



UNIVERSITÀ^{DEGLI} STUDI DI
NAPOLI FEDERICO II

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

Progetto di Architettura dei sistemi digitali

Teresa Di Dona (M63001437)

Preziosa Pietroluongo (M63001456)

INDICE

CAPITOLO 1: RETE DI INTERCONNESSIONE	6
1.1: COS' È IL MULTIPLEXER	6
1.2 SOLUZIONE	6
1.2.1 MUX 2:1.....	8
1.2.2 MUX 4:1.....	9
1.2.3 MUX 16:1.....	10
1.2.4 SIMULAZIONE.....	11
1.3 RETE DI INTERCONNESSIONE E DEMULTIPLEXER.....	12
1.3.1 DEMUX 1:2	13
1.3.2 DEMUX 1:4	14
1.3.3 RETE DI INTERCONNESSIONE.....	15
1.4 IMPLEMENTAZIONE SU BOARD	16
CAPITOLO 2: ENCODER BCD	19
2.1 DESCRIZIONE	19
2.2 CODICE VHDL.....	19
2.2.1 TESTBENCH.....	20
2.3 SECONDO PUNTO.....	21
2.4 DISPLAY.....	22
2.4.1: ENCODER_DISPLAY	23
CAPITOLO 3: RICONOSCITORE DI SEQUENZE	24
3.1 SOLUZIONE	24
3.1.1 SIMULAZIONE.....	27
3.2 SOLUZIONE	29
CAPITOLO 4: SHIFT REGISTER	32
4.1: COS'È LO SHIFT REGISTER	32
4.2: APPROCCIO COMPORTAMENTALE	32
4.3: APPROCCIO STRUTTURALE.....	34
4.3.1: FLIP-FLOP D	35
4.3.2: REALIZZAZIONE	36
CAPITOLO 5: CRONOMETRO.....	39
5.1: SOLUZIONE	39
5.1.1 Implementazione VHDL.....	41
5.1.1.1 Implementazione contatore con ff	41

5.1.1.2 Implementazione cronometro	43
5.1.2: SIMULAZIONE	45
5.2: SOLUZIONE	48
5.2.1 Implementazione VHDL	48
5.3: SOLUZIONE	50
5.3.1 Implementazione VHDL	51
5.3.1.1 Shift register	51
5.3.1.2 Banco di registri	51
5.3.1.3 Controllo	52
5.3.1.3.1: Osservazione	54
5.3.1.4: Implementazione c_on_board	54
CAPITOLO 6: SISTEMA DI TESTING	58
6.1: ROM, CONTATORE E MACCHINA COMBINATORIA	58
6.1.1: MEMORIA	60
6.1.2: RETE DI CONTROLLO	65
6.1.3: ARCHITETTURA	67
6.1.4: SIMULAZIONE	70
6.2: IMPLEMENTAZIONE SU BOARD	71
CAPITOLO 7: COMUNICAZIONE CON HANDSHAKING	73
7.1: SOLUZIONE	73
7.1.1: A	73
7.1.2: B	76
7.2: IMPLEMENTAZIONE VHDL	77
7.2.1: A	78
7.2.1.1: Contatore	78
7.2.1.2: Memoria	78
7.2.1.3: RC_A	79
7.2.1.4: NODO A	82
7.2.2: B	84
7.2.2.1: Registro R_IN	84
7.2.2.2: Memoria	84
7.2.2.3: Contatore	85
7.2.2.4: Addizionatore	86
7.2.2.5: RC_B	86
7.2.3: AB	93
7.3: SIMULAZIONE	95

CAPITOLO 8: PROCESSORE	97
8.1: PROCESSORE MIC-1	97
8.1.1: Parte operativa.....	97
8.1.2: UNITA' DI CONTROLLO	100
8.2: ANALISI ISTRUZIONI.....	101
8.2.1: BIPUSH	101
8.2.2: IF_ICMPEQ.....	102
8.3: MODIFICA DEL CODICE OPERATIVO	105
CAPITOLO 9: INTERFACCIA SERIALE	106
9.1: PERIFERICA UART.....	106
9.1.1: TRASMETTITORE	107
9.1.2: RICEVITORE	108
9.2: UART IN CONFIGURAZIONE TAPPO	109
9.3: 2_UART_MEM	111
CAPITOLO 10: SWITCH MULTISTADIO	124
10.1: COS'È LO SWITCH MULTISTADIO.....	124
10.2: SVOLGIMENTO	124
10.2.1: PARTE DI CONTROLLO.....	129
10.2.2: INTERCONNESSIONE	131
10.2.3: TESTBENCH.....	133
10.3: RETE DI 8 NODI.....	135
CAPITOLO 11: MOLTIPLICATORE DI BOOTH	140
11.1: DESCRIZIONE	140
11.2: FUNZIONAMENTO	142
11.2.1: Automa.....	143
11.3: IMPLEMENTAZIONE VHDL	144
11.3.1: SHIFT REGISTER A.....	144
11.3.2: Implementazione Q	145
11.3.3: Implementazione M	146
11.3.4: Implementazione Full Adder	146
11.3.5: Implementazione RCA	147
11.3.6: Implementazione ADD_SUB	148
11.3.7: Implementazione rete di controllo	150
11.3.8: Implementazione Moltiplicatore di Booth.....	153
11.3.9: Implementazione gestione_uscita	156

11.3.10: Implementazione tot	156
11.4: SIMULAZIONE.....	158
11.5: Scheda	159
CAPITOLO 12: ESERCIZIO LIBERO	162
12.1: SOLUZIONE.....	162
12.1.1: A	163
12.1.2: B.....	165
12.1.3: C.....	166
12.2: IMPLEMENTAZIONE VHDL.....	168
12.2.1: Nodo A	168
12.2.1.1: Contatore	168
12.2.1.2: ROM	169
12.2.1.3: MicroRom.....	170
12.2.1.4: RC_A	173
12.2.1.5: A	176
12.2.2: Nodo B.....	178
12.2.2.1: Registro	178
12.2.2.2: Comparatore	178
12.2.2.3: RC_B	179
12.2.2.4: B	181
12.2.3: Nodo C.....	183
12.2.3.1: REGISTRO, CONTATORE, COMPARATORE	183
12.2.3.2: RC_C	183
12.2.3.3: C	186
12.2.4: TOTALE.....	189
12.3: SIMULAZIONE.....	192

CAPITOLO 1: RETE DI INTERCONNESSIONE

Esercizio 1.1

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

Esercizio 1.2

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

Esercizio 1.3

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

1.1: COS' è IL MULTIPLEXER

Un multiplexer (spesso abbreviato in "MUX") è una macchina combinatoria che permette, tramite una selezione, di individuare uno tra più segnali di ingresso e instradare solo quello in uscita.

Il multiplexer prevede n ingressi (in base a quanto grande si decide di progettare), $\log(n)$ selezioni e una singola linea di uscita.

Il multiplexer più piccolo che si possa realizzare prevede 2 ingressi, una selezione e naturalmente un'uscita, come dimostra anche la figura 1.1.

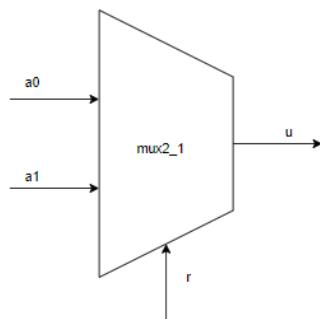


Figura 1.1: Multiplexer 2:1

Nel nostro caso, il primo punto di questo esercizio richiede di progettare un multiplexer da 16 ingressi (16:1) a partire da un multiplexer da 4 ingressi (4:1) seguendo un approccio strutturale.

1.2 SOLUZIONE

Per poter realizzare un multiplexer da 16 ingressi, siamo partite dalla realizzazione di multiplexer da 2 ingressi. Tramite una descrizione di tipo dataflow, abbiamo così progettato la cella elementare della struttura (come descritto nel paragrafo 1.2.1) e, a partire da questa, abbiamo potuto realizzare in maniera strutturale prima un multiplexer da 4 e infine da 16 ingressi. Per la realizzazione del mux 4:1 abbiamo sfruttato 3 multiplexer da 2, in particolare i 4 ingressi entrano nei due multiplexer da due, posti sul primo livello (i primi due ingressi nel primo multiplexer 2:1 e gli altri 2 ingressi nel secondo mux 2:1), come riportato dalla figura 1.2; le uscite di questi due mux rappresenteranno gli ingressi del terzo mux 2:1, posto sul secondo livello. L'uscita di questo mux risulterà essere l'uscita finale. Le due selezioni di ingresso consentono di individuare l'input corretto da visualizzare in uscita,

in particolare, la prima selezione entra in ingresso ai primi due mux sul primo livello, mentre la seconda selezione entra nel terzo mux sul secondo livello. Grazie a queste due selezioni riusciamo così a capire quale dei quattro ingressi (al mux 4:1) portare in uscita.

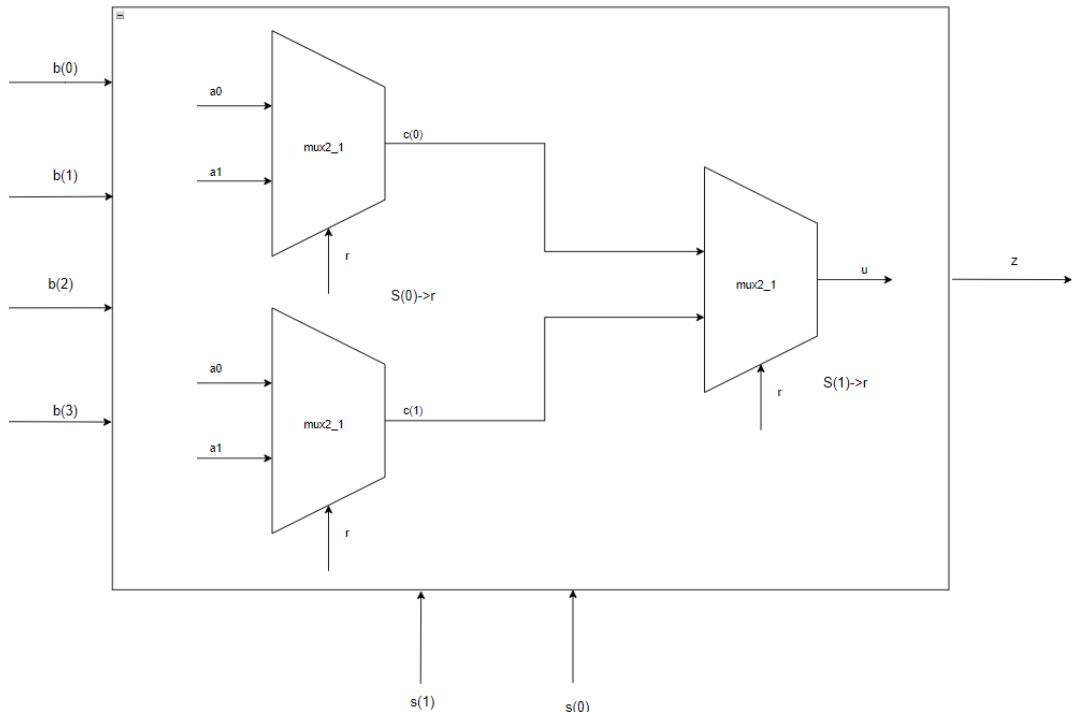


Figura 1.2: Multiplexer 4:1

In maniera del tutto analoga, abbiamo poi progettato il componente desiderato (mux 16:1). In particolare, sfruttando cinque mux da 4 ingressi, e connettendo opportunamente le uscite dei primi 4 multiplexer (come visibile anche in figura 1.3) nel quinto multiplexer sul secondo livello, otteniamo l'uscita del mux 16:1. Prendiamo, infatti, 16 valori di ingresso che dividiamo in 4 gruppi da 4, così che ogni gruppo possa essere l'ingresso di un dato mux 4:1. Avremo in questo caso $\log(16)=4$ ingressi di selezione, di cui, i primi due entranti nei 4 mux del primo livello e i restanti altri due entranti nel quinto e ultimo mux (sul secondo livello).

Dunque, otteniamo lo schema in figura 1.3.

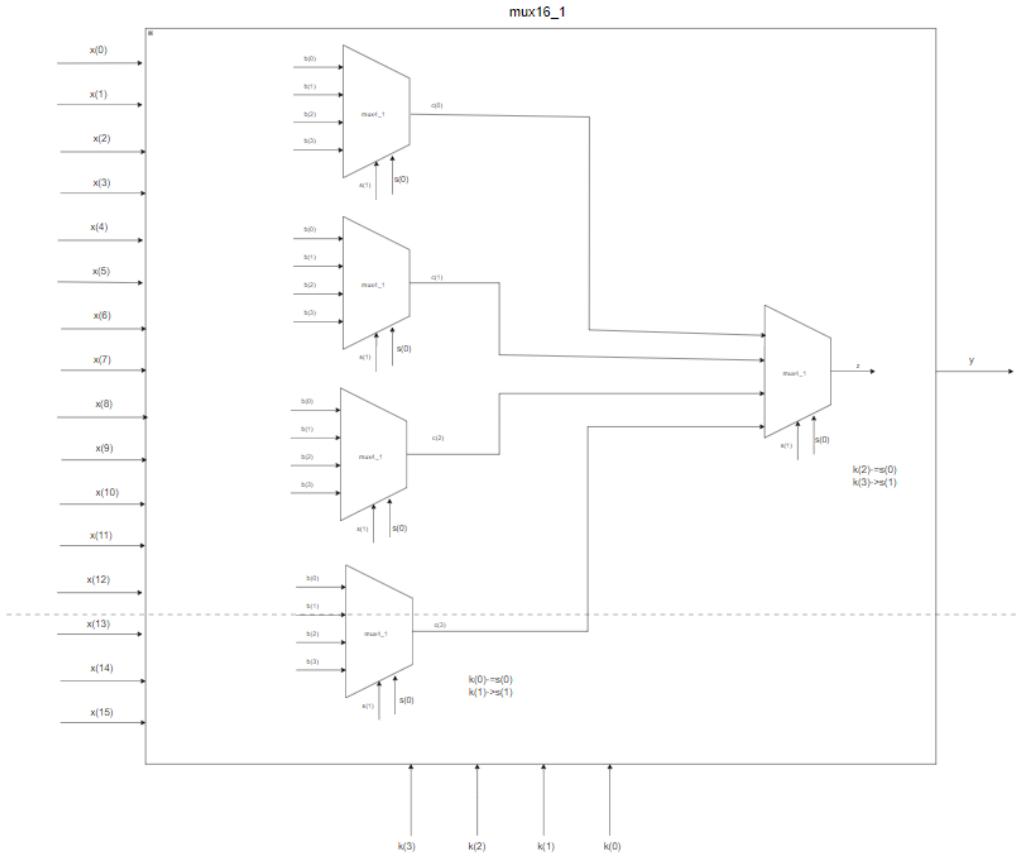


Figura 1.3: Multiplexer 16:1

1.2.1 MUX 2:1

L'implementazione in VHDL di questo componente elementare è la seguente:

```

entity mux_2_1 is
  port (
    a0 : in std_logic;
    a1 : in std_logic;
    r : in std_logic;
    u : out std_logic
  );
end mux_2_1;

architecture dataflow of mux_2_1 is
begin
  u <= ((not r) and a0) or (r and a1);
end dataflow;

```

1.2.2 MUX 4:1

L'implementazione in VHDL del mux 4:1 è la seguente:

```
entity mux_4_1 is
  port (
    b: in std_logic_vector(0 to 3);
    s: in std_logic_vector(1 downto 0);
    z: out std_logic
  );
end mux_4_1;

architecture structural of mux_4_1 is

component mux_2_1 is
  port (
    a0 : in std_logic;
    a1 : in std_logic;
    r : in std_logic;
    u : out std_logic
  );
end component;

signal c: std_logic_vector (0 to 1);

begin

  mux0_0: mux_2_1
  port map (
    a0 => b(0),
    a1 => b(1),
    r => s(0),
    u => c(0)
  );

  mux0_1: mux_2_1
  port map (
    a0 => b(2),
    a1 => b(3),
    r => s(0),
    u => c(1)
  );

  mux1: mux_2_1
  port map (
    a0 => c(0),
    a1 => c(1),
    r => s(1),
    u => z
  );

end structural;
```

1.2.3 MUX 16:1

L'implementazione in VHDL del mux 16:1 è la seguente:

```
entity mux_16_1 is
port (
    x: in std_logic_vector(0 to 15);
    k: in std_logic_vector(3 downto 0);
    y: out std_logic
);
end mux_16_1;

architecture structural of mux_16_1 is

component mux_4_1 is
port (
    b: in std_logic_vector(0 to 3);
    s: in std_logic_vector(1 downto 0);
    z: out std_logic
);
end component;

signal c: std_logic_vector(0 to 3);

begin

    mux0_0: mux_4_1
    port map (
        b(0 to 3) => x(0 to 3),
        s(1 downto 0) => k(1 downto 0),
        z => c(0)
    );

    mux0_1: mux_4_1
    port map (
        b(0 to 3) => x(4 to 7),
        s(1 downto 0) => k(1 downto 0),
        z => c(1)
    );

    mux0_2: mux_4_1
    port map (
        b(0 to 3) => x(8 to 11),
        s(1 downto 0) => k(1 downto 0),
        z => c(2)
    );

    mux0_3: mux_4_1
    port map (
        b(0 to 3) => x(12 to 15),
        s(1 downto 0) => k(1 downto 0),
        z => c(3)
    );

    mux1: mux_4_1
    port map (
        b(0 to 3) => c(0 to 3),
        s(1 downto 0) => k(3 downto 2),
        z => y
    );

end structural;
```

1.2.4 SIMULAZIONE

Per poter effettuare la simulazione sfruttiamo il testbench. Il testbench è un modulo in cui si definiscono i segnali di ingresso e uscita necessari per testare il funzionamento del circuito da simulare. Il testbench non è l'oggetto da realizzare, ma serve per verificare che il sistema funzioni correttamente. Nel codice mostrato di seguito, il corpo dell'entity è vuoto perché il testbench non è un oggetto che deve essere realizzato fisicamente, ma è utilizzato solo per effettuare la simulazione del circuito.

Il testbench, inoltre, non ha segnali di ingresso né di uscita perché non viene istanziato da nessun blocco, ma istanzia al suo interno altri blocchi (in questo caso il mux 16:1), per effettuare il test.

In particolare, per simulare il multiplexer sono stati forniti tre ingressi differenti con 3 abilitazioni differenti e uscite annesse; in particolare il primo test è stato fatto fornendo la stringa "1001101100001101" e con abilitazione "1001", dunque il risultato atteso era lo zero, come si evince dal codice sottostante.

```
○ wait for 10ns;
○ xl<="1001101100001101";
○ kl<="1001";
○ wait for 10ns;
○ assert y1='0'
○ report "errore0"
○ severity failure;

○ wait for 10ns;
○ xl<="0001101100001101";
○ kl<="0000";
○ wait for 10ns;
○ assert y1='0'
○ report "errore1"
○ severity failure;

○ wait for 10ns;
○ xl<="1111101100001101";
○ kl<="0011";
○ wait for 10ns;
○ assert y1='1'
○ report "errore2"
○ severity failure;
```

I risultati della simulazione sono visibili in figura 1.4.



Figura 1.4: Simulazione mux 16:1

1.3 RETE DI INTERCONNESSIONE E DEMULTIPLEXER

Per la realizzazione della *rete di interconnessione a 16 ingressi e 4 uscite* è stato necessario realizzare dei demultiplexer, in modo da poterli così collegare al mux 16:1 realizzato nel punto precedente.

Un demultiplexer (demux) è un circuito digitale che ha un singolo ingresso e più uscite; viene utilizzato per instradare il segnale in ingresso ad una delle possibili uscite in base al segnale di controllo; quest'ultimo stabilisce quale delle possibili uscite deve essere attivata, in modo da poter instradare il segnale nella corrispondente uscita. Svolge di fatti, l'operazione complementare del multiplexer.

Anche per la realizzazione di questo componente siamo partiti dal componente di base, con un segnale di abilitazione e due uscite, come riporta lo schema in figura 1.5.

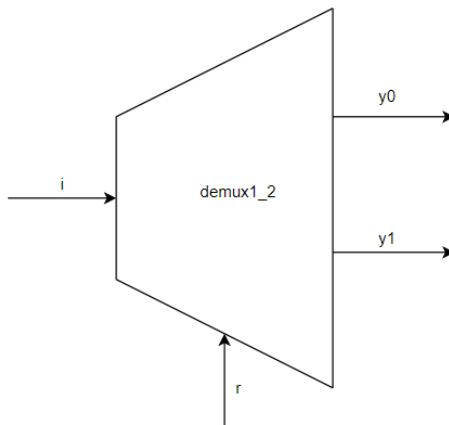


Figura 1.5: Demultiplexer 1:2

Seguendo un approccio strutturale, analogo a quello adottato per il multiplexer, è stato possibile realizzare, a partire dal demux 1:2 un demux 1:4; in particolare, sono stati utilizzati tre demux 1:2 opportunamente interconnessi (come mostra la figura 1.6), legando le uscite del primo demux sul primo livello agli ingressi dei due demux sul secondo livello. Le selezioni in ingresso al demux 1:4 sono state necessarie per abilitare i dispositivi così da poter visualizzare le opportune uscite.

