149     enScore <= '0';
150     en_result <= '0';
151     if(end_game = '1') then
152         reset_out <= '1';
153         stato_corrente <= startGame;
154     else
155         stato_corrente <= startGame;
156     end if;
157 end if;
158 end process;
159 end Behavioral;

```

Figura 12. 25: VHDL Unità di Controllo

## GUESS THE SEQUENCE

La struttura complessiva della macchina è quindi composta da Unità Operativa, Unità di Controllo e un Button Debouncer per il segnale di start. E' riportato i seguito il codice VHDL del sistema.

```

34 entity GuessTheSequence is
35 Port (
36     clock_in: std_logic;
37     reset_in: std_logic;
38     anodes_out: out std_logic_vector(7 downto 0);
39     cathodes_out: out std_logic_vector(7 downto 0);
40     btn1: in std_logic; --up
41     btn2: in std_logic; --down
42     btn3: in std_logic; --right
43     btn4: in std_logic; --left
44     led_outb: out std_logic;
45     led_outc: out std_logic;
46     led_outr: out std_logic;
47     btn_start: in std_logic
48 );
49 end GuessTheSequence;
50
51 architecture Structural of GuessTheSequence is
52
53 component UnitaDiControllo is
54 Port (
55     clock: in std_logic;
56     reset: in std_logic;
57     reset_out: out std_logic;
58     end_game: in std_logic;
59     en_Player: out std_logic;
60     end_visual: in std_logic;
61     en_shift: out std_logic;
62     rstSH: out std_logic;
63     enFD: out std_logic;
64     enScore: out std_logic;
65     nextLevel: in std_logic;
66     start: in std_logic;
67     en_result: out std_logic;
68     end_timer: in std_logic;
69     resultGame: in std_logic
70 );
71 end component;
72
73 component UnitaOperativa is
74 Port (
75     clock: in std_logic;
76     reset: in std_logic;
77     end_game: out std_logic;
78     en_Player: in std_logic;
79     end_visual: out std_logic;
80     en_shift: in std_logic;
81     rstSH: in std_logic;
82     enFD: in std_logic;
83     enScore: in std_logic;
84     anodes_out: out std_logic_vector(7 downto 0);
85     cathodes_out: out std_logic_vector(7 downto 0);
86     btn1: in std_logic; --up
87     btn2: in std_logic; --down
88     btn3: in std_logic; --right
89     btn4: in std_logic; --left
90     nextLevel: out std_logic;
91     en_result: in std_logic;
92     end_timer: out std_logic;
93     led: out std_logic_vector(2 downto 0);
94     resultGame: out std_logic
95 );
96 end component;
97
98 component ButtonDebouncer is
99 generic (
100     CLK_period: integer := 10; -- periodo del cl
101     btn_noise_time: integer := 650000000 --interv
102     --assumo che di
103 );
104 Port ( RST : in STD_LOGIC;
105         CLK : in STD_LOGIC;
106         BTN : in STD_LOGIC;
107         CLEARED_BTN : out STD_LOGIC);
108 end component;
109

```

Figura 12. 26: VHDL Guess The Sequence

```

111 signal reset_out_temp: std_logic;
112 signal end_game_temp: std_logic;
113 signal en_Player_temp: std_logic;
114 signal end_visual_temp: std_logic;
115 signal en_shift_temp: std_logic;
116 signal rstSH_temp : std_logic;
117 signal enFFD_temp : std_logic;
118 signal enScore_temp : std_logic;
119 signal nextLevel_temp : std_logic;
120 signal start_read_temp : std_logic;
121 signal resultGame_temp : std_logic;
122 signal en_result_temp: std_logic;
123 signal end_timer_temp: std_logic;
124
125 signal reset_n : std_logic;
126
127 begin
128
129 reset_n <= not reset_in;
130
131 @ Controllo: UnitaDiControllo
132 port map (
133 clock => clock_in,
134 reset => reset_n,
135 reset_out => reset_out_temp,
136 end_game => end_game_temp,
137 en_Player => en_Player_temp,
138 end_visual => end_visual_temp,
139 en_shift => en_shift_temp,
140 rstSH => rstSH_temp,
141 enFFD => enFFD_temp,
142 enScore => enScore_temp,
143 nextLevel => nextLevel_temp,
144 start => start_read_temp,
145 en_result => en_result_temp,
146 end_timer => end_timer_temp,
147 resultGame => resultGame_temp
148 );
149
150 @ Operativa: UnitaOperativa
151 port map (
152 clock => clock_in,
153 reset => reset_out_temp,
154 end_game => end_game_temp,
155 en_Player => en_Player_temp,
156 end_visual => end_visual_temp,
157 en_shift => en_shift_temp,
158 rstSH => rstSH_temp,
159 enFFD => enFFD_temp,
160 enScore => enScore_temp,
161 anodes_out => anodes_out,
162 cathodes_out => cathodes_out,
163 btn1 => btn1,
164 btn2 => btn2,
165 btn3 => btn3,
166 btn4 => btn4,
167 nextLevel => nextLevel_temp,
168 en_result => en_result_temp,
169 end_timer => end_timer_temp,
170 led(2) => led_outb,
171 led(0) => led_outg,
172 led(1) => led_outr,
173 resultGame => resultGame_temp
174 );
175
176 @ BTND: ButtonDebouncer
177 port map (
178 RST => reset_n,
179 CLK => clock_in,
180 BTN => btn_start,
181 CLEARED_BTN => start_read_temp
182 );
183
184 end Structural;
185

```

Figura 12. 27: VHDL Guess The Sequence