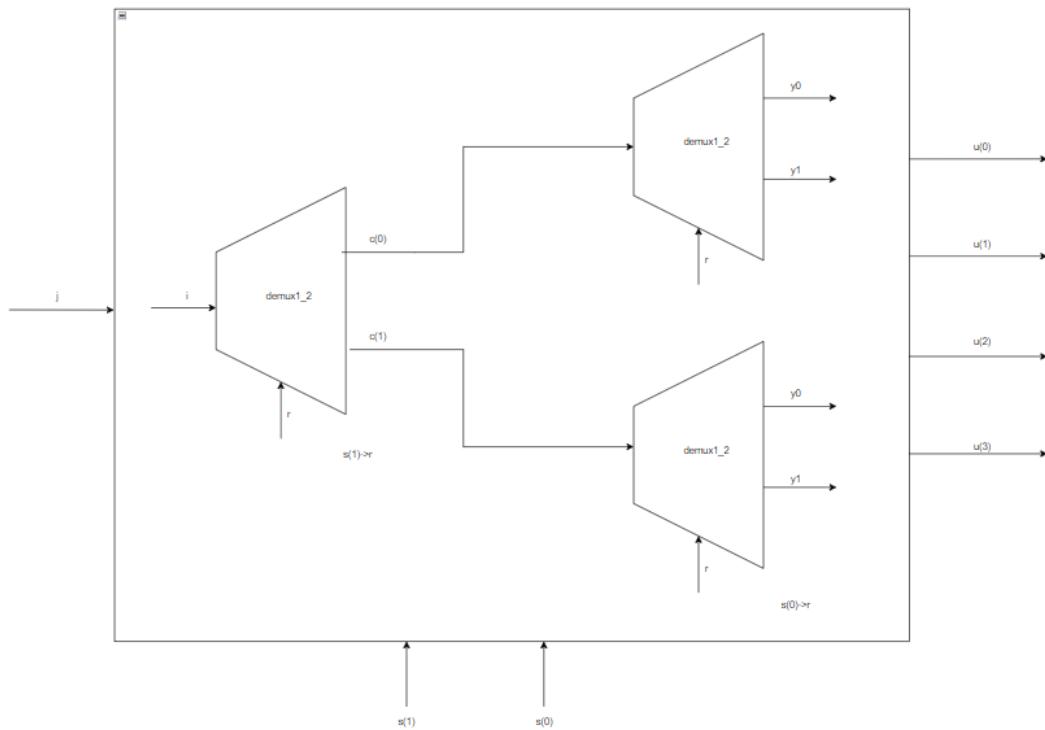


Figura 1.6: Demultiplexer 1:4

1.3.1 DEMUX 1:2

L'implementazione in VHDL di questo componente elementare è la seguente:

```

entity demux_1_2 is
  port(
    i: in std_logic;
    r: in std_logic;
    y0: out std_logic;
    y1: out std_logic
  );
end demux_1_2;

architecture dataflow of demux_1_2 is

begin

  y0 <= (not r) and i;
  y1 <= r and i;

end dataflow;

```

1.3.2 DEMUX 1:4

L'implementazione in VHDL del demux 1:4 è la seguente:

```
entity demux_1_4 is
  port(
    j: in std_logic;
    s: in std_logic_vector(1 downto 0);
    u: out std_logic_vector(0 to 3)
  );
end demux_1_4;

architecture structural of demux_1_4 is

component demux_1_2 is
  port(
    i: in std_logic;
    r: in std_logic;
    y0: out std_logic;
    y1: out std_logic
  );
end component;

signal c: std_logic_vector(0 to 1);

begin

  demux_0: demux_1_2
    port map (
      i => j,
      r => s(1),
      y0 => c(0),
      y1 => c(1)
    );

  demux_1_0: demux_1_2
    port map (
      i => c(0),
      r => s(0),
      y0 => u(0),
      y1 => u(1)
    );

  demux_1_1: demux_1_2
    port map (
      i => c(1),
      r => s(0),
      y0 => u(2),
      y1 => u(3)
    );

end structural;
```

1.3.3 RETE DI INTERCONNESSIONE

Realizzati a questo punto i componenti elementare, è stato possibile interconnetterli come nello schema seguente (figura 1.7).

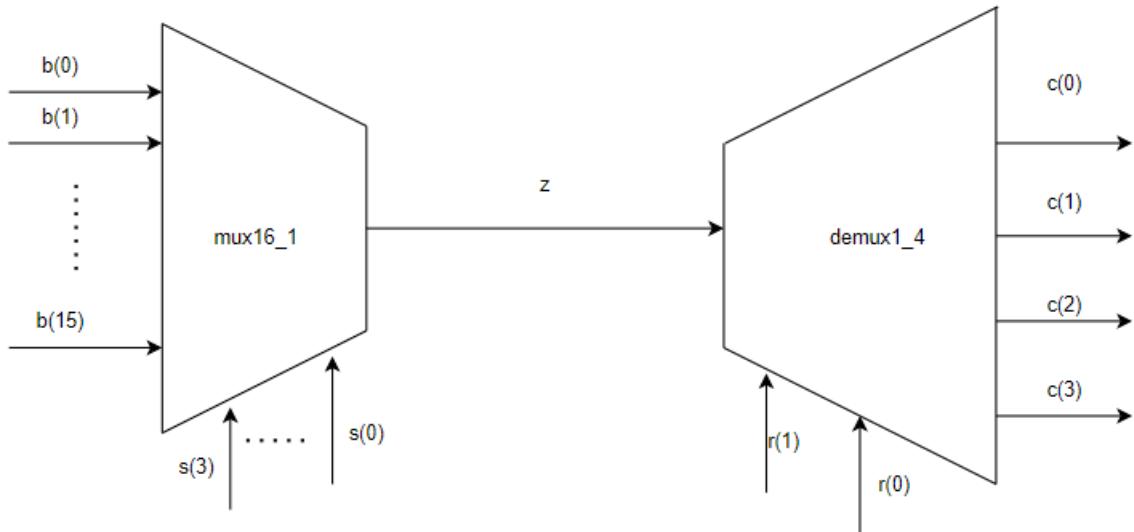


Figura 1.7: Rete di interconnessione

Come si evince anche dal codice riportato di seguito, l'implementazione in VHDL prevedeva semplicemente di collegare l'uscita del multiplexer con l'ingresso del demultiplexer, e di fornire gli opportuni segnali di selezione.

```

entity conn_16_4 is
    port (
        a: in std_logic_vector(0 to 15);
        s_mux: in std_logic_vector(3 downto 0);
        s_demux: in std_logic_vector(1 downto 0);
        y: out std_logic_vector(0 to 3)
    );
end conn_16_4;

architecture structural of conn_16_4 is

component mux_16_1 is
    port (
        x: in std_logic_vector(0 to 15);
        k: in std_logic_vector(3 downto 0);
        y: out std_logic
    );
end component;

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

signal c: std_logic;

```

```

begin

    mux: mux_16_1
    port map (
        x(0 to 15) => a(0 to 15),
        k(3 downto 0) => s_mux(3 downto 0),
        y => c
    );

    demux: demux_1_4
    port map (
        j => c,
        s(1 downto 0) => s_demux(1 downto 0),
        u(0 to 3) => y(0 to 3)
    );

end structural;

```

1.4 IMPLEMENTAZIONE SU BOARD

Progettata a questo punto la rete di interconnessione, non ci restava altro che caricarla su scheda. A tal proposito, abbiamo progettato una rete di controllo che consentisse l'acquisizione della stringa dati in ingresso sugli switch della board, oltre naturalmente ad utilizzare gli switch per gli input di selezione.

Come si può osservare nel codice riportato sotto, abbiamo utilizzato dei bottoni (opportunamente settati nel *Constraints*) per decidere dove memorizzare il valore letto dagli switch e quindi se si trattasse di un dato o di un segnale di selezione. Abbiamo poi memorizzato i valori in dei registri, rispettivamente: *reg_inputs* (per i bit di dato), *reg_selmux* (per i bit di selezione del multiplexer) e *reg_seldem* (per i bit di selezione del demultiplexer).

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity unita_di_controllo is port (
    switch : in std_logic_vector (0 to 15);
    button_0 : in std_logic; -- button [0] per i selettori button [1] per i dati
    button_1 : in std_logic;
    reset : in std_logic;
    clock : in std_logic;
    reg_inputs : out std_logic_vector (0 to 15) := (others => '0');
    reg_selmux : out std_logic_vector (3 downto 0) := "0000";
    reg_seldem : out std_logic_vector (1 downto 0) := "00"
);
end unita_di_controllo;

architecture Behavioral of unita_di_controllo is

begin
main: process (clock)
begin
    if (rising_edge(clock)) then
        if (reset = '1') then
            reg_inputs <= (others => '0');
            reg_selmux <= (others => '0');
            reg_seldem <= (others => '0');
        end if;
        if (button_0= '1') then
            reg_selmux (3) <= switch (15);
            reg_selmux (2) <= switch (14);
    end if;
end process;

```

```

        reg_selmux (1) <= switch (13);
        reg_selmux (0) <= switch (12);
        reg_seldem (1) <= switch (11);
        reg_seldem (0) <= switch (10);
    end if;
    if (button_1= '1') then
        reg_inputs <= switch (0 to 15);
    end if;
end if;

end process;

end Behavioral;

```

Per concludere l'esercizio richiesto abbiamo in fine realizzato un'unica grande scatola che abbiamo chiamato *macchina* che contenesse questi componenti: l'unità di controllo e la rete di interconnessione a 16 sorgenti e 4 destinazioni. In questo modo abbiamo potuto collegare i dati raccolti dall'esterno (tramite gli switch) e memorizzati sui registri, con gli ingressi della rete di controllo e le uscite di quest'ultima con i led presenti sulla board, in modo da visualizzare i 4 bit di uscita. L'intero procedimento è descritto in maniera dettagliata nel codice presente di seguito:

```

entity macchina is port (
    switch      : in std_logic_vector (0 to 15);
    button_0    : in std_logic;      -- button [0] per i selettori button [1] per i dati
    button_1    : in std_logic;
    reset       : in std_logic;
    clock       : in std_logic;
    led         : out std_logic_vector (0 to 3)
);
end macchina;

architecture structural of macchina is
component conn_16_4 is port (
    a: in std_logic_vector(0 to 15);
    s_mux: in std_logic_vector(3 downto 0);
    s_demux: in std_logic_vector(1 downto 0);
    y: out std_logic_vector(0 to 3)
);
end component;

component unita_di_controllo is port (
    switch      : in std_logic_vector (0 to 15);
    button_0    : in std_logic;      -- button [0] per i selettori button [1] per i dati
    button_1    : in std_logic;
    reset       : in std_logic;
    clock       : in std_logic;
    reg_inputs  : out std_logic_vector (15 downto 0) := (others => '0');
    reg_selmux  : out std_logic_vector (3 downto 0) := "0000";
    reg_seldem  : out std_logic_vector (1 downto 0) := "00"
);
end component;

```

```

signal inputs    : std_logic_vector (0 to 15);
signal selmux   : std_logic_vector (3 downto 0);
signal seldem    : std_logic_vector (1 downto 0);

begin

| cu : unita_di_controllo port map (
  switch      => switch,
  button_0    =>button_0,
  button_1    =>button_1,
  reset       =>reset,
  clock        =>clock,
  reg_inputs  => inputs,
  reg_selmux  => selmux,
  reg_seldem  => seldem
| );
| );

| po : conn_16_4 port map (
  a           => inputs,
  s_mux       => selmux,
  s_demux    => seldem,
  y           => led
| );
| end structural;

```

CAPITOLO 2: ENCODER BCD

Esercizio 2.1

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

Input: 0000000001 Output: 0000 (cifra 0)

Input: 0000000010 Output: 0001 (cifra 1)

Input: 0000000100 Output: 0010 (cifra 2)

....

Esercizio 2.2

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

Esercizio 2.3

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

2.1 DESCRIZIONE

L'encoder, o codificatore, è costituito da un numero di ingressi (i) minore o uguale di 2^n , dove n rappresenta il numero delle uscite; in particolare, l'encoder BCD non sfrutta tutte le $2^4 = 16$ possibili combinazioni, prevedendo solo 10 ingressi.

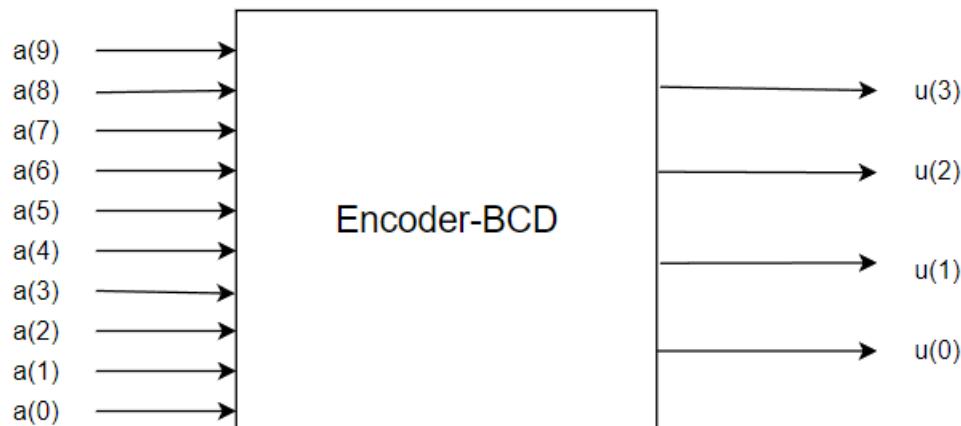


Figura 2.1: Encoder_BCD

2.2 CODICE VHDL

Per realizzare un encoder di tipo BCD abbiamo utilizzato una descrizione di tipo dataflow. Attraverso questo tipo di architettura ed usando i costrutti WHEN ELSE che il linguaggio VHDL mette a disposizione, abbiamo descritto la tabella di verità di questa macchina combinatoria.

```

entity Encoder_BCD is
  port (
    a : in std_logic_vector(9 downto 0);
    u : out std_logic_vector(3 downto 0)
  );
end Encoder_BCD;

architecture Dataflow of Encoder_BCD is

begin

  u <= "0000" when a = "0000000001" else
    "0001" when a = "0000000010" else
    "0010" when a = "0000000100" else
    "0011" when a = "0000001000" else
    "0100" when a = "0000010000" else
    "0101" when a = "0000100000" else
    "0110" when a = "0001000000" else
    "0111" when a = "0010000000" else
    "1000" when a = "0100000000" else
    "1001" when a = "1000000000" else
    "1111";

end Dataflow;

```

2.2.1 TESTBENCH

Generiamo il testbench per testare il corretto funzionamento dell'encoder.

Nell'architettura di questa entità, specifichiamo il componente Encoder BCD con i suoi ingressi e le sue uscite. Generiamo due segnali interni, inputs e outputs, inizialmente inizializzati a un valore "Undefined". Questi due segnali saranno associati agli ingressi "a" e uscite "u" dell'Encoder.

Nel process sono stati definiti diversi casi di test, specificando, dato un certo input, il risultato atteso e un errore nel caso in cui non si verifichi la corrispondenza tra output atteso e quello ottenuto.

```

entity Encoder_BCD_tb is
end Encoder_BCD_tb;

architecture Behavioral of Encoder_BCD_tb is

component Encoder_BCD is
  port(
    a : in std_logic_vector(9 downto 0);
    u : out std_logic_vector(3 downto 0)
  );
end component;

signal inputs : std_logic_vector(9 downto 0) := (others => 'U');
signal outputs : std_logic_vector(3 downto 0) := (others => 'U');

begin

uut: Encoder_BCD
  port map (
    a => inputs,
    u => outputs
  );

stim_proc: process

```

```

begin

wait for 100 ns;
-- a7
wait for 10 ns;
inputs <= "0010000000";
wait for 10 ns;
assert outputs = "0111"
report "errore caso 1"
severity failure;

-- a2
wait for 10 ns;
inputs <= "0000000100";
wait for 10 ns;
assert outputs = "0010"
report "errore caso 2"
severity failure;

-- a5
wait for 10 ns;
inputs <= "0000100000";
wait for 10 ns;
assert outputs = "0101"
report "errore caso 3"
severity failure;

wait;

end process;
end Behavioral;

```

La simulazione è visibile in figura 2.2.

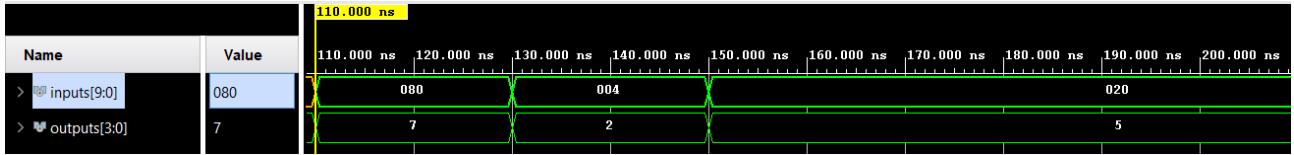


Figura 2.2: Testbench

2.3 SECONDO PUNTO

Per associare gli ingressi agli switch e per visualizzare l'uscita sui led della board abbiamo modificato il file constraints “Nexys A7-50T”. Di seguito riportiamo le modifiche apportate al file:

```

##Switches
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { a[0] }]; #IO_L24N_T3_RSO_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { a[1] }]; #IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { a[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { a[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { a[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { a[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { a[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { a[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMOS18 } [get_ports { a[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS18 } [get_ports { a[9] }]; #IO_25_34 Sch=sw[9]
#set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMOS33 } [get_ports { SW[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
#set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMOS33 } [get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
#set_property -dict { PACKAGE_PIN H6 IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
#set_property -dict { PACKAGE_PIN U12 IOSTANDARD LVCMOS33 } [get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
#set_property -dict { PACKAGE_PIN U11 IOSTANDARD LVCMOS33 } [get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
#set_property -dict { PACKAGE_PIN V10 IOSTANDARD LVCMOS33 } [get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]

```

```

## LEDs
set_property -dict { PACKAGE_PIN H17  IOSTANDARD LVCMOS33 } [get_ports { u[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15  IOSTANDARD LVCMOS33 } [get_ports { u[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13  IOSTANDARD LVCMOS33 } [get_ports { u[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14  IOSTANDARD LVCMOS33 } [get_ports { u[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
#set_property -dict { PACKAGE_PIN R18  IOSTANDARD LVCMOS33 } [get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17  IOSTANDARD LVCMOS33 } [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
#set_property -dict { PACKAGE_PIN U17  IOSTANDARD LVCMOS33 } [get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
#set_property -dict { PACKAGE_PIN U16  IOSTANDARD LVCMOS33 } [get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
#set_property -dict { PACKAGE_PIN V16  IOSTANDARD LVCMOS33 } [get_ports { LED[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
#set_property -dict { PACKAGE_PIN T15  IOSTANDARD LVCMOS33 } [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
#set_property -dict { PACKAGE_PIN U14  IOSTANDARD LVCMOS33 } [get_ports { LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
#set_property -dict { PACKAGE_PIN T16  IOSTANDARD LVCMOS33 } [get_ports { LED[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
#set_property -dict { PACKAGE_PIN V15  IOSTANDARD LVCMOS33 } [get_ports { LED[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
#set_property -dict { PACKAGE_PIN V14  IOSTANDARD LVCMOS33 } [get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
#set_property -dict { PACKAGE_PIN V12  IOSTANDARD LVCMOS33 } [get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
#set_property -dict { PACKAGE_PIN V11  IOSTANDARD LVCMOS33 } [get_ports { LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

```

2.4 DISPLAY

Per soddisfare il terzo punto dell'esercizio, dovendo rappresentare l'uscita dell'encoder sul display a 7 segmenti, abbiamo utilizzato delle costanti da zero a nove, essendo questo il range dei valori da rappresentare. Abbiamo, inoltre, utilizzato una costante in più, ovvero la "e", per i valori che non appartengono al range.

```

entity display is port (
    punto : in std_logic;                                -- bit da mettere a 0 se si vuole displayare il punto
    u    : in std_logic_vector (3 downto 0); -- numero
    catodi : out std_logic_vector (7 downto 0); -- a partire da: ca a dp
    anodi : out std_logic_vector (7 downto 0) -- a partire da: an7 a an0
);
end display;

architecture dataflow of display is

constant zero  : std_logic_vector(6 downto 0) := "1000000";
constant one   : std_logic_vector(6 downto 0) := "1111001";
constant two   : std_logic_vector(6 downto 0) := "0100100";
constant three : std_logic_vector(6 downto 0) := "0110000";
constant four  : std_logic_vector(6 downto 0) := "0011001";
constant five  : std_logic_vector(6 downto 0) := "0010010";
constant six   : std_logic_vector(6 downto 0) := "0000010";
constant seven : std_logic_vector(6 downto 0) := "1111000";
constant eight : std_logic_vector(6 downto 0) := "0000000";
constant nine  : std_logic_vector(6 downto 0) := "0010000";
--constant a    : std_logic_vector(6 downto 0) := "0001000";
--constant b    : std_logic_vector(6 downto 0) := "0000011";
--constant c    : std_logic_vector(6 downto 0) := "1000110";
--constant d    : std_logic_vector(6 downto 0) := "0100001";
constant e    : std_logic_vector(6 downto 0) := "0000110";
--constant f    : std_logic_vector(6 downto 0) := "0001110";

signal cifra_anodo : std_logic_vector (7 downto 0) := "01111111";
signal numero  : std_logic_vector (6 downto 0);

begin
    numero <= zero    when u = "0000" else
                    one     when u = "0001" else
                    two     when u = "0010" else
                    three   when u = "0011" else
                    four    when u = "0100" else
                    five    when u = "0101" else
                    six     when u = "0110" else
                    seven   when u = "0111" else
                    eight   when u = "1000" else
                    nine    when u = "1001" else
                    e;
    catodi <= (not punto) & numero;
    anodi  <= cifra_anodo;
end dataflow;

```

Come possiamo osservare dal codice sopra riportato, il valore verrà visualizzato su un unico anodo del display.

Riportiamo nel paragrafo successivo il codice VHDL dell'encoder sul display.

2.4.1: ENCODER_DISPLAY

```

entity encoder_con_display is port (
    a : in std_logic_vector(9 downto 0);
    u : out std_logic_vector(3 downto 0);
    catodi : out std_logic_vector (7 downto 0); -- a partire da: ca a dp
    anodi : out std_logic_vector (7 downto 0) -- a partire da: an7 a an0
);
end encoder_con_display;

architecture structural of encoder_con_display is

component display port (
    punto : in std_logic;                      -- bit da mettere a 0 se si vuole displayare il punto
    u : in std_logic_vector (3 downto 0); -- numero
    catodi : out std_logic_vector (7 downto 0); -- a partire da: ca a dp
    anodi : out std_logic_vector (7 downto 0) -- a partire da: an7 a an0
);
end component;

component Encoder_BCD port (
    a : in std_logic_vector(9 downto 0);
    u : out std_logic_vector(3 downto 0)
);
end component;

signal punto : std_logic := '1';
signal cifra : std_logic_vector (3 downto 0);

begin

display_0 : display PORT MAP(
    punto      => punto,
    u          => cifra,
    catodi    => catodi,
    anodi     => anodi
);

encoder : Encoder_BCD PORT MAP (
    a => a,
    u => cifra
);
u <= cifra;

end structural;

```

L'entità *encoder_con_display* è stata realizzata attraverso un approccio strutturale. Esso include il componente display e l'encoder BCD. Attraverso il port map effettuiamo le opportune interconnessioni, come mostrato nel codice.

CAPITOLO 3: RICONOSCITORE DI SEQUENZE

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

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

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

3.1 SOLUZIONE

Per il riconoscitore di sequenze "1001" abbiamo realizzato l'automa mostrato in figura 3.1. Attraverso un automa a stati finiti possiamo rappresentare il comportamento di una macchina sequenziale. L'ASF è un modello matematico di sistema caratterizzato da un insieme finito di ingressi, uscite e stati. Un automa è quindi definito da

- l'insieme finito Q degli stati
- l'insieme finito I degli ingressi
- l'insieme finito U delle uscite
- la funzione di transizione t che fa passare il sistema da uno stato al successivo
- la funzione di uscita w

È possibile realizzare un automa utilizzando il modello della macchina di Moore o di Mealy. Nella macchina di Moore le uscite dell'automa sono determinate in funzione del solo stato corrente e non dallo stato di ingresso. Nella macchina di Mealy, invece, le uscite dell'automa sono determinate dallo stato attuale e dall'ingresso corrente.

Per questo esercizio abbiamo deciso di usare il modello di Mealy. Di seguito è riportato l'automa:

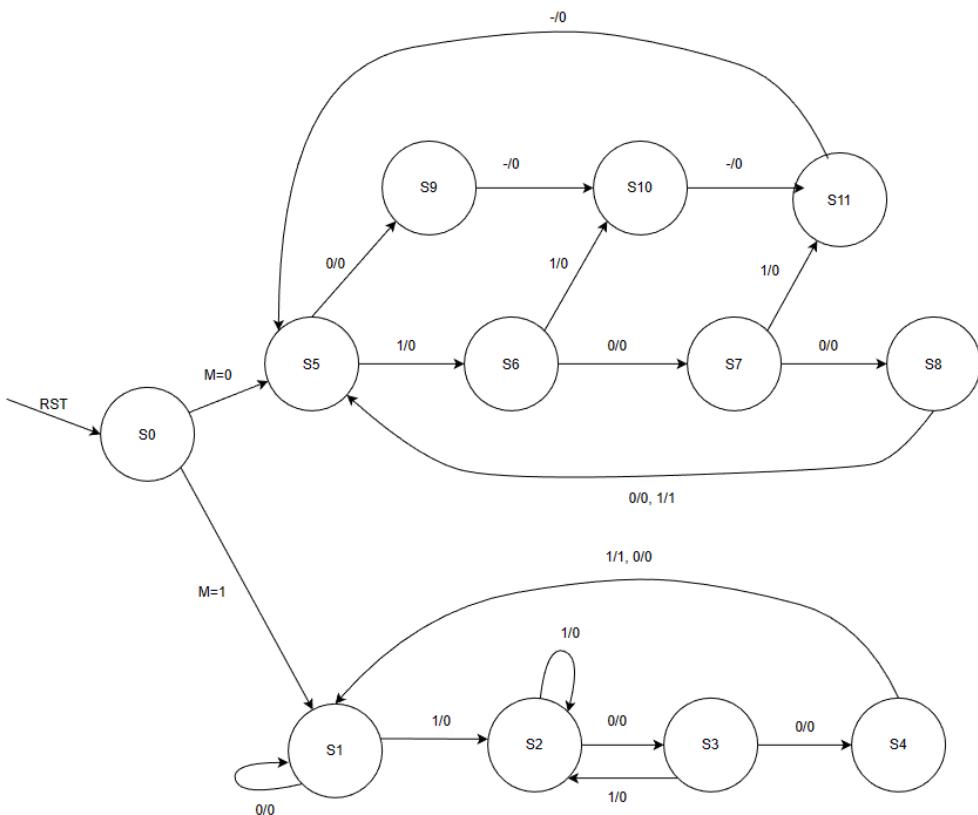


Figura 3.1: Automa

Dallo stato S_0 , una volta scelto il segnale di modo transitiamo nello stato S_5 o S_1 . Come scelta progettuale una volta dato il segnale M non torniamo più nello stato S_0 , a meno che non si alzi il segnale di reset che ci riporta nello stato di partenza.

Riportiamo il codice in VHDL del riconoscitore di sequenza:

```

entity Riconoscitore_2modi is
  port(
    i : in std_logic;
    RST: in std_logic;
    CLK: in std_logic;
    M : in std_logic;
    uscita : out std_logic
  );
end Riconoscitore_2modi;

architecture Behavioral of Riconoscitore_2modi is

type stato is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11);

signal stato_corrente : stato := S0;
signal stato_prossimo : stato;
signal Y : std_logic;

begin

stato_uscita: process (i, stato_corrente)
begin

  case stato_corrente is
    when S0 =>
      if (M = '0') then
        stato_prossimo <= S5;
        Y <= '0';
      else
        stato_prossimo <= S1;
        Y <= '0';
      end if;
    when S1 =>
      if (i = '0') then
        stato_prossimo <= S2;
        Y <= '0';
      else
        stato_prossimo <= S3;
        Y <= '1';
      end if;
    when S2 =>
      if (i = '0') then
        stato_prossimo <= S3;
        Y <= '0';
      else
        stato_prossimo <= S4;
        Y <= '1';
      end if;
    when S3 =>
      if (i = '1') then
        stato_prossimo <= S4;
        Y <= '0';
      else
        stato_prossimo <= S2;
        Y <= '1';
      end if;
    when S4 =>
      if (i = '1') then
        stato_prossimo <= S3;
        Y <= '0';
      else
        stato_prossimo <= S2;
        Y <= '1';
      end if;
    when S5 =>
      if (i = '0') then
        stato_prossimo <= S9;
        Y <= '0';
      else
        stato_prossimo <= S6;
        Y <= '1';
      end if;
    when S6 =>
      if (i = '0') then
        stato_prossimo <= S10;
        Y <= '0';
      else
        stato_prossimo <= S7;
        Y <= '1';
      end if;
    when S7 =>
      if (i = '0') then
        stato_prossimo <= S8;
        Y <= '0';
      else
        stato_prossimo <= S11;
        Y <= '1';
      end if;
    when S8 =>
      if (i = '0') then
        stato_prossimo <= S11;
        Y <= '0';
      else
        stato_prossimo <= S5;
        Y <= '1';
      end if;
    when S9 =>
      if (i = '0') then
        stato_prossimo <= S5;
        Y <= '0';
      else
        stato_prossimo <= S10;
        Y <= '1';
      end if;
    when S10 =>
      if (i = '0') then
        stato_prossimo <= S5;
        Y <= '0';
      else
        stato_prossimo <= S11;
        Y <= '1';
      end if;
    when S11 =>
      if (i = '0') then
        stato_prossimo <= S5;
        Y <= '0';
      else
        stato_prossimo <= S11;
        Y <= '1';
      end if;
  end case;
end process;

```

```

    else
        stato_prossimo <= s1;
        Y <= '0';
    end if;

when S1 =>
    if (i = '1') then
        stato_prossimo <= s2;
        Y <= '0';
    else
        stato_prossimo <= s1;
        Y <= '0';
    end if;

when S2 =>
    if (i = '1') then
        stato_prossimo <= s2;
        Y <= '0';
    else
        stato_prossimo <= s3;
        Y <= '0';
    end if;

when S3 =>
    if (i = '1') then
        stato_prossimo <= s2;
        Y <= '0';
    else
        stato_prossimo <= s4;
        Y <= '0';
    end if;

when S4 =>
    if (i = '1') then
        stato_prossimo <= s1;
        Y <= '1';
    else
        stato_prossimo <= s1;
        Y <= '0';
    end if;

when S5 =>
    if (i = '1') then
        stato_prossimo <= s6;
        Y <= '0';
    else
        stato_prossimo <= s9;
        Y <= '0';
    end if;

when S6 =>
    if (i = '0') then
        stato_prossimo <= s7;
        Y <= '0';
    else
        stato_prossimo <= s10;
        Y <= '0';
    end if;

when S7 =>
    if (i = '0') then
        stato_prossimo <= s8;
        Y <= '0';
    else

```

```

when S7 =>
    if (i = '0') then
        stato_prossimo <= S8;
        Y <= '0';
    else
        stato_prossimo <= S11;
        Y <= '0';
    end if;

when S8 =>
    if (i = '0') then
        stato_prossimo <= S5;
        Y <= '0';
    else
        stato_prossimo <= S5;
        Y <= '1';
    end if;

when S9 =>
    stato_prossimo <= S10;
    Y <= '0';

when S10 =>
    stato_prossimo <= S11;
    Y <= '0';

when S11 =>
    stato_prossimo <= S5;
    Y <= '0';

end case;

```

Inizialmente abbiamo definito un type “stato” in cui sono indicati tutti i possibili valori che esso può assumere. Successivamente abbiamo dichiarato due signal, uno per lo stato corrente e uno per lo stato prossimo, ponendo come stato iniziale S0. Questi segnali ci permettono di realizzare la transizione di stato.

Nella nostra implementazione abbiamo usato due process.

Il process sopra riportato è il processo combinatorio che include nella sensitivity list tutti i segnali letti. Il process sequenziale è, invece, sensibile al solo segnale di clock:

```

end process;

registro: process (CLK)
begin
    if(CLK'event and CLK='1') then
        if(RST='1') then
            stato_corrente <= S0;
            uscita <= '0';
        else
            stato_corrente <= stato_prossimo;
            uscita <= Y;
        end if;
    end if;
end process;

```

Nel caso il segnale di reset sia alto, ci dirigiamo nello stato S0. In caso contrario, aggiorniamo lo stato corrente al prossimo.

3.1.1 SIMULAZIONE

Per effettuare la simulazione creiamo il testbench al cui interno instanziamo il blocco “Riconoscitore_2modi” per effettuarne il test. Definiamo un process per generare il clock (nel nostro caso con periodo di 10 ns) e un process che il testbench eseguirà per effettuare il testing. Scelto un modo iniziale, simuliamo il comportamento del riconoscitore con una sequenza di ingressi arbitrariamente scelta:

```

entity Riconoscitore_2modi_TB is
end Riconoscitore_2modi_TB;

architecture Behavioral of Riconoscitore_2modi_TB is

component Riconoscitore_2modi
port (
    i : in std_logic;
    RST: in std_logic;
    CLK: in std_logic;
    M : in std_logic;
    uscita : out std_logic
);
end component;

signal ingresso : std_logic := '0';
signal reset : std_logic := '0';
signal clock : std_logic := '0';
signal modo : std_logic;
signal usc : std_logic;

begin

riconoscitore : Riconoscitore_2modi
port map(
    i => ingresso,
    RST => reset,
    CLK => clock,
    M => modo,
    uscita => usc
);

stim_proc : process
begin

--resetto
|
    wait for 100 ns;

modo <= '1';
reset <= '1';
wait for 10 ns;
reset <= '0';
wait for 10 ns;

-- provo la sequenza di stati S0-S1-S2-S3-S2-S3-S4-S1 con uscita 1
ingresso <= '1';
wait for 10 ns;
ingresso <= '1';
wait for 10 ns;
ingresso <= '0';
wait for 10 ns;
ingresso <= '1';
wait for 10 ns;
ingresso <= '0';
wait for 10 ns;
ingresso <= '0';
wait for 10 ns;
ingresso <= '1';
wait for 10 ns;

assert usc = '1'
report "error1"
severity failure;

clock_proc : process
begin

clock <= '0';
wait for 5 ns;
clock <= '1';
wait for 5 ns;

end process;

```

Data la sequenza di ingressi “1-1-0-1-0-0-1”, l’uscita che ci aspettiamo è 1. Usiamo l’assert per verificare che l’output sia pari a 1. Se così non fosse si segnala l’errore e si arresta il processo. Lanciando la simulazione verifichiamo che, dati gli input descritti, l’uscita è effettivamente 1:



Figura 3.2: Simulazione

3.2 SOLUZIONE

Per realizzare l’implementazione su board della rete sviluppata precedentemente utilizziamo un divisore di frequenza, l’automa del debouncer e, ovviamente, il riconoscitore di sequenza.

L’uso del divisore di frequenza è motivato dalla richiesta esplicita della traccia di utilizzare un segnale di temporizzazione ricavato dal clock della scheda. A partire dal clock della board di 100MHz abbiamo realizzato un segnale con una frequenza di 50MHz. Questo segnale viene posto in ingresso sia al riconoscitore che al debouncer per realizzare il sincronismo richiesto.

Dato l’utilizzo di due pulsanti della scheda per l’acquisizione degli input, abbiamo usufruito del codice a disposizione nel materiale didattico per poter realizzare l’automa del debouncer. L’uso del debouncer è dovuto ai rimbalzi imprevedibili che possono verificarsi alla pressione di un pulsante sull’FPGA. Le transizioni di apertura/chiusura spurie che i pulsanti sono in grado di generare quando vengono premuti possono essere lette come pressioni multiple in un tempo breve che portano in inganno il programma. Infatti, quando un pulsante viene premuto o rilasciato, può brevemente subire oscillazioni ad alta velocità. È evidente, quindi, la necessità di risolvere questo problema ed evitare che ci siano così dei risultati indesiderati. Dato il segnale generato alla pressione di un pulsante, il nostro obiettivo è quello di elaborare questo segnale e ottenere un singolo impulso per ogni pressione del pulsante, filtrando le oscillazioni.

Per quanto concerne il codice in vhdl, abbiamo apportato una piccola modifica al codice del debouncer inserendo solo il divisore di frequenza e aggiungendo il controllo nello stato “controllo_pressione”:

```

when controllo_pressione =>
    if (cont /= max_count-1) then
        cont := cont+1;
    else
        if (div = '1') then
            if (BTN = '1') then
                btn_state<=premuto;
                cont:=0;
                CLEARED_BTN <='1';
            else
                btn_state <= non_premuto;
            end if;
        end if;
    end if;
end when;

```

Il riconoscitore di sequenza è identico a quello descritto precedentemente. Come modifica abbiamo inserito il divisore di frequenza, due ingressi (“m_read”, “i_read”) usati per la lettura del segnale di modo e dell’ingresso (che poi saranno collegati a 2 bottoni sulla board), ed infine abbiamo aggiunto un’uscita in più per visualizzare lo stato in cui ci troviamo, codificato su 4 bit.

Riportiamo di seguito il codice del process sequenziale del riconoscitore per visualizzare le modifiche:

```

registro: process (CLK)
begin
    if(CLK'event and CLK='1') then
        if(RST='1') then
            stato_corrente <= S0;
            uscita <= '0';
        else
            if (div = '1') then
                if (stato_corrente = S0 and m_read='1') then
                    stato_corrente <= stato_prossimo;
                    state <= state_tmp;
                    uscita <= Y;
                elsif (stato_corrente /= S0 and i_read ='1') then
                    stato_corrente <= stato_prossimo;
                    state <= state_tmp;
                    uscita <= Y;
                end if;
            end if;
        end if;
    end if;

end process;

```

Il riconoscitore su board ha come ingresso due bottoni e due switch per l’inserimento dei dati in ingresso. L’uscita e lo stato corrente verranno visualizzati sui led.

```

entity Riconoscitore_su_board is port (
    clock      : in std_logic;
    reset      : in std_logic;
    button1    : in std_logic;
    button2    : in std_logic;
    switch1    : in std_logic;
    switch2    : in std_logic;
    led         : out std_logic;
    led_stato  : out std_logic_vector(3 downto 0)
);
end Riconoscitore_su_board;

architecture structural of Riconoscitore_su_board is

component Riconoscitore_2modi is
    port(
        i : in std_logic;
        div : in std_logic;
        RST: in std_logic;
        CLK: in std_logic;
        M : in std_logic;
        i_read : in std_logic;
        M_read : in std_logic;
        uscita : out std_logic;
        state  : out std_logic_vector(3 downto 0)
    );
end component;

```

```

component automa_debouncer is
    generic (
        CLK_period: integer := 10; -- periodo del clock in nanosec
        btn_noise_time: integer := 30 --durata dell'oscillazione in nanosec
    );
    Port ( RST : in STD_LOGIC;
            div : in std_logic;
            CLK : in STD_LOGIC;
            BTN : in STD_LOGIC;
            CLEARED_BTN : out STD_LOGIC);
end component;

component Divisore_freq is
    generic (
        f_in : integer := 100000000;
        f_out : integer := 50000000
    );
    port (
        reset : in std_logic;
        clock : in std_logic;
        div : out std_logic
    );
end component;

signal sig_div : std_logic;
signal sig_in : std_logic;
signal sig_m : std_logic;

begin

```

I signal creati sono stati utilizzati per effettuare il port map dei componenti. In particolare `sig_in` e `sig_m` vengono associati, rispettivamente, agli ingressi “`i_read`” e “`m_read`” del riconoscitore e ai segnali di uscita “cleared” dell’automa_debouncer:

```

riconoscitore : Riconoscitore_2modi port map (
    i      => switch1,
    div    => sig_div,
    RST    => reset,
    CLK    => clock,
    M      => switch2,
    i_read => sig_in,
    M_read => sig_m,
    uscita => led,
    state   => led_stato
);

deb_in : automa_debouncer port map (
    RST      => '0',
    div      => sig_div,
    CLK      => clock,
    BTN      => button1,
    CLEARED_BTN => sig_in
);

deb_m : automa_debouncer port map (
    RST      => '0',
    div      => sig_div,
    CLK      => clock,
    BTN      => button2,
    CLEARED_BTN => sig_m
);

divisore : Divisore_freq port map (
    reset   => '0',
    clock   => clock,
    div     => sig_div);

```

CAPITOLO 4: SHIFT REGISTER

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

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

4.1: COS'È LO SHIFT REGISTER

Lo shift register, o registro a scorrimento, è fatto da un insieme di registri che consentono di memorizzare stringhe di bit. La particolarità di questo componente è che può shiftare a destra o sinistra, il che si rivela particolarmente utile essendo che in binario questo significa anche dividere e moltiplicare. Il comportamento di questo componente può essere descritto utilizzando sia un approccio comportamentale che un approccio strutturale; nel secondo caso dovremo scegliere il singolo elemento che lo compone, vale a dire il flip-flop, con i quali sarà possibile memorizzare ciascun bit, inoltre, è possibile distinguere diverse architetture progettuali, che prevedono di definire il modo in cui i dati possono essere caricati e scaricati dal registro stesso, e quindi se in parallelo o in serie.

4.2: APPROCCIO COMPORTAMENTALE

Per realizzare il punto a dell'esercizio e quindi implementare il registro in modo comportamentale, abbiamo progettato un registro in grado di memorizzare 4 bit e che preveda in ingresso un segnale di abilitazione (E), l'ingresso da inserire nello shift register (X) e un ingresso su 2 bit di selezione (S), l'uscita è invece stata rappresentata su un bit.

Al fronte di salita del clock, se il segnale di reset è alto, i valori contenuti nello shift register vengono azzerati, così da partire da uno stato noto. Quando, invece, il segnale di abilitazione si alza, si distinguono quattro casi a seconda della selezione in ingresso, in particolare:

- Se la selezione dovesse essere '00', si effettua uno shift a destra di una posizione, quindi in ingresso al primo registro mettiamo il valore di ingresso X, mentre nei registri successivi carichiamo il valore contenuto dai registri precedenti. L'uscita in questo caso è il valore contenuto dall'ultimo registro, che alla fine del process sarà tmp(2).
- Se la selezione dovesse essere '01', si effettua uno shift a sinistra di una posizione, quindi in ingresso all'ultimo registro (cioè quello più a destra) mettiamo il valore di ingresso X, mentre nei registri successivi carichiamo il valore contenuto nei registri precedenti (cioè quelli allo loro destra). L'uscita in questo caso è il valore contenuto dal primo registro, che alla fine del process sarà tmp(1).
- Se la selezione dovesse essere '10', si effettua uno shift a destra di due posizioni, quindi in ingresso ai primi due registri mettiamo il valore di ingresso X, mentre nei registri successivi carichiamo il valore contenuto dai due registri precedenti, quindi in tmp(2) verrà caricato il valore di tmp(0), mentre in tmp(3) verrà caricato il valore tmp(1). L'uscita è il valore contenuto dall'ultimo registro, che alla fine del process sarà tmp(1).
- Se la selezione dovesse essere '11', si effettua uno shift a sinistra di due posizioni, quindi in ingresso agli ultimi due registri mettiamo il valore di ingresso X, mentre nei registri successivi (cioè quelli più a sinistra) carichiamo il valore contenuto nei due registri precedente, quindi

in `tmp(0)` verrà caricato `tmp(2)`, mentre in `tmp(1)` verrà caricato `tmp(3)`. L'uscita in questo caso è il valore contenuto nel primo registro, che alla fine del process sarà `temp(2)`.

Il codice in VHDL è riportato di seguito:

```

entity Shift_register is port(
    CLK, RST, X : in std_logic;
    E : in std_logic;
    S : in std_logic_vector(1 downto 0);
    U : out std_logic);
end Shift_register;

architecture Behavioral of Shift_register is
signal tmp: std_logic_vector(0 to 3);
signal c: std_logic;
begin
    process(CLK)
    begin
        if (rising_edge(CLK)) then
            if (RST='1') then
                tmp <= (others=>'0');
            else if(E ='1') then
                case S is
                    when "00" =>--shift a destra di 1
                        tmp(0) <= X;
                        tmp(1) <= tmp(0);
                        tmp(2) <= tmp(1);
                        tmp(3) <= tmp(2);
                        c<=tmp(2);

                    when "01" =>--shift a sinistra di 1
                        tmp(3) <= X;
                        tmp(2) <= tmp(3);
                        tmp(1) <= tmp(2);
                        tmp(0) <= tmp(1);
                        c<=tmp(1);

                    when "10" =>--shift a destra di 2
                        tmp(0) <= X;
                        tmp(1) <=X;
                        tmp(2) <= tmp(0);
                        tmp(3) <= tmp(1);
                        c<=tmp(1);

                    when "11" =>--shift a sinistra di 2
                        tmp(0) <= tmp(2);
                        tmp(1) <=tmp(3);
                        tmp(2) <= X;
                        tmp(3) <= X;
                        c<=tmp(2);

                    when others=>report "unreachable" severity failure;
                end case;
            end if;
        end if;

        end if;
    end process;
    U <= c;
end Behavioral;

```

Per eseguire il testbench abbiamo in primis eseguito un reset globale iniziale, dopodiché abbiamo fornito abilitazione, selezione e valore in ingresso, diverse volte, per poterne osservare il comportamento.

```

rst <= '1';      wait for 85ns;
wait for 100ns;
rst <='0';       input <= '1';
ab <="11";
input <= '1';    enable <='1';
ab <="00";       wait for 15 ns;
enable <='1';   enable <='0';
wait for 15 ns;
enable <='0';   wait for 85ns;

wait for 85ns;  input <= '1';
ab <="01";
input <= '0';    enable <='1';
ab <="01";       wait for 15 ns;
enable <='1';   enable <='0';
wait for 15 ns;
enable <='0';   wait for 85ns;

wait for 85ns;  input <= '0';
ab <="10";
input <= '1';    enable <='1';
ab <="10";       wait for 15 ns;
enable <='1';   enable <='0';
wait for 15 ns;
enable <='0';   wait for 85ns;

```

La simulazione di questo programma è riportata in figura 4.1

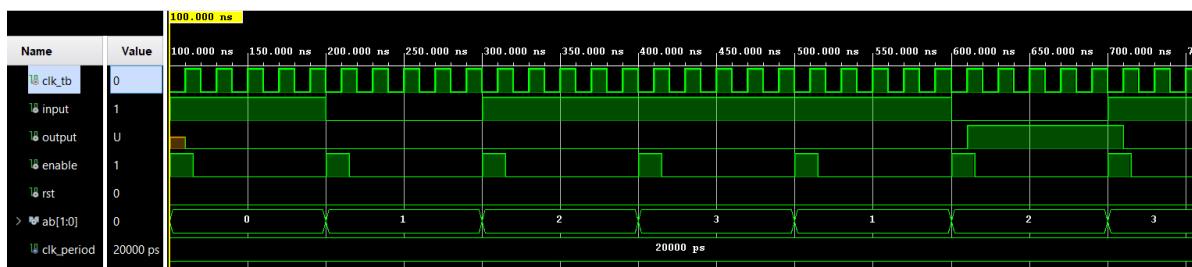


Figura 4.1: Simulazione caso comportamentale

Come si può osservare, essendo lo shift register inizialmente resettato, i valori di partenza, in uscita sono sempre bassi, avendo scelto di fornire le selezioni in maniera “ordinata”; quando viene fornita la selezione “01” per la seconda volta, invece, possiamo notare come effettivamente il valore in uscita sia alto, essendo che in quel momento il valore memorizzato dallo shift register è “0001”.

4.3: APPROCCIO STRUTTURALE

Per la realizzazione dello shift register in modo strutturale, sono stati impiegati quattro flip-flop di tipo D e quattro multiplexer 4:1, necessari per poter gestire opportunamente i 4 casi visti prima (shift a destra di una posizione, shift a sinistra di una posizione, shift a destra di due posizioni e shift a sinistra di due posizioni).

L'architettura realizzata è visibile in figura 4.2.

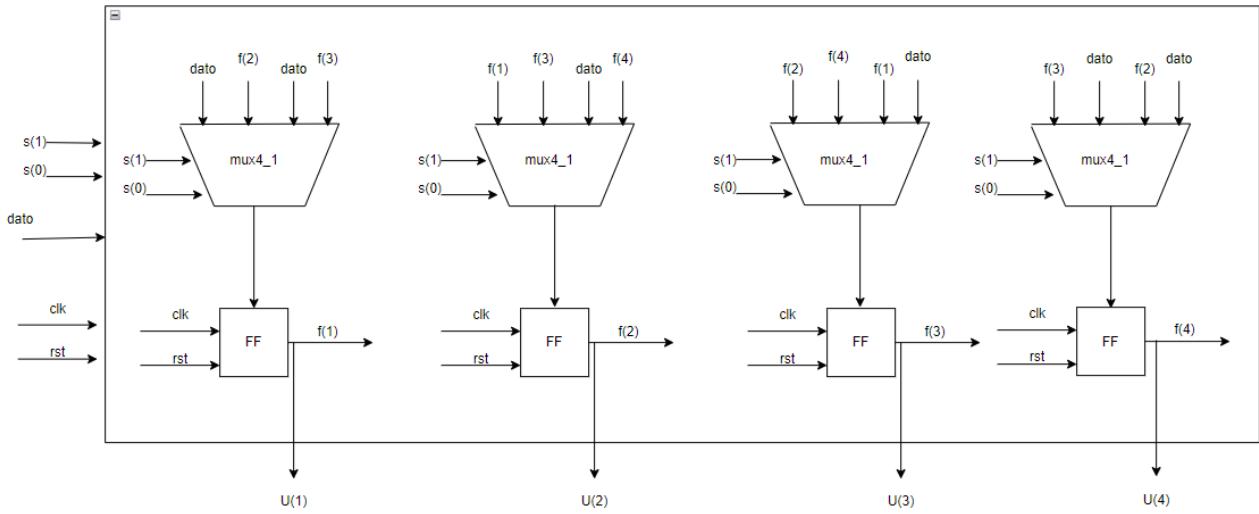


Figura 4.2: Architettura SR strutturale

4.3.1: FLIP-FLOP D

Il flip-flop D rappresenta la cella elementare di memoria dello shift register. Questo componente è stato realizzato in modo da essere sensibile al fronte di salita del clock e prevede in ingresso, oltre al segnale di clock stesso e al reset, che mi permette di azzerarne il contenuto, anche un segnale di abilitazione e il dato che verrà opportunamente caricato all'alzarsi del segnale di abilitazione.

Il codice è riportato di seguito:

```

entity FlipFlop is port(
    D: in std_logic;
    CLK: in std_logic;
    RST: in std_logic;
    E: in std_logic;
    Q: out std_logic
);

end FlipFlop;

architecture Behavioral of FlipFlop is

begin
    FF: process(CLK)
    begin
        if(CLK' event AND CLK='1') then
            if(RST='1') then
                Q<='0';
            else if(E='1') then
                Q<=D;
            end if;
        end if;
    end process;
end Behavioral;

```

4.3.2: REALIZZAZIONE

Per la risoluzione del punto b, che prevedeva di realizzare lo shift register in modo strutturale, abbiamo optato per la visualizzazione in uscita di tutti i dati contenuti nello shift e non solo dell'ultimo bit. I multiplexer sono stati utilizzati per selezionare il dato da caricare nel flip-flop. In particolare, la selezione dei multiplexer corrisponde alla selezione in ingresso allo shift, sulla base della quale viene stabilito il tipo di shift da effettuare e quindi se a destra o a sinistra e se di una posizione o di due. Gli ingessi del multiplexer variano a seconda del multiplexer legato al flip-flop, e quindi ad esempio: il primo mux, connesso al primo ff, cioè quello atto a contenere il primo bit del registro (quello più a sinistra), presenta in ingresso rispettivamente: il dato, l'uscita del secondo ff, di nuovo il dato e l'uscita del terzo ff, in questo modo:

- se la selezione dovesse essere 00, verrebbe preso il primo ingresso del mux, che è il dato, infatti in questo caso (con selezione 00) bisogna effettuare uno shift verso destra di una posizione, quindi in ingresso al ff dovrà essere caricato il dato I_1 ;
- se la selezione dovesse essere 01, verrebbe preso, e posto in ingresso al primo ff, il secondo ingresso del mux, che contiene l'uscita del secondo ff, infatti in questo caso bisognerebbe effettuare uno shift verso sinistra di una posizione e dunque nel primo ff verrebbe caricato il dato contenuto nel ff successivo, appunto il secondo;
- se la selezione dovesse essere 10, verrebbe preso il terzo ingresso del mux, che contiene nuovamente il dato, infatti in questo caso bisognerebbe effettuare uno shift verso destra di due posizioni, quindi l'ingresso del primo flip-flop dovrà essere il dato I_1 ;
- infine, se la selezione dovesse essere 11, verrebbe preso il quarto ingresso al mux, che è legato all'uscita del terzo ff, infatti in questo caso bisognerebbe effettuare uno shift verso sinistra di due posizioni, quindi nel primo ff verrebbe caricato il valore contenuto nel terzo

Un ragionamento analogo può essere applicato a ciascuna coppia multiplexer-ff, e quindi lo schema finale è fornito dal codice sottostante. Le uscite di ciascun ff sono collegate all'uscita dello shift register.

```

entity Shift_register_s is port(
  I: in std_logic;
  S: in std_logic_vector(1 downto 0);
  CLK: in std_logic;
  E: in std_logic;
  RST: in std_logic;
  U: out std_logic_vector(0 to 3)
);
end Shift_register_s;

architecture Structural of Shift_register_s is
  component mux4_1 is
    port (
      b: in std_logic_vector(0 to 3);
      s: in std_logic_vector(1 downto 0);
      z: out std_logic
    );
  end component;

  component FlipFlop is port(
    D: in std_logic;
    CLK: in std_logic;
    E : in std_logic;
    RST: in std_logic;
    Q: out std_logic
  );
  end component;

  signal F: std_logic_vector(0 to 3);
  signal M : std_logic_vector(0 to 3);
  signal C: std_logic;

begin

  mux1: mux4_1 port map(
    b(0) => I,
    b(1) => F(1),
    b(2) => I,
    b(3) => F(2),
    s => S,
    z => M(0)
  );

  mux2: mux4_1 port map(
    b(0) => F(0),
    b(1) => F(2),
    b(2) => I,
    b(3) => F(3),
    s => S,
    z => M(1)
  );

  mux3: mux4_1 port map(
    b(0) => F(1),
    b(1) => F(3),
    b(2) => F(0),
    b(3) => I,
    s => S,
    z => M(2)
  );

  mux4: mux4_1 port map(
    b(0) => F(2),
    b(1) => I,
    b(2) => F(1),
    b(3) => I,
    s => C
  );

```

```

    s => s,
    z => M(3)
);

ff1: FlipFlop port map(
    CLK => CLK,
    RST => RST,
    E => E,
    D => M(0),
    Q => F(0)
);
U(0)<= F(0);

ff2: FlipFlop port map(
    CLK => CLK,
    RST => RST,
    E => E,
    D => M(1),
    Q => F(1)
);
U(1)<= F(1);

ff3: FlipFlop port map(
    CLK => CLK,
    RST => RST,
    E => E,
    D => M(2),
    Q => F(2)
);
U(2)<= F(2);

. . .
ff4: FlipFlop port map(
    CLK => CLK,
    RST => RST,
    E => E,
    D => M(3),
    Q => F(3)
);
U(3)<= F(3);

end Structural;

```

Come si può osservare dalla simulazione riportati in figura 4.3 il comportamento è del tutto analogo a quello del caso comportamentale, con l'unica sostanziale differenza che l'uscita è data dal contenuto dell'intero shift register.

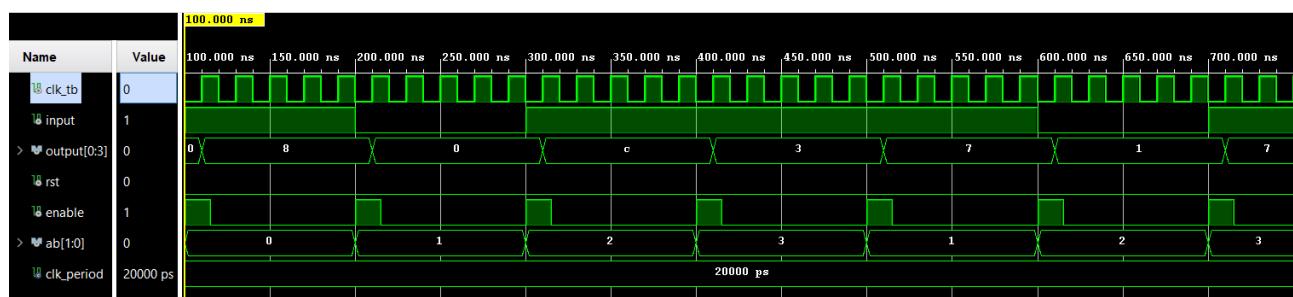


Figura 4.3: Simulazione caso strutturale

CAPITOLO 5: CRONOMETRO

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

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

2) 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).

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

5.1: SOLUZIONE

Per la progettazione del cronometro abbiamo, prima di tutto, identificato di quale componente possiamo usufruire per realizzare il comportamento dell'oggetto desiderato. Questo componente è il contatore. In particolare, facciamo uso di tre contatori: uno per i secondi, uno per i minuti e uno per le ore. Quello dei secondi e dei minuti è un contatore modulo 60 che realizzeremo a partire da un contatore modulo 64 applicando opportunamente il reset, quando il conteggio effettuato arriva a 59. Lo stesso procedimento viene effettuato per il contatore delle ore, il cui conteggio arriverà a 23 a partire da un contatore modulo 32.

Per la realizzazione dei singoli contatori abbiamo deciso di usare un approccio strutturale. Partiremo, quindi, dal componente base dei contatori, ovvero il flip flop T, per la descrizione e la progettazione dell'oggetto richiesto.

Di seguito riportiamo lo schema per realizzare un generico contatore modulo 24 a partire da uno modulo 32. Necessitiamo, quindi, di 5 flip flop posti in cascata:

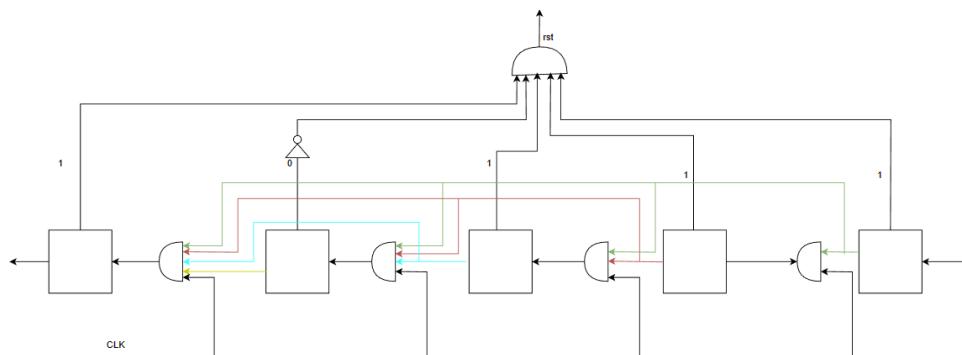


Figura 5.1: Contatore modulo 24

Questo contatore è sviluppato secondo un modello parallelo. In questo caso, il primo contatore commuta ogni volta che arriva il clock, il secondo commuta quando arriva il clock e il primo è alto, il

terzo invece quando arriva il clock e sia il primo che il secondo sono alti, e così via. Realizziamo questo comportamento attraverso delle porte and.

Per far sì che un contatore sia modulo 24, esso deve contare fino a 23(10111 in binario). Inseriamo, allora, all'interno di una and il valore "10111", negando i bit pari a '0'. In questo modo, quando avremo raggiunto il valore 23 si attiverà il reset e la macchina partirà da 0.

Dopo aver realizzato i singoli contatori per il conteggio dei secondi, dei minuti e delle ore, costruiremo il cronometro secondo un approccio strutturale, interconnettendo opportunamente i componenti realizzati. Useremo un divisore di frequenza in modo da dividere la frequenza di 100MHz della scheda per ottenere una di 1Hz, adeguata per il conteggio dei secondi.

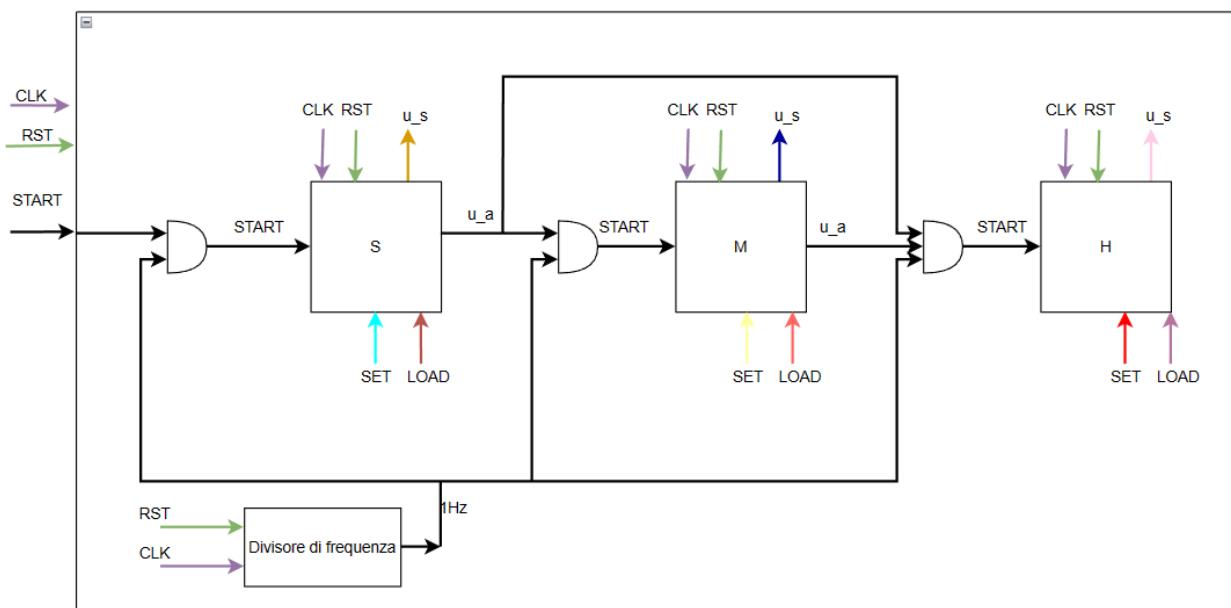


Figura 5.2: Cronometro

Come possiamo osservare dalla figura 5.2, passiamo a tutti i componenti il segnale di clock. Successivamente come enable ai contatori useremo: per i secondi il segnale del divisore di frequenza in and con lo start della macchina (usato per far partire il conteggio); per il contatore dei minuti la and tra il segnale del divisore e l'abilitazione al conteggio proveniente dai secondi (la quale diviene alta nel momento in cui i secondi arrivano a 59); per le ore viene messo in ingresso come start sempre il segnale di divisore, l'abilitazione proveniente dai secondi e dai minuti (necessarie entrambe perché solo quando sia i secondi che i minuti arrivano a 59 si può abilitare il conteggio delle ore. In questo modo se volessimo contare la prima ora avremmo: arrivati a 59:59:00 scatterà, successivamente, l'ora 00:00:01).

Da notare che il segnale del divisore viene usato non come clock ma come enable del clock, per abilitare il conteggio.

5.1.1 Implementazione VHDL

5.1.1.1 Implementazione contatore con ff

Riportiamo, prima di tutto, l'implementazione del flip flop:

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ffT is  Port (
  CLK: in std_logic;
  RST: in std_logic;
  RST1: in std_logic;
  ENABLE: in std_logic;
  LOAD : in std_logic;
  SET: in std_logic;
  Y: out std_logic
);
end ffT;

architecture Behavioral of ffT is

signal ty: std_logic;

begin

ff: process(CLK)
begin
  if(falling_edge(CLK)) then
    if(RST='1') then
      ty<='0';
    else
      if(LOAD='1') then
        ty<=SET;
      else
        if(RST1='1') then
          ty<='0';
        else
          if(ENABLE='1') then
            ty<=not ty;
          end if;
        end if;
      end if;
    end if;
  end if;
end process;
Y<=ty;
end Behavioral;
```

Questo componente possiede in ingresso, oltre al segnale di clock e di reset, un segnale di set e di load, per impostare il valore memorizzato all'attivazione del segnale di load, e un segnale di enable per abilitare il flip flop a cambiare il suo valore. Notiamo, inoltre, che abbiamo inserito un segnale RST1 che viene usato per azzerare il valore memorizzato quando il conteggio arriva al suo valore finale. Come già detto in precedenza, interconnettendo opportunamente questi componenti posso realizzare un contatore, di cui riportiamo l'implementazione:

```

entity cont_s_m is Port(
    CLK: in std_logic;
    RST: in std_logic;
    LOAD: in std_logic;
    SET: in std_logic_vector(0 to 5);
    START: in std_logic;-- end no e start cronom
    u_s: out std_logic_vector(0 to 5);
    u_a: out std_logic
);
end cont_s_m;

architecture Structural of cont_s_m is

component fft is
    Port (
        CLK: in std_logic;
        RST: in std_logic;
        RST1: in std_logic;
        ENABLE: in std_logic;
        LOAD: in std_logic;
        SET: in std_logic;
        Y: out std_logic
    );
end component;
signal S1: std_logic;
signal S2: std_logic;
signal S3: std_logic;
signal S4: std_logic;
signal S5: std_logic;
signal S6: std_logic;
signal reset: std_logic;
begin
reset<= S6 and S5 and S4 and not S3 and S2 and S1;
u_a<=reset;
ff1: fft port map(
    CLR => CLK,
    RST => RST,
    RST1 => reset and START,
    ENABLE => START,
    LOAD => LOAD,
    SET => SET(5),
    Y=>S1
);
ff2: fft port map(
    CLK => CLK,
    RST => RST,
    RST1 => reset and START,
    ENABLE => S1 and START,
    LOAD => LOAD,
    SET => SET(4),
    Y=>S2
);
ff3: fft port map(
    CLK => CLK,
    RST => RST,
    RST1 => reset and START,
    ENABLE => S1 and S2 and START,
    LOAD => LOAD,
    SET => SET(3),
    Y=>S3
);

```

```

Y=>s3
);
ff4: fft port map(
    CLK => CLK,
    RST => RST,
    RST1 => reset and START,
    ENABLE => S1 and S2 and S3 and START,
    LOAD => LOAD,
    SET => SET(2),
    Y=>s4
);
ff5: fft port map(
    CLK => CLK,
    RST => RST,
    RST1 => reset and START,
    ENABLE => S1 and S2 and S3 and S4 and START,
    LOAD => LOAD,
    SET => SET(1),
    Y=>s5
);
ff6: fft port map(
    CLK => CLK,
    RST => RST,
    RST1 => reset and START,
    ENABLE => S1 and S2 and S3 and S4 and S5 and START,
    LOAD => LOAD,
    SET => SET(0),
    Y=>s6
);
u_s <= s6 & s5 & s4 & s3 & s2 & s1;
end Structural;

```

Abbiamo riportato l'implementazione del contatore dei secondi e minuti. Quella delle ore è analoga.

Consideriamo, per esempio, il contatore dei secondi per fare alcune considerazioni. Possiamo notare come al segnale "u_a", ovvero quello di abilitazione al conteggio dei minuti, sia assegnato il valore del reset, che a sua volta è pari alla and tra i bit di uscita dei singoli flip flop. Infatti, quando il valore del conteggio arriva a 59 si alza sia il segnale di reset che di u_a, in modo tale che dopo il cinquantanovesimo secondo si azzerino i secondi e si incrementino i minuti.

Notiamo inoltre come nel segnale RST1 passiamo il signal reset con lo START. Questo si rende necessario per fare in modo che i secondi dopo essere arrivati a 59 ritornino a zero solo dopo che il secondo sia trascorso. Non mettendo quella and potremmo incorrere nel problema che il 59esimo secondo non duri effettivamente un secondo ma che si azzeri al successivo colpo di clock della scheda.

L'uscita del contatore è data dal concatenamento di tutti i bit di uscita dei flip flop.

5.1.1.2 Implementazione cronometro

Nel codice vhdl del cronometro, riportato di seguito, possiamo quindi notare l'approccio strutturale usato per la sua costruzione, avvalendoci dello schema mostrato nella figura 5.2. Oltre, quindi, ai contatori abbiamo usato un divisore di frequenza ponendo come frequenza in uscita quella di 1Hz.

Di seguito riporteremo anche il codice di simulazione per testare l'oggetto. Notiamo come nella simulazione possiamo porre il divisore ad una frequenza più alta rispetto ad 1Hz per evitare di aspettare troppo tempo. Bisogna ovviamente porre attenzione nel riportare quest'oggetto alla frequenza di 1Hz nel momento in cui si prova il sistema sulla board.

Il codice vhdl per il cronometro è il seguente:

```
entity CRONOMETRO is
  Port (
    CLK: in std_logic;
    RST: in std_logic;
    START: in std_logic;
    LOAD_s: in std_logic;
    LOAD_m: in std_logic;
    LOAD_h: in std_logic;
    SET_s: in std_logic_vector(0 to 5);
    SET_m: in std_logic_vector(0 to 5);
    SET_h: in std_logic_vector(0 to 4);
    u_sec: out std_logic_vector(0 to 5);
    u_min: out std_logic_vector(0 to 5);
    u_ore: out std_logic_vector(0 to 4)
  );
end CRONOMETRO;

architecture Structural of CRONOMETRO is

signal nc: std_logic;
signal abmin: std_logic;
signal abore: std_logic;
signal a: std_logic;
signal a1: std_logic;
signal a2: std_logic;

component Divisore
  generic(
    clock_frequency_in : integer := 10000000;--100MHz
    --clock_frequency_out : integer := 1000000
    clock_frequency_out : integer := 1 --1 Hz
  );
  Port (
    clock_in : in STD_LOGIC;
    reset : in STD_LOGIC;
    clock_out : out STD_LOGIC
  );
end component;

component cont_s_m
  port(
    CLK: in std_logic;
    RST: in std_logic;
    LOAD: in std_logic;
    SET: in std_logic_vector(0 to 5);
    START: in std_logic;
    u_s: out std_logic_vector(0 to 5);
    u_a: out std_logic
  );
end component;
```

```
component cont_h is Port(
```

```
    CLK: in std_logic;
```

```
    RST: in std_logic;
```

```
    LOAD: in std_logic;
```

```
    SET: in std_logic_vector(0 to 4);
```

```
    START: in std_logic;-- end nc e start cronometro
```

```
    u_s: out std_logic_vector(0 to 4)
```

```
);
```

```
end component;
```

```
begin
```

```
clock_div: Divisore port map(
```

```
    clock_in => CLK,
```

```
    reset => RST,
```

```
    clock_out=>nc
```

```
);
```

```
a2<=START and nc;
```

```
s: cont_s_m port map(
```

```
    CLK => CLK,
```

```
    RST => RST,
```

```
    LOAD =>LOAD_s,
```

```
    SET =>SET_s,
```

```
    START =>a2,
```

```
    u_s => u_sec,
```

```
    u_a => abmin
```

```
);
```

```
a1<=abmin and nc;
```

```
m: cont_s_m port map(
```

```
    CLK => CLK,
```

```
    START => a1,
```

```
    RST => RST,
```

```
    LOAD =>LOAD_m,
```

```
    SET =>SET_m,
```

```
    u_s => u_min,
```

```
    u_a=> abore
```

```
);
```

```
a<= abore and abmin and nc;
```

```
h: cont_h port map(
```

```
    CLK => CLK,
```

```
    START => a,
```

```
    RST => RST,
```

```
    LOAD =>LOAD_h,
```

```
    SET =>SET_h,
```

```
    u_s => u_ore
```

```
);
```

```
end Structural;
```

5.1.2: SIMULAZIONE

Del testbench del cronometro abbiamo riportato direttamente la parte del settaggio dei parametri di ingresso, in quanto il codice del test è sempre analogo e il procedimento effettuato per la sua realizzazione è sempre lo stesso.

Come possiamo notare, abbiamo abilitato il set per i secondi, i minuti e le ore, a cui abbiamo assegnato come valore di partenza i valori 59:59:23 rispettivamente. Partendo da questi parametri, i valori successivi che ci aspettiamo sono 00:00:00. È possibile effettuare il settaggio di qualsiasi

parametro in qualsiasi momento abilitando opportunamente i segnali di load. Il conteggio proseguirebbe, poi, a partire da quei valori. Per far partire il tutto bisogna, ovviamente, dare il segnale di start:

```

test: process
begin
    rst <= '1';
    wait for 100ns; --global reset
    rst <='0';

    l_s <='1';
    l_m <='1';
    l_h<='1';

    s_s <="111011";
    s_m <="111011";
    s_h<="10111";

    wait for 20ns;
    l_s<='0';
    l_m<='0';
    l_h<='0';

    st<='1';

```

Possiamo osservare come i risultati ottenuti dal testbench sono concordi con quanto detto fin ora:

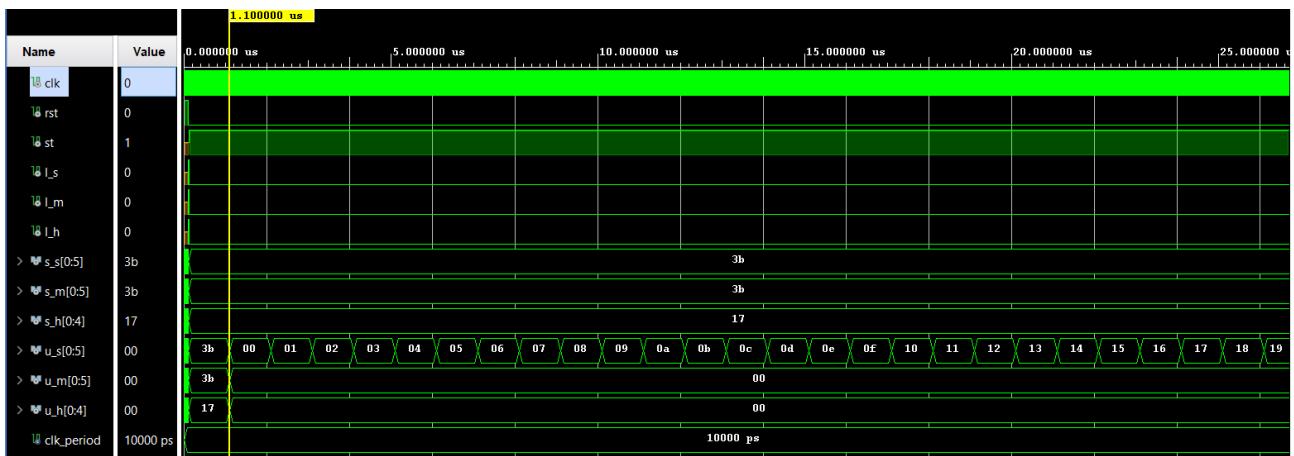


Figura 5.3: Testbench

Infatti, dopo i valori di settaggio dati, il cronometro riparte da 00 sia per i secondi, i minuti e le ore e procede poi al conteggio come previsto.

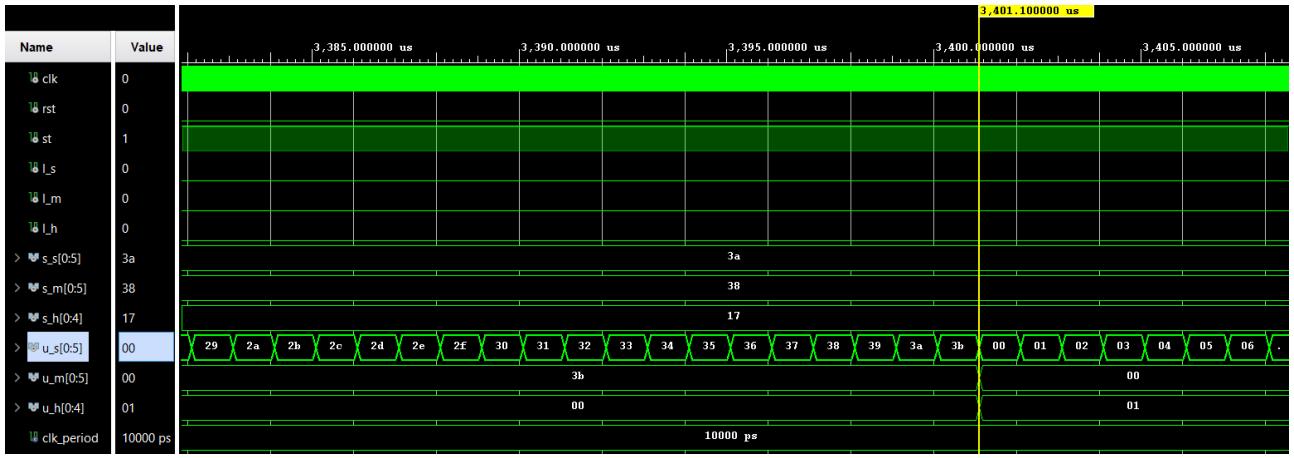


Figura 5.4: Testbench

L'immagine immediatamente sopra riportata dimostra come, una volta raggiunti sia i secondi che i minuti a 59, si passi successivamente a 00 per questi ultimi e a 01 per le ore.

Come si può notare, la rappresentazione riportata dei numeri non è decimale ma esadecimale. Il valore di 3b corrisponde a 59 in decimale e 17 a 23.

Tutto il procedimento viene quindi eseguito correttamente.

5.2: SOLUZIONE

Dovendo utilizzare il display a 7 segmenti, per mostrare il cronometro su board, utilizziamo l'implementazione che ci è stata fornita. In questo caso il numero di cifre che vogliamo illuminare sono 6, 2 per ogni parametro. Per questo motivo, il contatore modulo 8, che viene utilizzato per gestire il display, è stato leggermente modificato per far sì che il suo conteggio arrivi fino a 6.

Creeremo, quindi, secondo un approccio strutturale, un componente che include il cronometro e il display, nonché i debouncer dei bottoni utilizzati per il load, reset. Le uscite del cronometro andranno in ingresso al display. Prima di passarli a questo dispositivo dobbiamo concatenare i risultati dei secondi, minuti e ore per formare un'unica stringa. Essendo ogni cifra del display a 4 bit, non ci basta solo concatenare quei parametri ma dobbiamo aggiungere degli 0. In particolare, i secondi e i minuti sono formati da 6 bit e quindi i primi 4 bit meno significativi corrisponderanno alla prima cifra(quella più a destra) e i rimanenti 2 dovranno essere concatenati con due zeri affinché possano corrispondere con la seconda cifra. Per le ore il procedimento è lo stesso, bisogna solo aggiungere un bit 0 in più per la seconda cifra, essendo rappresentate su 5 bit.

5.2.1 Implementazione VHDL

```
entity c_on_board is
    Port (
        CLK: in std_logic;
        b_RST: in std_logic;
        START: in std_logic;
        b_LOAD_s: in std_logic;
        b_LOAD_m: in std_logic;
        b_LOAD_h: in std_logic;
        switch_set_sec_m: in std_logic_vector(0 to 5);
        switch_set_ore: in std_logic_vector(0 to 4);
        anodes : out STD_LOGIC_VECTOR (7 downto 0);
        cathodes : out STD_LOGIC_VECTOR (6 downto 0)
    );
end c_on_board;

architecture Structural of c_on_board is

signal s_c: std_logic_vector(0 to 5);
signal m_c: std_logic_vector(0 to 5);
signal h_c: std_logic_vector(0 to 4);
signal t_c: std_logic_vector(23 downto 0);
signal a: std_logic;

--bottom
signal br:std_logic;
signal bsec: std_logic;
signal bm: std_logic;
signal bh: std_logic;
```

Per immettere il set dell'orario usiamo gli switch della board, in combinazione con i bottoni per il caricamento. Notiamo come abbiamo usato un unico ingresso da 5 bit per poter caricare entrambi i valori dei secondi e dei minuti. Sarebbe stato superfluo aggiungere un altro ingresso identico in quanto possiamo distinguere dove caricare i 5 bit sulla base di quale bottone viene premuto.

Di seguito riportiamo i componenti che formano l'entità `c_on_board` e il relativo port map:

```

component CRONOMETRO
  Port (
    CLK: in std_logic;
    RST: in std_logic;
    START: in std_logic;
    LOAD_s: in std_logic;
    LOAD_m: in std_logic;
    LOAD_h: in std_logic;
    SET_s: in std_logic_vector(0 to 5);
    SET_m: in std_logic_vector(0 to 5);
    SET_h: in std_logic_vector(0 to 4);
    u_sec: out std_logic_vector(0 to 5);
    u_min: out std_logic_vector(0 to 5);
    u_ore: out std_logic_vector(0 to 4)
  );
end component;

component seven_segment_array
  Generic(
    clock_frequency_in : integer := 50000000;
    clock_frequency_out : integer := 500
  );
  Port ( clock : in STD_LOGIC;
         reset : in STD_LOGIC;
         value32_in : in STD_LOGIC_VECTOR (23 downto 0);
         anodes : out STD_LOGIC_VECTOR (7 downto 0);
         cathodes : out STD_LOGIC_VECTOR (6 downto 0));
end component;

component B_DB
  generic (
    CLK_period: integer := 10; -- periodo del clock in nanosec
    btn_noise_time: integer := 10000000 --durata dell'oscillazione in nanosec
  );
  Port ( RST : in STD_LOGIC;
         CLK : in STD_LOGIC;
         BTN : in STD_LOGIC;
         CLEARED_BTN : out STD_LOGIC);
end component;

begin
  begin
    cron_o: CRONOMETRO port map(
      CLK=> CLK,
      RST=> br,
      START => '1',
      LOAD_s => bsec,
      LOAD_m => bm,
      LOAD_h => bh,
      SET_s => switch_set_sec_m,
      SET_m => switch_set_sec_m,
      SET_h => switch_set_ore,
      u_sec => s_c,
      u_min => m_c,
      u_ore => h_c
    );
    t_c<= '0' & '0' & '0' & h_c & '0' & '0' & m_c & '0' & '0' & s_c;
    dsp: seven_segment_array port map(
      clock => CLK,
      reset =>br,
      value32_in =>t_c,
      anodes =>anodes,
      cathodes =>cathodes
    );
  end;

```

```

bott_reset: B_DB port map(
    RST => '0',
    CLK => CLK,
    BTN =>b_RST,
    CLEARED_BTN => br
);

bott_secondi: B_DB port map(
    RST => '0',
    CLK => CLK,
    BTN =>b_LOAD_s,
    CLEARED_BTN => bsec
);
bott_minuti: B_DB port map(
    RST => '0',
    CLK => CLK,
    BTN =>b_LOAD_m,
    CLEARED_BTN => bm
);
bott_ore: B_DB port map(
    RST => '0',
    CLK => CLK,
    BTN =>b_LOAD_h,
    CLEARED_BTN => bh
);
end Structural;

```

5.3: SOLUZIONE

Per prendere l'intertempo occorre un registro in grado di memorizzare il valore assunto dal conteggio in quel determinato momento. Per memorizzare N intertempi useremo un banco di registri a scorrimento. È conveniente, in questo caso, usare gli shift register perché non dobbiamo usare un indirizzo, mentre se volessimo usare la memoria dovremmo indirizzarla.

Notiamo che, il registro avendo un ordine di memorizzazione tipicamente sequenziale, nel caso volessi visualizzare il 4° intertempo memorizzato dovrei prima visualizzare i primi tre in sequenza.

Dato l'orario da memorizzare espresso su 24 bit, per memorizzare N intertempi dobbiamo realizzare 24 registri da N bit ciascuno. Ogni registro memorizzerà un bit per un totale di 24 bit. Verrà quindi memorizzata l'intera stringa. I registri sono ad inserimento e uscita seriale. Se immaginiamo ogni registro disposto verticalmente, ogni volta che vogliamo salvare un nuovo intertempo, si effettuerà uno shift dall'alto verso il basso. In questo modo, i bit già memorizzati scorreranno verso il basso e i nuovi appena entrati si porranno in "alto" nel registro.

Per inserire gli intertempi abbiamo utilizzato un bottone, il quale, ogni volta che viene premuto, ci permette di effettuare lo shift e quindi l'inserimento in ogni shift register. Per la visualizzazione invece abbiamo usato un altro bottone che, ad ogni pressione, effettuerà lo shift sempre dall'alto verso il basso e ci permette di visualizzare il valore in uscita dai registri. Abbiamo supposto che, una volta visualizzato il valore dell'intertempo, esso viene perso. Se non volessimo perdere l'intertempo memorizzato, potremmo pensare di usare un registro a scorrimento circolare. In tal modo, una volta effettuato lo shift, il valore in uscita viene posto nuovamente in ingresso e non si perderà.

Modificheremo, quindi, l'entità `c_on_board` inserendo altri 2 bottoni, il banco di registri e un'unità di controllo per scegliere quale tra le due uscite (del cronometro o del banco di registri) far uscire.

Notiamo come il cronometro non viene in alcun modo modificato. Esso continua a lavorare, a contare normalmente. Con il bottone viene gestita la possibilità di far visualizzare l'intertempo per un certo numero di secondi arbitrari al posto del valore del cronometro.

5.3.1 Implementazione VHDL

5.3.1.1 Shift register

Partiremo col mostrare l'implementazione dello shift register, descritto in modo comportamentale:

```

entity shift_register is port(
    CLK, RST, X : in std_logic;
    E_in : in std_logic;
    E_visual : in std_logic;
    U : out std_logic
);
end shift_register;

architecture Behavioral of shift_register is
signal tmp: std_logic_vector(0 to 3);
signal c: std_logic;
begin
begin
    process(CLK)
    begin
        if (falling_edge(CLK)) then
            if (RST='1') then
                tmp <= (others=>'0');
            else if(E_in ='1') then --caricamento in shift register
                tmp(0) <= X;
                tmp(1) <= tmp(0);
                tmp(2) <= tmp(1);
                tmp(3) <= tmp(2);
            end if;
            if(E_visual='1') then
                tmp(0) <= '0';
                tmp(1) <= tmp(0);
                tmp(2) <= tmp(1);
                tmp(3) <= tmp(2);
                c<=tmp(3);
            end if;
        end if;
    end if;
    end process;
    U <= c;
end Behavioral;

```

5.3.1.2 Banco di registri

Come abbiamo già detto, necessitiamo di un banco di registri. Costruiremo quest'ultimo secondo un approccio strutturale. Essendo 24 registri riporteremo solo il port map del primo e degli ultimi tre registri essendo tutti pressoché identici. L'unico cambiamento è relativo al bit, tra i 24, da inserire nel rispettivo registro e quello da porre in uscita:

```

entity banco_reg is  Port (
  CLK : in std_logic;
  RST: in std_logic;
  value24_in : in std_logic_vector(23 downto 0);
  E_in: in std_logic;
  E_visual: in std_logic;
  value24_out : out std_logic_vector(23 downto 0)
);
end banco_reg;

architecture Structural of banco_reg is
component shift_register port(
  CLK, RST, X : in std_logic;
  E_in : in std_logic;
  E_visual : in std_logic;
  U : out std_logic
);
end component;

begin
sh_0: shift_register port map(
  CLK => CLK,
  RST =>RST,
  X=>value24_in(23),
  E_in =>E_in,
  E_visual =>E_visual,
  U=>value24_out(23)
);

sh_21: shift_register port map(
  CLK => CLK,
  RST =>RST,
  X=>value24_in(2),
  E_in =>E_in,
  E_visual =>E_visual,
  U=>value24_out(2)
);

sh_22: shift_register port map(
  CLK => CLK,
  RST =>RST,
  X=>value24_in(1),
  E_in =>E_in,
  E_visual =>E_visual,
  U=>value24_out(1)
);

sh_23: shift_register port map(
  CLK => CLK,
  RST =>RST,
  X=>value24_in(0),
  E_in =>E_in,
  E_visual =>E_visual,
  U=>value24_out(0)
);
end Structural;

```

5.3.1.3 Controllo

L'entità controllo dispone di due ingressi: quello proveniente dal cronometro e quello dal banco di registri. Il suo compito è decidere quale di questi due input porre in uscita. Generalmente viene posto in uscita il valore corrente del cronometro, ma nel caso il bottone per visualizzare l'intertempo venga premuto l'uscita cambierà e il display mostrerà il valore in uscita dal banco.

A questo punto sorge un problema, ovvero per quanto tempo visualizzare l'intertempo per poi ritornare a visualizzare il valore del cronometro. Abbiamo deciso quindi di far uso di un divisore di frequenza. Potevamo usare quello con una frequenza di uscita di un Hz, ma una volta premuto il bottone era effettivamente troppo poco il tempo per verificare che il risultato in uscita fosse quello che ci aspettavamo. Per visualizzare per più tempo l'intertempo abbiamo usato un divisore di frequenza che ci permetesse di visualizzarlo per certo numero di secondi. Per realizzare questo obiettivo abbiamo leggermente modificato il divisore di frequenza di 1 Hz moltiplicando il valore della variabile count_max (che ci indica per quanti conteggi il segnale deve rimanere basso per poi alzarsi) per un numero arbitrario. Se moltiplichiamo questo valore per 5, otterremo un segnale che rimane basso per all'incirca 5 secondi. In questo modo, una volta premuto il bottone visualizzeremo il valore memorizzato nel banco per all'incirca un 4/5 secondi. In realtà, più precisamente, il tempo di visualizzazione dipende da quando premiamo il bottone e, quindi, in base a quanto tempo ci rimane prima che il divisore si alzi.

Riportiamo l'implementazione di seguito:

```

entity controllo is  Port (
    CLK:  in std_logic;
    RST:  in std_logic;
    value_cron: in std_logic_vector(23 downto 0);
    value_shift: in std_logic_vector(23 downto 0);
    E_visual: in std_logic;
    d: in std_logic;
    value_out : out std_logic_vector(23 downto 0)
);
end controllo;

architecture Behavioral of controllo is

signal y: std_logic_vector(23 downto 0);
signal c: std_logic;

begin
process(CLK)
begin
    if(falling_edge(CLK)) then
        if(E_visual='1') then
            c<='1';
        else
            if(c='1' and d='0') then
                y<=value_shift;
            else
                y<=value_cron;
                c<='0';
            end if;
        end if;
    end if;
end process;
value_out<=y;
end Behavioral;

```

Analizziamo più approfonditamente il codice.

Sul fronte di discesa del clock, verifichiamo se il bottone per visualizzare l'intertempo sia stato, oppure no, premuto. Nel caso di pressione allora alziamo un signal c. Metteremo questo signal in and con il divisore d. In questo modo per tutto il tempo in cui il divisore rimane basso noi

visualizzeremo l'intertempo. Non appena il “d” si alza allora porremo in uscita il cronometro e abbasseremo il segnale “c”.

5.3.1.3.1: Osservazione

Il controllo sopra descritto si poteva realizzare in modo più semplice. Potevamo usare 2 bottoni, uno per visualizzare l'intertempo e uno per il valore di conteggio del cronometro.

A questo punto si effettuava un controllo, ovvero: se l'abilitazione per visualizzare l'intertempo (proveniente da un bottone) fosse stata alta allora avremmo cambiato l'uscita in modo che nel display potessimo visualizzare l'intertempo. Quando l'abilitazione per visualizzare l'orario era alta (sempre dovuta alla pressione di un bottone) cambiavamo l'uscita e sul display veniva rappresentato il tempo del cronometro. In questo modo potevamo gestire tranquillamente per quanto tempo visualizzare un determinato parametro. L'uscita non cambiava finché non eravamo noi a sceglierlo attraverso la pressione di un bottone.

Data la disponibilità di soli 5 bottoni, e avendoli usati tutti, abbiamo quindi provato ad usare un'altra strada per risolvere il problema della visualizzazione.

5.3.1.4: Implementazione c_on_board

Riportiamo l'implementazione di questo componente dopo aver aggiunto, a quello descritto precedentemente, i pezzi necessari per realizzare quanto richiesto:

```
entity c_on_board is
  Port (
    CLK: in std_logic;
    b_RST: in std_logic;
    START: in std_logic;
    b_LOAD_s: in std_logic;
    b_LOAD_m: in std_logic;
    b_LOAD_h: in std_logic;
    b_inser_sr: in std_logic;
    b_visual_sr: in std_logic;
    switch_set_sec_m: in std_logic_vector(0 to 5);
    switch_set_ore: in std_logic_vector(0 to 4);
    anodes : out STD_LOGIC_VECTOR (7 downto 0);
    cathodes : out STD_LOGIC_VECTOR (6 downto 0)
  );
end c_on_board;

architecture Structural of c_on_board is

signal s_c: std_logic_vector(0 to 5);
signal m_c: std_logic_vector(0 to 5);
signal h_c: std_logic_vector(0 to 4);
signal t_c: std_logic_vector(23 downto 0);
signal a: std_logic;
signal u_alternativa: std_logic_vector(23 downto 0);
signal u_def: std_logic_vector(23 downto 0);
signal d: std_logic;
```

```

--bottom
signal br: std_logic;
signal bsec: std_logic;
signal bm: std_logic;
signal bh: std_logic;
signal b_i_sr: std_logic;
signal b_v_sr: std_logic;

component CRONOMETRO
    Port (
        CLK: in std_logic;
        RST: in std_logic;
        START: in std_logic;
        LOAD_s: in std_logic;
        LOAD_m: in std_logic;
        LOAD_h: in std_logic;
        SET_s: in std_logic_vector(0 to 5);
        SET_m: in std_logic_vector(0 to 5);
        SET_h: in std_logic_vector(0 to 4);
        u_sec: out std_logic_vector(0 to 5);
        u_min: out std_logic_vector(0 to 5);
        u_ore: out std_logic_vector(0 to 4)
    );
end component;

component seven_segment_array
    Generic(
        clock_frequency_in : integer := 50000000;
        clock_frequency_out : integer := 500
    );
    Port ( clock : in STD_LOGIC;
            reset : in STD_LOGIC;
            value32_in : in STD_LOGIC_VECTOR (23 downto 0);
            anodes : out STD_LOGIC_VECTOR (7 downto 0);
            cathodes : out STD_LOGIC_VECTOR (6 downto 0));
end component;

component B_DB
generic (
    CLK_period: integer := 10; -- periodo del clock in nanosec
    btn_noise_time: integer := 10000000 --durata dell'oscillazione in nanosec
    );
    Port ( RST : in STD_LOGIC;
            CLK : in STD_LOGIC;
            BTN : in STD_LOGIC;
            CLEARED_BTN : out STD_LOGIC);
end component;

component banco_reg is Port (
    CLK : in std_logic;
    RST: in std_logic;
    value24_in : in std_logic_vector(23 downto 0);
    E_in: in std_logic;
    E_visual: in std_logic;
    value24_out : out std_logic_vector(23 downto 0)
);
end component;

component div_p_2 is
    generic(
        clock_frequency_in : integer := 100000000;--100MHz
        clock_frequency_out : integer := 1
    );
    Port (
        clock_in : in STD_LOGIC;
        reset : in STD_LOGIC;
        clock_out : out STD_LOGIC
    );
end component;

component controllo is Port (
    CLK: in std_logic;
    RST: in std_logic;
    value_cron: in std_logic_vector(23 downto 0);
    value_shift: in std_logic_vector(23 downto 0);
    E_visual: in std_logic;
    d: in std_logic;
    value_out : out std_logic_vector(23 downto 0)
);
end component;

```

```

begin

cron_o: CRONOMETRO port map(
    CLK=> CLK,
    RST=> br,
    START => '1',
    LOAD_s => bsec,
    LOAD_m => bm,
    LOAD_h => bh,
    SET_s => switch_set_sec_m,
    SET_m => switch_set_sec_m,
    SET_h => switch_set_ore,
    u_sec => s_c,
    u_min => m_c,
    u_ore => h_c
);

t_c<= '0' & '0' & '0' & h_c & '0' & '0' & m_c & '0' & '0' & s_c;

b_r: banco_reg port map(
    CLK =>CLK,
    RST =>br,
    value24_in =>t_c,
    E_in => b_i_sr,
    E_visual => b_v_sr,
    value24_out => u_alternativa
);

div2: div_p_2 port map(
    clock_in =>CLK,
    reset =>br,
    clock_out =>d
);

control: controllo port map(
    CLK => CLK,
    RST => br,
    value_cron => t_c,
    value_shift => u_alternativa,
    E_visual => b_v_sr,
    d => d,
    value_out => u_def
);

dsp: seven_segment_array port map(
    clock => CLK,
    reset =>br,
    value32_in =>u_def,
    anodes =>anodes,
    cathodes =>cathodes
);

bott_reset: B_DB port map(
    RST => '0',
    CLK => CLK,
    BTN =>b_RST,
    CLEARED_BTN => br
);

```

```

bott_secondi: B_DB port map(
    RST => '0',
    CLK => CLK,
    BTN =>b_LOAD_s,
    CLEARED_BTN => bsec
);
bott_minuti: B_DB port map(
    RST => '0',
    CLK => CLK,
    BTN =>b_LOAD_m,
    CLEARED_BTN => bm
);
bott_ore: B_DB port map(
    RST => '0',
    CLK => CLK,
    BTN =>b_LOAD_h,
    CLEARED_BTN => bh
);
bott_inser_shift: B_DB port map(
    RST => '0',
    CLK => CLK,
    BTN =>b_inser_sr,
    CLEARED_BTN => b_i_sr
);
bott_visual_shift: B_DB port map(
    RST => '0',
    CLK => CLK,
    BTN =>b_visual_sr,
    CLEARED_BTN => b_v_sr
);
end Structural;

```

Come possiamo notare, rispetto all'implementazione precedente, l'uscita del cronometro non viene posta direttamente in ingresso al display ma passa per l'entità "controllo" che si preoccupa di dare al display l'ingresso desiderato da far visualizzare all'utente.

CAPITOLO 6: SISTEMA DI TESTING

Esercizio 6.1

Progettare, implementare in VHDL e verificare mediante simulazione un sistema in grado di testare in maniera automatica una macchina combinatoria M avente 4 ingressi e 3 uscite binarie sottoponendole N ingressi diversi (si considerino una macchina M e un numero di input N a scelta dello studente).

Gli N valori di input per il test devono essere letti da una ROM, in cui essi sono precaricati, in corrispondenza di un segnale read. Le N uscite fornite della macchina in corrispondenza di ciascuno degli input devono essere memorizzate in una memoria interna, che deve poter essere svuotata in qualsiasi momento in presenza di un segnale di reset.

Esercizio 6.2

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.

6.1: ROM, CONTATORE E MACCHINA COMBINATORIA

Per poter progettare il sistema richiesto siamo partite dalla realizzazione della ROM, che abbiamo precaricato con 8 valori da 4 bit. Con un approccio di tipo comportamentale abbiamo assegnato all'uscita, all'arrivo di un segnale di abilitazione "read", il valore dell'attuale cella di memoria puntata dal segnale ADDR. Per poter scorrere la memoria, è stato infatti necessario un contatore modulo 8, così che al variare del conteggio riportato, cambiasse anche la cella della ROM di riferimento.

```
) architecture behavioral of ROM is
begin
type rom_type_1 is array (0 to 7) of std_logic_vector(3 downto 0);
signal ROM : rom_type_1 := (
    b"0000",
    b"0001",
    b"0010",
    b"0011",
    b"0100",
    b"0101",
    b"0110",
    b"0111"
);
begin
process(CLK)
begin
    if rising_edge(CLK) then
        if(READ='1') then
            DATA <= ROM(conv_integer(ADDR));
        end if;
    end if;
end process;
```

Nel contatore è inoltre presente anche un segnale di abilitazione 'enable', in particolare quando questo è alto, si incrementa il valore del conteggio.

```

entity counter_mod8 is
  Port ( clock : in STD_LOGIC;
         reset : in STD_LOGIC;
         enable : in STD_LOGIC; --questo è l'enable del clock, insieme danno l'impulso di conteggio (sarebbe il READ della ROM)
         counter : out STD_LOGIC_VECTOR (2 downto 0));
end counter_mod8;

architecture Behavioral of counter_mod8 is

signal c : std_logic_vector (2 downto 0) := (others => '0');

begin
  counter <= c;

  counter_process: process(clock)
begin
  if (clock'event AND clock = '1') then
    if (reset = '1') then
      c <= (others => '0');
    else
      if (enable='1') then
        c <= std_logic_vector(unsigned(c) + 1);
      end if;
    end if;
  end if;
end process;

```

Il valore letto dalla ROM andava a questo punto caricato nella macchina combinatoria per poter essere elaborato.

La macchina combinatoria è stata pensata per produrre in uscita un valore su tre bit con significato differente:

- 1) Il primo bit (quello meno significativo) rappresenta se il numero è pari o dispari, in particolare riporta 1 in uscita se il bit è dispari e 0 se è pari
- 2) Il secondo bit, di uscita rappresenta se il numero di zeri è maggiore o uguale del numero degli 1 presenti nella stringa (se così fosse l'uscita sarebbe 1)
- 3) Il terzo e ultimo bit (quello più significativo) rappresenta se il numero degli 1 è maggiore o uguale del numero degli 0 presenti nella stringa

La logica adottata è visibile nel seguente codice:

```

| entity macchina_comb is
|   Port (
|     i: in std_logic_vector(3 downto 0);
|     u: out std_logic_vector(2 downto 0)
|   );
| end macchina_comb;

| architecture dataflow of macchina_comb is

begin
  u(0)<=i(0);
  u(1)<=(not(i(0)) and not i(1)) or (not(i(2)) and not i(3)) or (not i(0) and not i(2)) or (not i(0) and not i(3)) or
    (not i(3) and not i(1) and i(0)) or (not i(1) and not i(2)) or (not i(1) and not i(3) and i(2));
  u(2)<= (i(1) and i(3)) or (i(0) and i(3)) or (i(1) and i(2)) or (i(0) and i(2)) or (i(0) and i(1)) or (i(2) and i(3));

| end dataflow;

```

In particolare, per l'uscita $u(1)$ e $u(2)$ sono state sfruttate le mappe di Karnaugh (a titolo di esempio, una di esse è riportata in figura 6.1), per determinare quali dovessero essere le uscite al variare degli ingressi.

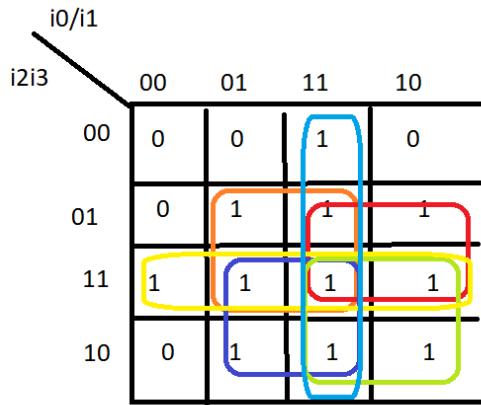


Figura 6.1: Mappa di karnaugh

6.1.1: MEMORIA

In fine, è stato realizzato l'ultimo componente, vale a dire la memoria. L'approccio scelto è stato quello di tipo comportamentale; in particolare sono stati utilizzati due segnali di abilitazione, uno per la lettura e uno per la scrittura. Quando il segnale di abilitazione alla scrittura è alto, vuol dire che stiamo fornendo il permesso di caricare in memoria il dato all'indirizzo indicato dal campo *Address_b*, quando invece il segnale di abilitazione alla lettura è alto, aggiorniamo l'uscita al nuovo valore puntato. In questo modo carichiamo e visualizziamo il dato solo quando richiesto.

```

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

entity mem_b is
port(
    clk: in std_logic;
    rst: in std_logic;
    S: in std_logic_vector(2 downto 0);
    Address_B: in std_logic_vector(2 downto 0);
    Am_w: in std_logic;
    Am_r: in std_logic;
    Y: out std_logic_vector(2 downto 0)

);
end mem_b;

```

```

architecture Behavioral of mem_b is

type RAM_ARRAY is array (0 to 7) of std_logic_vector (2 downto 0);

signal RAM: RAM_ARRAY;

begin
process(clk)
begin
if(rising_edge(clk)) then
  if(rst='1') then
    RAM <= (OTHERS => (OTHERS => '0'));
    Y<="000";
  else
    if(Am_w='1') then
      RAM(to_integer(unsigned(Address_B))) <= S;
    end if;
    if(Am_r='1') then
      Y<= RAM(to_integer(unsigned(Address_B)));
    end if;
  end if;
end if;
end process;

end Behavioral;

```

A scopo puramente informativo, e guidate dalla curiosità abbiamo provato anche a realizzare la memoria in maniera strutturale, utilizzando cascate di flip-flop per memorizzare i singoli bit. In particolare, abbiamo realizzato il componente di base “dfm”, che abbiamo poi usato per ciascun bit di uscita della macchina combinatoria, usando un demultiplexer, un multiplexer e 8 flip flop.

Ciascun bit in uscita della macchina combinatoria rappresenta l’ingresso di un dfm, e in particolare del demultiplexer 1:8 di cui esso è composto. A seconda del valore di abilitazione, fornito dal conteggio del contatore, si individua la linea di uscita del demultiplexer. Ciascuna di queste linee sono connesse a flip flop diversi, per un totale di 8 flip flop per dfm. Sulla base di un segnale di abilitazione *r_w* viene caricato e quindi posto in uscita il dato sul flip flop.

A questo punto, tutte queste uscite dei diversi flip flop convogliano in un multiplexer, che ha la stessa abilitazione del demux e mette così in uscita il dato.

In totale vengono utilizzati tre di questi blocchetti dfm, uno per ogni bit. Quindi in sostanza, all’arrivo dei tre dati di uscita della macchina combinatoria, i tre bit vengono indirizzati in tre blocchetti dfm differenti. Se il conteggio del contatore è pari a zero, e quindi vuol dire che stiamo lavorando sulla prima stringa della ROM, allora i singoli bit vengono memorizzati nei primi flip flop dei rispettivi blocchi dfm. Se il conteggio del contatore è pari a uno, e quindi vuol dire che stiamo lavorando sulla seconda stringa della ROM, allora i singoli bit vengono memorizzati nei secondi flip flop dei rispettivi blocchi dfm, e così via.

In figura 6.2 è riportata l’architettura di base per i tre bit.

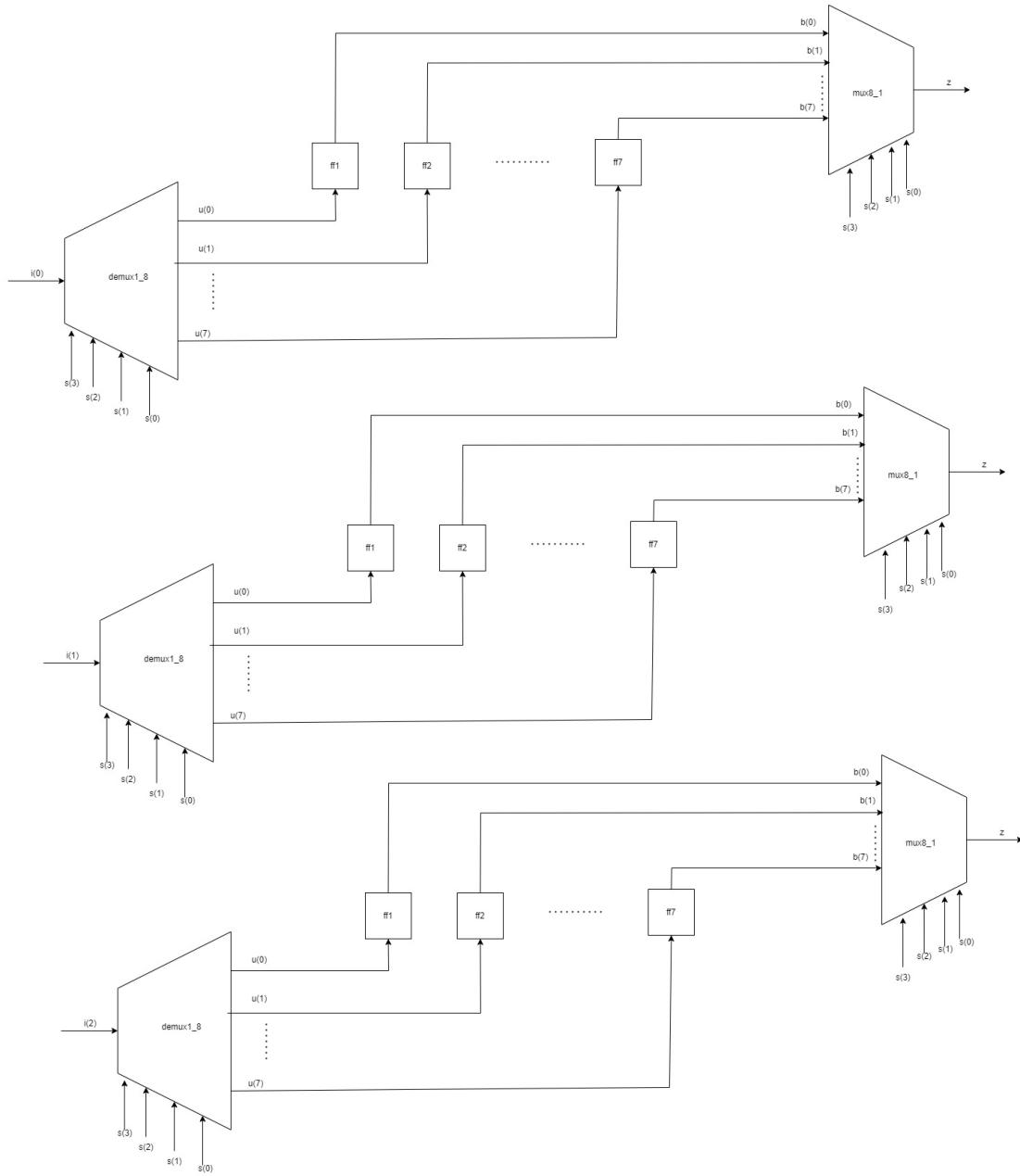


Figura 6.2: Architettura dei tre dfm

Di seguito l'implementazione in VHDL.

```

entity dfm is
  Port (
    CLK: in std_logic;
    RST: in std_logic;
    i: in std_logic;
    a: in std_logic_vector(2 downto 0);
    r_w: in std_logic;
    u: out std_logic
  );
end dfm;

architecture structural of dfm is

component demux_1_8 is
Port (
  ingr: in std_logic;
  ab: in std_logic_vector(2 downto 0);
  usc: out std_logic_vector(0 to 7)
);
end component;

component mux8_1 is
  port (
    x: in std_logic_vector(0 to 7);
    k: in std_logic_vector(2 downto 0);
    y: out std_logic
  );
end component;

component FlipFlop is port(
  D: in std_logic;
  CLK: in std_logic;
  RST: in std_logic;
  r_w: in std_logic;
  Q: out std_logic
);
end component;

signal d: std_logic_vector(0 to 7);
signal m: std_logic_vector(0 to 7);

begin

de: demux_1_8 port map(
  ingr=>i,
  ab=>a,
  usc=>d
);

ff0: FlipFlop port map(
  CLK=>CLK,
  RST=> RST,
  D=>d(0),
  r_w=>r_w,
  Q=>m(0)
);

```

```

ff1: FlipFlop port map(
    CLK=>CLK,
    RST=> RST,
    D=>d(1),
    r_w=>r_w,
    Q=>m(1)
);

ff2: FlipFlop port map(
    CLK=>CLK,
    RST=> RST,
    D=>d(2),
    r_w=>r_w,
    Q=>m(2)
);

ff3: FlipFlop port map(
    CLK=>CLK,
    RST=> RST,
    D=>d(3),
    r_w=>r_w,
    Q=>m(3)
);

ff4: FlipFlop port map(
    CLK=>CLK,
    RST=> RST,
    D=>d(4),
    r_w=>r_w,
    Q=>m(4)
);

ff5: FlipFlop port map(
    CLK=>CLK,
    RST=> RST,
    D=>d(5),
    r_w=>r_w,
    Q=>m(5)
);

ff6: FlipFlop port map(
    CLK=>CLK,
    RST=> RST,
    D=>d(6),
    r_w=>r_w,
    Q=>m(6)
);

ff7: FlipFlop port map(
    CLK=>CLK,
    RST=> RST,
    D=>d(7),
    r_w=>r_w,
    Q=>m(7)
);

mu: mux8_1 port map (
    x=>m,
    k=>a,
    y=>u
);
end structural;

```

6.1.2: RETE DI CONTROLLO

Per poter gestire tutti i segnali è stata poi elaborata una rete di controllo che consentisse, all'arrivo del segnale di abilitazione della lettura della ROM, di abilitare la scrittura nella memoria, e successivamente di leggere il valore in essa memorizzata. Solo alla fine di queste operazioni è stata alzata l'abilitazione del contatore, in modo da poter essere incrementato e da poter così puntare alla prossima locazione di memoria della ROM da dover leggere.

L'automa realizzato è visibile in figura 6.3.

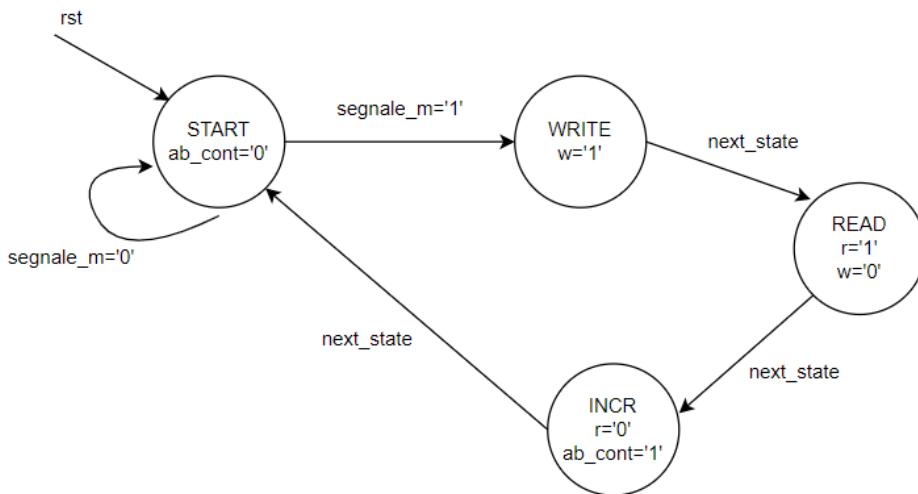


Figura 6.3: Automa della rete di controllo

Il codice VHDL per la sua realizzazione è riportato di seguito:

```

entity rete_controllo is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    segnale_m: in std_logic;
    r: out std_logic;
    w: out std_logic;
    ab_cont: out std_logic
  );
end rete_controllo;

architecture Behavioral of rete_controllo is
  type state is(
    START, WRITE, READ, INCR
  );
  signal current_state, next_state: state;
begin
  begin
    ag: process(clk)
    begin
      if(rising_edge(clk)) then
        if(rst='1') then
          current_state<=START;
        else
          current_state<=next_state;
        end if;
      end if;
    end process;
  end
  
```

```

c: process(current_state, segnale_m)
begin

    r<='0';
    w<='0';
    ab_cont<='0';

    case current_state is

        when START =>

            if(segnale_m='0')then
                next_state <=START;
            else
                if(segnale_m='1')then
                    next_state <= WRITE;
                end if;
            end if;

        when WRITE =>
            w<='1';

            next_state<=READ;

        when READ =>
            r<='1';

            next_state<=INCR;

        when INCR =>
            ab_cont<='1';

            next_state<=START;

    end case;
end process;

end Behavioral;

```

6.1.3: ARCHITETTURA

Per poter concludere il primo punto dell'esercizio, abbiamo in fine progettato un'entità, chiamato "blocco" che contenesse tutti i componenti e che, tramite un approccio strutturale, realizzasse la loro interconnessione, come visibile anche in figura 6.4:

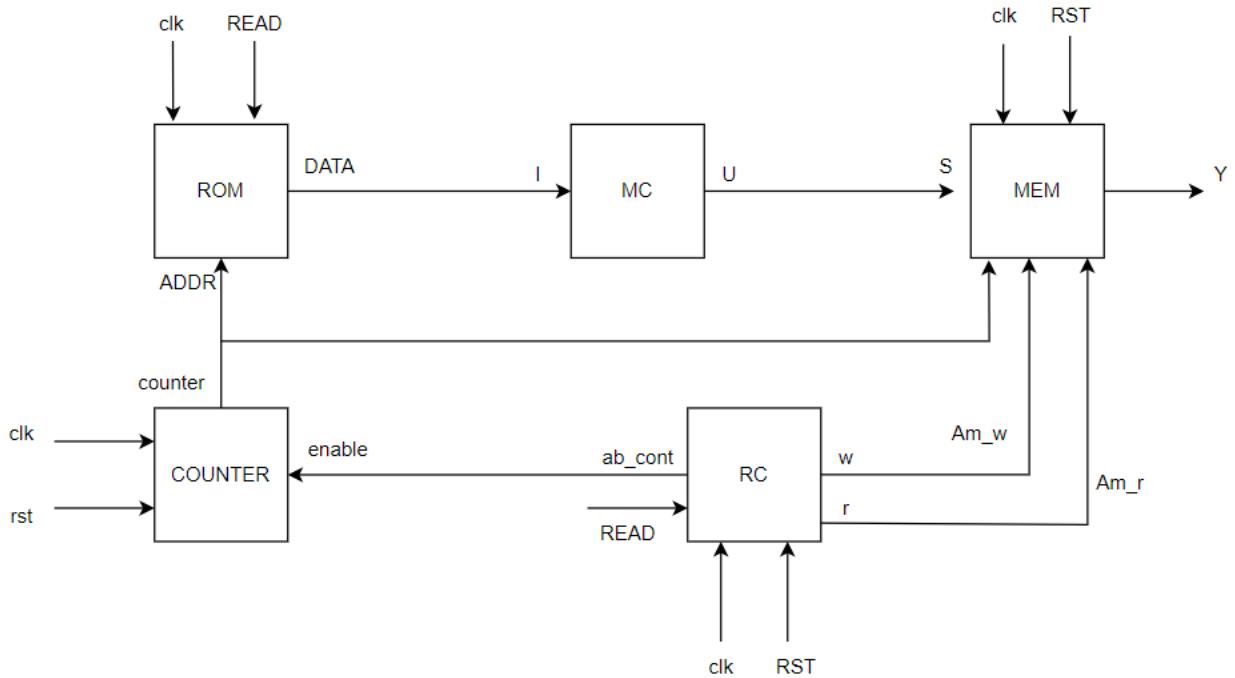


Figura 6.4: Architettura

Come si evince da questa figura il dato letto dalla ROM viene caricato nella macchina combinatoria, il quale lo elabora, producendo in uscita tre bit, che vengono caricati nella memoria sulla base di un segnale di abilitazione; il tutto è gestito dalla rete di controllo, che non solo fornisce i segnali di abilitazione alla memoria, ma anche al contatore, permettendo a quest'ultimo di scandire le varie locazioni sia della ROM (da cui deve prelevare il dato di ingresso) sia della memoria.

L'implementazione completa è riportata nel codice sottostante.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity blocco is
  Port (
    CLK: IN std_logic;
    RST: in std_logic;
    segnale_m : in std_logic;
    uu: out std_logic_vector(2 downto 0);
    uMC: out std_logic_vector(2 downto 0);

    count : out STD_LOGIC_VECTOR (2 downto 0)
  );
end blocco;

architecture Structural of blocco is

component counter_mod8 is
  Port (
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    enable : in STD_LOGIC;
    counter : out STD_LOGIC_VECTOR (2 downto 0));
end component;

component ROM is
port(
  CLK : in std_logic;
  READ : in std_logic;
  ADDR : in std_logic_vector(2 downto 0);
  DATA : out std_logic_vector(3 downto 0)
);
end component;

component macchina_comb is
  Port (
    i: in std_logic_vector(3 downto 0);
    u: out std_logic_vector(2 downto 0)
  );
end component;

component mem_b is
port(
  clk: in std_logic;
  rst: in std_logic;
  S: in std_logic_vector(2 downto 0);
  Address_B: in std_logic_vector(2 downto 0);
  Am_w: in std_logic;
  Am_r: in std_logic;
  Y: out std_logic_vector(2 downto 0)
);
end component;

```

```

component rete_controllo is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    segnale_m: in std_logic;
    r: out std_logic;
    w: out std_logic;
    ab_cont: out std_logic
  );
end component;

signal s: std_logic_vector(2 downto 0);
signal u_r: std_logic_vector(3 downto 0);
signal u_m: std_logic_vector(2 downto 0);
signal ab_c: std_logic;
signal wr: std_logic;
signal rd: std_logic;

begin
  cont: counter_mod8 port map(
    clock=>CLK,
    reset=>rst,
    enable=>ab_c,
    counter=>s
  );
  ro: ROM port map(
    CLK =>CLK,
    READ=>segnale_m,
    ADDR => s,
    DATA =>u_r
  );
  m_c: macchina_comb Port map (
    i=>u_r,
    u=>u_m
  );
  uMC<=u_m;
  count<=s;
  mem: mem_b port map(
    clk=>CLK,
    rst=>rst,
    S=>u_m,
    Address_B=>s,
    Am_w=>wr,
    Am_r=>rd,
    Y=>uu
  );
  r_c: rete_controllo Port map(
    clk=>clk,
    rst=>rst,
    segnale_m=>segnale_m,
    r=>rd,
    w=>wr,
    ab_cont=>ab_c
  );
end structural;

```

6.1.4: SIMULAZIONE

Per testare l'entità così progettata, abbiamo realizzato un testbench.

```
proc: process
begin

rst<='1';
wait for 100ns;
rst<='0';

sig<='1';
wait for 10ns;
sig<='0';
wait for 30ns;

sig<='1';
wait for 10ns;
sig<='0';
wait for 30ns;

sig<='1';
wait for 10ns;
sig<='0';
wait for 30ns;

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

sig<='1';
wait for 10ns;
sig<='0';
wait for 30ns;
```

Come si può osservare dal codice riportato, abbiamo semplicemente fornito il segnale di abilitazione alla lettura della ROM (chiamato qui *sig*) consentendo in questo modo di far evolvere la macchina. Abbiamo poi effettuato un reset per analizzarne il comportamento anche in presenza di questo segnale, e come si può osservare dalla simulazione in figura 6.5, tutto viene correttamente resettato: il contatore ricomincia a contare da zero, la ROM di conseguenza preleverà e porterà in uscita, e quindi in ingresso alla macchina combinatoria (sempre all'arrivo del segnale *sig*) il valore contenuto nella prima cella. La macchina combinatoria, d'altro canto, manterrà il vecchio valore, e solo all'arrivo del nuovo segnale aggiornerà il suo valore di uscita.

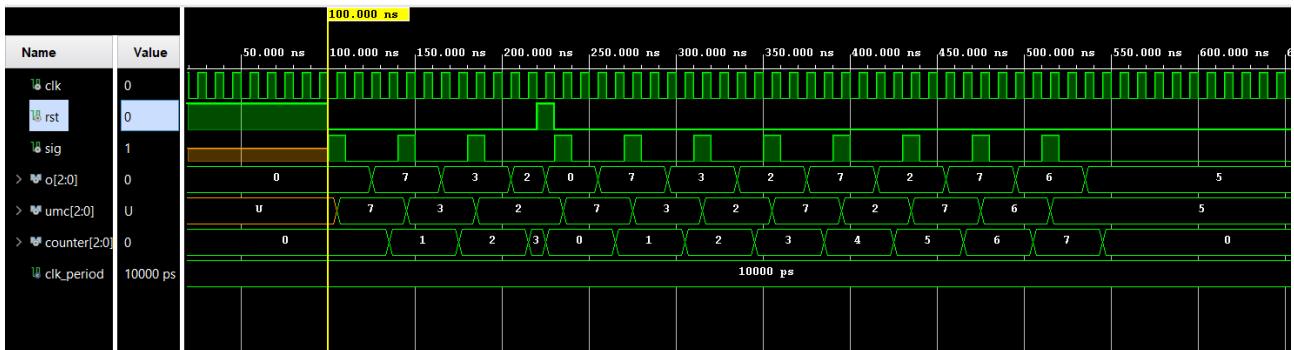


Figura 6.5: Testbench

6.2: IMPLEMENTAZIONE SU BOARD

Il secondo punto dell'esercizio chiedeva di sintetizzare ed implementare su board il componente sviluppato. Per poter realizzare la sintesi su board è stato fondamentale integrare nell'architettura due bottoni così da gestire gli ingressi di *read*, per la lettura dalla ROM, e *reset*.

```
entity scheda is
  Port (
    CLK: IN std_logic;
    b_RST: in std_logic;
    b_read: in std_logic;
    uMC: out std_logic_vector(2 downto 0)
  );
end scheda;

architecture structural of scheda is

component blocco is
  Port (
    CLK: IN std_logic;
    RST: in std_logic;
    segnale_m : in std_logic;
    uu: out std_logic_vector(2 downto 0);
    uMC: out std_logic_vector(2 downto 0);

    count : out STD_LOGIC_VECTOR (2 downto 0)
  );
end component;
```

```

component B_DB is
  generic (
    CLK_period: integer := 10;
    btn_noise_time: integer := 10000000
  );
  Port ( RST : in STD_LOGIC;
          CLK : in STD_LOGIC;
          BTN : in STD_LOGIC;
          CLEARED_BTN : out STD_LOGIC);
end component;

signal mem: std_logic_vector(2 downto 0);
signal r: std_logic;
signal seg: std_logic;

begin

b: blocco port map(
  CLK=>CLK,
  RST=> r,
  segnale_m=>seg,
  uu=> mem,
  uMC=>uMC
);

de: B_DB port map(
  RST => '0',
  CLK =>CLK,
  BTN=>b_RST,
  CLEARED_BTN => r
);

de2: B_DB port map(
  RST => '0',
  CLK =>CLK,
  BTN=>b_read,
  CLEARED_BTN => seg
);

end structural;

```

Affinché l'uscita potesse essere letta sui led, abbiamo poi legato l'uscita della macchina ai led nel file constraints, in maniera analoga a come fatto con i bottoni.

CAPITOLO 7: COMUNICAZIONE CON HANDSHAKING

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

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

7.1: SOLUZIONE

Rappresentiamo i due nodi nella figura sottostante per avere un'idea di come è articolata la struttura globale di questo sistema. Successivamente approfondiremo i nodi A e B nel dettaglio.

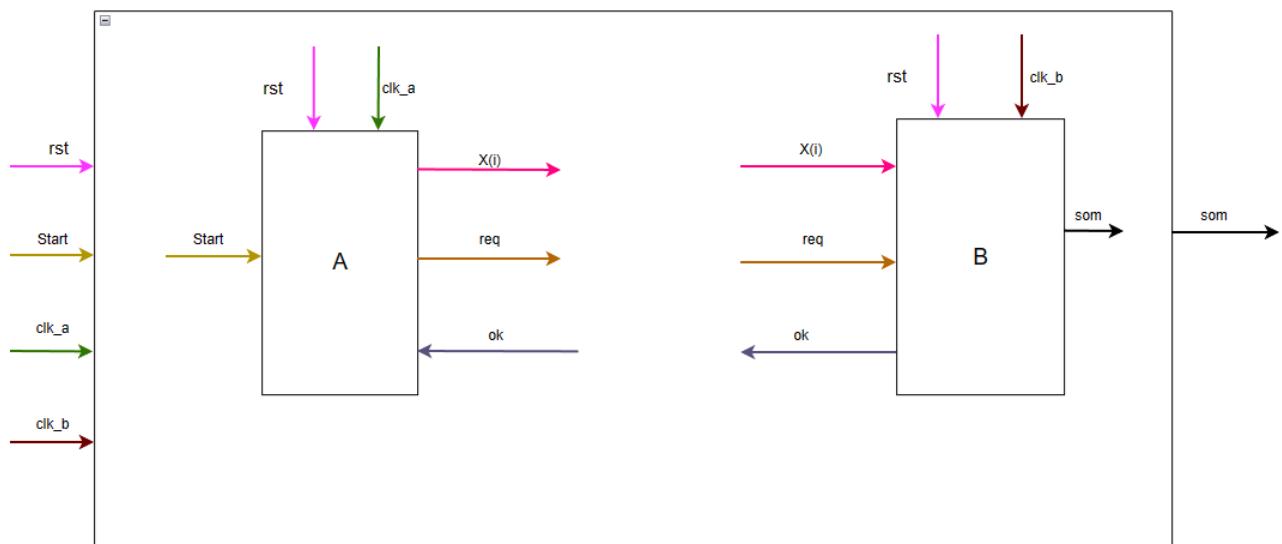


Figura 7.1: Schema globale_AB

Il protocollo di handshaking è un protocollo asincrono di comunicazione. Non è quindi necessario che le due entità che vogliono comunicare siano isocrone.

Abbiamo deciso di usare un protocollo di handshaking semplice. Il nodo A manda il dato ed un segnale di request ("req") per manifestare la sua volontà di comunicare. Il nodo B, invece, una volta che riceve la richiesta, fa le sue operazioni e, quando termina, invia ad A un segnale "ok".

Vediamo ora come sono formati internamente i blocchi A e B.

7.1.1: A

Il nodo A, oltre al segnale di clock e di reset, prevede come ingresso un segnale di start per iniziare la sua elaborazione. Il componente è formato da una parte operativa e una parte di controllo.

Nella parte operativa necessitiamo di un componente di memoria, in cui sono state precaricate un certo numero (4) di stringhe di 8 bit ciascuna, e di un contatore per scorrere gli indirizzi della memoria e che, quindi, ci indica quando A ha finito di inviare le N stringhe.

La parte di controllo invece è stata realizzata attraverso un automa. È il blocco che ha il compito di controllare la PO.

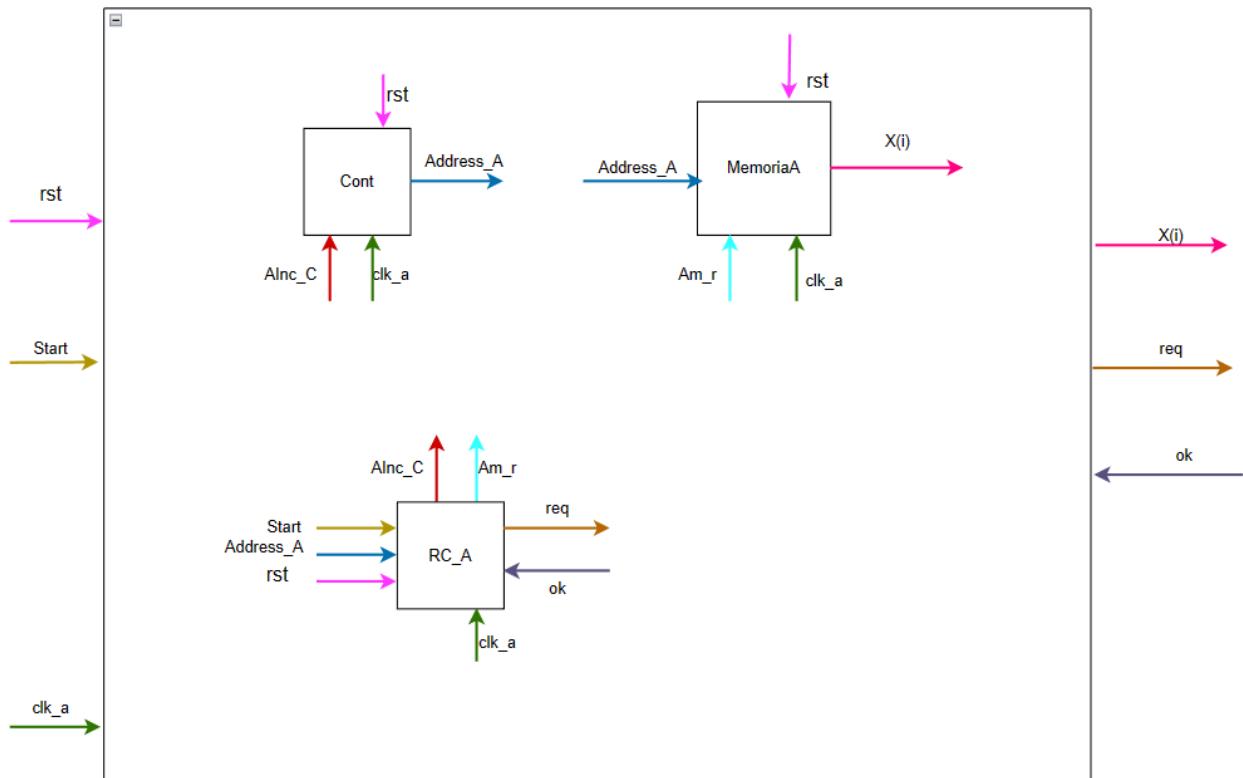


Figura 7.2: A

Tutti i componenti hanno come ingresso il segnale di clock e di reset.

La rete di controllo presenta i seguenti segnali di ingresso: start (per dare inizio alle operazioni), Address_A (uscita del contatore che serve alla rete di controllo per capire quando A ha finito, quando tutte le stringhe che doveva inviare sono state spedite), ok (quando il nodo B comunica di aver finito le sue operazioni, A può procedere con l'invio della stringa successiva).

I segnali in uscita dalla rete di controllo sono: req (richiesta di voler comunicare), Ainc_C (abilitazione per l'incremento del conteggio del contatore), Am_r (abilitazione della memoria a leggere e, quindi, ad aggiornare il suo valore di uscita a quello indicato dall'Address_A).

Riportiamo, nella figura 7.3, l'automa che realizza il blocco RC_A.

Come possiamo notare, abbiamo un primo stato di IDLE in cui il sistema attende fino all'arrivo del segnale di start, per dare inizio a tutte le operazioni. Quando start è pari a 1, passiamo nello stato di READ, in cui alziamo l'abilitazione della lettura dalla memoria. Al successivo colpo di clock, la memoria di A vede l'abilitazione a 1 e pone in uscita il dato alla posizione specificata dall'Address. Nello stato di SEND mandiamo la richiesta e aspettiamo di ricevere il segnale di "ok" da B.

Una volta ricevuto ok, A deve attendere che l'ok si abbassi. Questa operazione risulta necessaria perché se A andasse più veloce di B procederebbe negli stati di INC, READ, e in SEND. In quest'ultimo stato invierebbe una nuova richiesta e vedendo l'ok alto proseguirebbe con l'ok precedente, non aspettando, effettivamente, il "nuovo" ok di B, che dovrebbe comunicare la fine dell'operazione sul nuovo dato inviato da A. Per questo motivo inseriamo uno stato di CHECK in cui aspettiamo che il segnale di ok si abbassi.

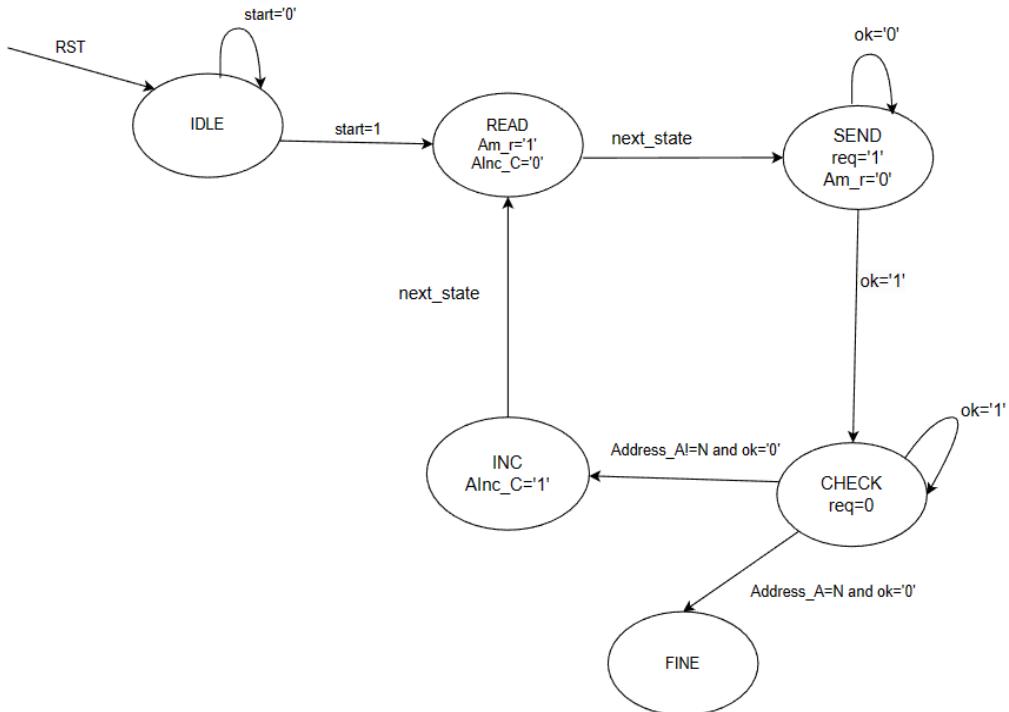


Figura 7.3: Automa_RC_A

Dopo lo stato di CHECK, abbiamo lo stato INC che alza l'abilitazione del contatore. Il contatore, al successivo colpo di clock, vede l'abilitazione alta e incrementa di uno il valore del suo conteggio.

A finisce il suo lavoro dopo aver inviato tutte le stringhe. Il contatore tiene traccia del numero di valori inviati e quando il suo conteggio arriva ad N possiamo transitare nello stato di FINE. In caso contrario procediamo con lo stato INC.

La scelta di andare in uno stato di FINE è puramente progettuale. In questo modo possiamo ritornare nello stato di IDLE solo con il segnale di reset. In modo alternativo, dallo stato di CHECK, verificato che il valore di Address_A sia pari a N, si proseguiva nello stato di IDLE e ci si poneva in attesa del segnale di start.

7.1.2: B

Riportiamo, in figura 7.4, la rappresentazione del nodo B con tutti i suoi componenti.

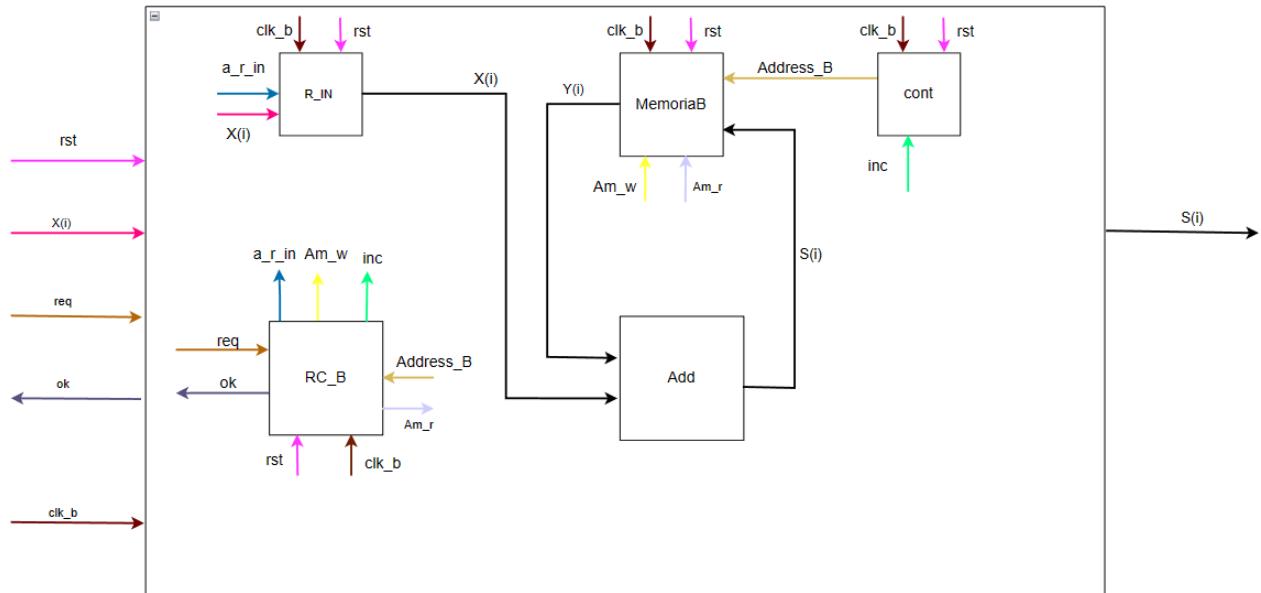


Figura 7.4: B

Il nodo B si avvale di un registro, per memorizzare la stringa i -esima ricevuta da A, di una memoria, di un blocco contatore, di un addizionatore e della rete di controllo.

La memoria di B contiene le stringhe $Y(i)$, che andranno poi a sommarsi alle rispettive stringhe $X(i)$, provenienti da A. Inoltre, in essa andranno memorizzate i risultati della somma.

Abbiamo deciso di memorizzare le somme in locazioni successive a quelle in cui sono memorizzate le stringhe $Y(i)$. Per fare ciò ci siamo avvalsi di un unico contatore (modulo $N=4$) e, nella memoria, abbiamo usato come indirizzo in cui memorizzare, scrivere, i valori $S(i)$, quello uscente dal contatore ($Address_B$) a cui aggiungiamo il valore N . In questo modo, data la somma delle stringhe $X(i)$ e $Y(i)$, poste all'indirizzo zero, andiamo a memorizzare il valore uscente dall'addizionatore alla locazione $0+N$, che corrisponde alla prima posizione libera nella memoria. Proseguiamo in questo modo anche per le stringhe successive. Le somme saranno memorizzate in locazioni libere e le stringhe $Y(i)$ non saranno sovrascritte. Notiamo che la somma è una stringa di 8 bit e che, quindi, non abbiamo considerato il valore del riporto finale.

La rete di controllo prevede in ingresso il segnale req , proveniente da A, e l' $Address_B$. Come segnali di uscita possiede: a_r_in (abilitazione del registro a memorizzare), Am_w (abilitazione alla memoria a scrivere il valore della somma nelle sue locazioni), Am_r (abilitazione alla lettura della memoria) e inc (abilitazione per l'incremento del contatore).

Tutti i componenti presentano in ingresso i segnali di clock e reset.

L'automa della rete di controllo di B è riportato in figura 7.5.

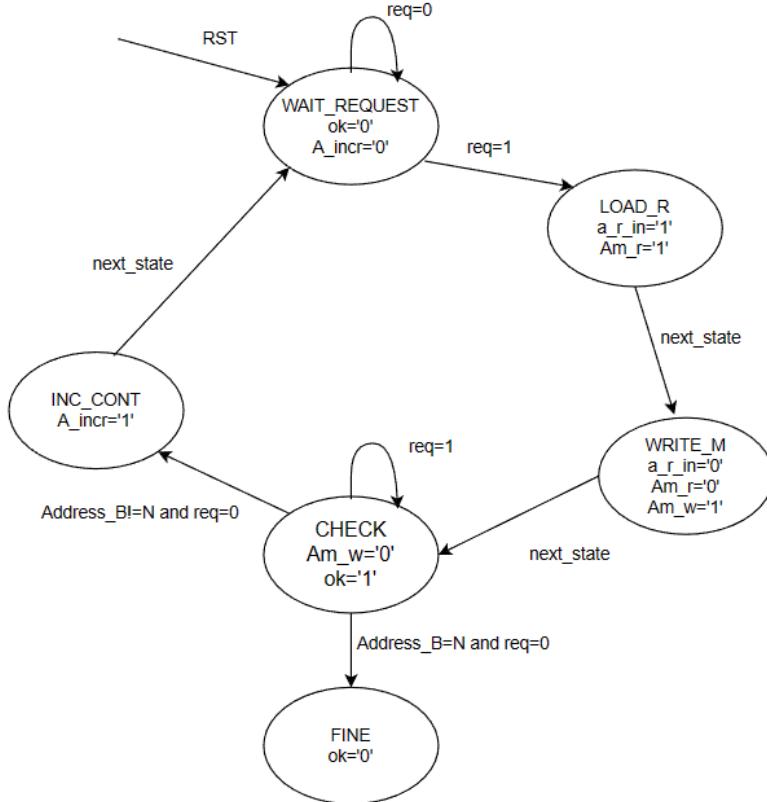


Figura 7.5: Automa_RC_B

Il nodo B attende, nello stato WAIT_REQUEST, il segnale di req da parte di A. Una volta ricevuto prosegue nello stato LOAD_R. In questo stato viene posto a 1 sia l'abilitazione del registro a memorizzare il valore ricevuto da A e sia l'abilitazione di lettura della memoria di B. Al successivo colpo di clock avrà in uscita dal registro, X(i), e in uscita dalla memoria, il valore Y(i). Queste due stringhe andranno in ingresso al sommatore. Lo stato WRITE_M abilita la scrittura nella memoria di B e pone come next_state CHECK. Lo stato di CHECK aspetta che la richiesta req si abbassi. Se non venisse effettuato questo passaggio, dato che il nodo B può essere più veloce del nodo A, si rischierebbe di ritornare nello stato di WAIT_REQUEST, con il segnale di req ancora alto e, quindi, proseguire tra gli stati con la precedente richiesta di A.

Dallo stato di CHECK si prosegue o nello stato di FINE, quando la richiesta è stata abbassata e il contatore ha raggiunto il valore N, oppure, in caso contrario, nello stato INC_CONT. In questo stato viene alzata l'abilitazione al contatore di B. Da questo si ritorna in WAIT_REQUEST.

7.2: IMPLEMENTAZIONE VHDL

Iniziamo col mostrare prima l'implementazione del nodo A, poi del nodo B ed infine del componente che li racchiude entrambi.

7.2.1: A

7.2.1.1: Contatore

```
entity cont is
    generic (
        I: integer :=2
    );
    Port ( clk : in STD_LOGIC;
            rst : in STD_LOGIC;
            inc : in STD_LOGIC; --questo è l'enable del clock, insieme danno l'impulso di conteggio
            Address : out STD_LOGIC_VECTOR ((I-1) downto 0));
end cont;

architecture Behavioral of cont is

signal c : std_logic_vector ((I-1) downto 0) := (others => '0');

begin

Address <= c;

counter_process: process(clk)
begin
    if (clk'event AND clk = '1') then
        if (rst = '1') then
            c <= (others => '0');
        else
            if (inc='1') then
                c <= std_logic_vector(unsigned(c) + 1);
            end if;
        end if;
    end if;
end process;

end Behavioral;
```

Avendo deciso di memorizzare un numero N=4 stringhe , il contatore di A sarà un generico contatore modulo 4.

7.2.1.2: Memoria

```
entity memoriaA is
    generic(
        I: integer :=2;
        N : integer:= 4;--n di stringhe
        M : integer:=8--lunghe stringa
    );
    port(
        CLK_A: in std_logic;
        RST: in std_logic;
        Am_r: in std_logic;
        Address_A: in STD_LOGIC_VECTOR((I-1) downto 0); -- Address to write/read RAM
        X1: out STD_LOGIC_VECTOR((M-1) downto 0) -- Data output of RAM
    );
end memoriaA;
```

```

architecture Behavioral of memoriaA is

type RAM_ARRAY is array (0 to (N-1)) of std_logic_vector ((M-1) downto 0);

signal RAM: RAM_ARRAY :=(
  b"00000001",
  b"00000010",
  b"10000100",
  b"00001000"
);
begin
process(CLK_A)
begin
  if(rising_edge(CLK_A)) then
    if(Am_r='1') then
      X1 <= RAM(to_integer(unsigned(Address_A)));
    end if;
  end if;
end process;

end Behavioral;

```

7.2.1.3: RC_A

```

entity RC_A is
  generic (
    I: integer :=2
  );
  Port (
    clk_a: in std_logic;
    rst: in std_logic;
    start: in std_logic;
    Address_A: in std_logic_vector((I-1) downto 0);
    ok: in std_logic;
    AInc_C: out std_logic;
    Am_r: out std_logic;
    req: out std_logic
  );
end RC_A;

```

```

architecture Behavioral of RC_A is
type state is(
  IDLE, READ, SEND,CHECK,INC,FINE
);
signal current_state, next_state: state;

begin
ag: process(clk_a)
begin
  if(rising_edge(clk_a)) then
    if(rst='1') then
      current_state<=IDLE;
    else
      current_state<=next_state;
    end if;
  end if;
end process;

```

```

c: process(current_state, start, ok, Address_A)
begin

    case current_state is

        when IDLE =>
            AIInc_C <='0';
            Am_r<='0';
            req<='0';

            if(start='0')then
                next_state <=IDLE;
            else
                if(start='1') then
                    next_state <= READ;
                end if;
            end if;

        when READ =>
            AIInc_C <='0';
            Am_r<='1';
            req<='0';

            next_state <=SEND;

        when SEND =>
            AIInc_C <='0';
            Am_r<='0';
            req<='1';

            if(ok='0') then
                next_state<=SEND;
            else
                next_state<=CHECK;
            end if;

        when CHECK =>
            AIInc_C <='0';
            Am_r<='0';
            req<='0';

            if(ok='1') then
                next_state<=CHECK;
            else
                if(ok='0' and Address_A=="11") then
                    next_state<=FINE;
                else
                    if(ok='0' and Address_A!="11") then
                        next_state<=INC;
                    end if;
                end if;
            end if;
    end case;
end process;

```

```

when READ =>
    AIInc_C <='0';
    Am_r<='1';
    req<='0';

    next_state <=SEND;

when SEND =>
    AIInc_C <='0';
    Am_r<='0';
    req<='1';

    if(ok='0') then
        next_state<=SEND;
    else
        next_state<=CHECK;
    end if;

when CHECK =>
    AIInc_C <='0';
    Am_r<='0';
    req<='0';

    if(ok='1') then
        next_state<=CHECK;
    else
        if(ok='0' and Address_A=="11") then
            next_state<=FINE;
        else
            if(ok='0' and Address_A!="11") then
                next_state<=INC;
            end if;
        end if;
    end if;

when INC =>
    AIInc_C <='1';
    Am_r<='0';
    req<='0';

    next_state<=READ;

when FINE =>
    AIInc_C <='0';
    Am_r<='0';
    req<='0';

end case;
end process;

end Behavioral;

```

La rete di controllo di A rispetta a pieno quanto descritto in figura 7.3 nel paragrafo 7.1.1.

7.2.1.4: NODO A

Riportiamo l'implementazione del nodo A con la sua parte operativa, la sua parte di controllo e i rispettivi port map. Dal nodo A porteremo in uscita, oltre ai segnali già previsti e mostrati in figura 7.2, anche le abilitazioni del contatore e della memoria, per poterne analizzare e visualizzare il comportamento nel testbench, che in seguito effettueremo.

```
entity A is
generic (
    I: integer :=2;
    M : integer:=8--lunghe stringhe
);
Port (
    clk_a: in std_logic;
    rst: in std_logic;
    start: in std_logic;
    ok: in std_logic;
    X: out std_logic_vector((M-1) downto 0);
    req: out std_logic;

    AInc_C1: out std_logic;
    Am_rA: out std_logic
);
end A;

architecture Structural of A is

component cont is
generic (
    I: integer :=2
);
Port ( clk : in STD_LOGIC;
       rst : in STD_LOGIC;
       inc : in STD_LOGIC; --questo è l'enable del clock, insieme danno l'impulso di conteggio
       Address : out STD_LOGIC_VECTOR ((I-1) downto 0));
end component;

component memoriaA is
generic(
    I: integer :=2;
    N : integer:= 4;--n di stringhe
    M : integer:=8--lunghe stringhe
);
port(
    CLK_A: in std_logic;
    RST: in std_logic;
    Am_r: in std_logic;
    Address_A: in std_logic_vector((I-1) downto 0); -- Address to write/read RAM
    X1: out std_logic_vector((M-1) downto 0) -- Data output of RAM
);
end component;
```

```

component RC_A is
    generic (
        I: integer :=2
    );
    Port (
        clk_a: in std_logic;
        rst: in std_logic;
        start: in std_logic;
        Address_A: in std_logic_vector((I-1) downto 0);
        ok: in std_logic;
        AInc_C: out std_logic;
        Am_r: out std_logic;
        req: out std_logic
    );
end component;

signal ab_cont: std_logic;
signal ab_mem_r: std_logic;
signal add: std_logic_vector((I-1) downto 0);

begin

contA: cont port map (
    clk =>clk_a,
    rst => rst,
    inc=> ab_cont,
    Address =>add
);

mem: memoriaA port map(
    CLK_A=> clk_a,
    RST => rst,
    Address_A=> add,
    Am_r => ab_mem_r,
    X1=>X
);
retec: RC_A Port map (
    clk_a=> clk_a,
    rst =>rst,
    start => start,
    Address_A => add,
    Am_r => ab_mem_r,
    ok=>ok,
    AInc_C => ab_cont,
    req=>req
);

AInc_C1<=ab_cont;
Am_rA <= ab_mem_r;

end Structural;

```

7.2.2: B

7.2.2.1: Registro R_IN

```
entity R_IN is
generic(
    M : integer:=8
);
Port (
    clk_b: in std_logic;
    rst: in std_logic;
    X: in std_logic_vector((M-1) downto 0);
    a_r_in: in std_logic;
    X_u: out std_logic_vector((M-1) downto 0)
);
end R_IN;

architecture Behavioral of R_IN is

begin
process(clk_b)
begin
    if(rising_edge(clk_b)) then
        if(rst='1') then
            X_u<="00000000";
        else
            if(a_r_in='1') then
                X_u<=X;
            end if;
        end if;
    end if;
end process;
end Behavioral;
```

Date le stringhe X(i) di lunghezza 8, R_IN risulta essere un generico registro adatto a memorizzare 8 bit, come già precedentemente detto.

7.2.2.2: Memoria

```
entity MemoriaB is
generic(
    I: integer :=2;
    N : integer:= 4;--n di stringhe
    M : integer:=8--lunghe stringhe
);
port(
    clk_b: in std_logic;
    rst: in std_logic;
    S: in std_logic_vector((M-1) downto 0);
    Address_B: in std_logic_vector((I-1) downto 0);
    Am_w: in std_logic;
    Am_r: in std_logic;
    Y: out std_logic_vector((M-1) downto 0);

    S1: out std_logic_vector((M-1) downto 0)
);
end MemoriaB;
```

```

architecture Behavioral of MemoriaB is

type RAM_ARRAY is array (0 to 2*N-1) of std_logic_vector ((M-1) downto 0);

signal RAM: RAM_ARRAY :=(
  b"00001000",
  b"00000011",
  b"10001001",
  b"11001000",
  b"00000000",
  b"00000000",
  b"00000000",
  b"00000000"

);

begin
process(clk_b)
begin
  if(rising_edge(clk_b)) then
    if(rst='1') then
      Y<="00000000";
    else
      if(Am_w='1') then
        RAM(to_integer(unsigned(Address_B))+N) <= s;
        s1<=s;
      end if;
      if(Am_r='1') then
        Y<= RAM(to_integer(unsigned(Address_B)));
      end if;
    end if;
  end if;
end process;

end Behavioral;

```

La memoria di B presenta 2 uscite. Un'uscita rappresenta la stringa Y(i) e l'altra uscita la somma.

Come già detto nel paragrafo precedente 7.1.2, quando procediamo con la scrittura in memoria, accediamo all'indirizzo della memoria composto dalla somma dell'Address_B e dal numero delle stringhe N. La memoria B conterrà, quindi, il un numero di stringhe pari a $2^*N = 8$.

7.2.2.3: Contatore

Il componente contatore di B è uguale al contatore di A.

Per questo motivo evitiamo di riportare nuovamente il suo codice, che possiamo osservare nel paragrafo 7.2.1.1. del nodo A.

7.2.2.4: Addizionatore

Per l'addizionatore abbiamo usato un approccio di tipo comportamentale:

```
entity Add is Port (
    X: in std_logic_vector(7 downto 0);
    Y: in std_logic_vector(7 downto 0);
    S: out std_logic_vector(7 downto 0)
);
end Add;

architecture Behavioral of Add is

begin
    process(X,Y)
    begin
        S(7 downto 0)<=X(7 downto 0)+Y(7 downto 0);
    end process;
end Behavioral;
```

7.2.2.5: RC_B

Riportiamo di seguito l'automa della rete di controllo di B, descritto nel paragrafo 7.1.2.

```
entity RC_B is
generic (
    I: integer :=2
);
Port (
    clk_b: in std_logic;
    rst: in std_logic;
    req: in std_logic;
    Address_B: in std_logic_vector((I-1) downto 0);
    a_r_in: out std_logic;
    Am_w: out std_logic;
    Am_r: out std_logic;
    A_inc: out std_logic;
    ok: out std_logic
);
end RC_B;

architecture Behavioral of RC_B is

type state is(
    WAIT_REQUEST, LOAD_R, WRITE_M,CHECK,INC_CONT,FINE
);
signal current_state, next_state: state;

begin
```

Abbiamo descritto l'automa di B secondo un approccio comportamentale, con l'uso di due process. Il primo process realizza la parte sequenziale ed è sensibile al clock di B. Al fronte di salita del clock, se il reset (quindi sincrono, in questo caso) è alto ci portiamo nello stato di IDLE, altrimenti, transitiamo al nuovo stato previsto.

```

ag_1: process(clk_b)
begin
    if(rising_edge(clk_b)) then
        if(rst='1') then
            current_state<=WAIT_REQUEST;
        else
            current_state<=next_state;
        end if;
    end if;
end process;

```

Il secondo process realizza la parte combinatoria. Esso è sensibile ai segnali di ingresso della rete di controllo (ovvero req e Address_B), nonché al current_state.

In ogni stato vengono settate le uscite della rete di controllo (l'abilitazione al registro, memoria, contatore e il segnale di ok).

```

c_1: process(current_state,req, Address_B)
begin

    case current_state is

        when WAIT_REQUEST =>
            a_r_in<='0';
            Am_w<='0';
            Am_r<='0';
            A_inc<='0';
            ok<='0';

            if(req='0')then
                next_state <=WAIT_REQUEST;
            else
                next_state <= LOAD_R;
            end if;

        when LOAD_R=>
            a_r_in<='1';
            Am_w<='0';
            Am_r<='1';
            A_inc<='0';
            ok<='0';

            next_state<=WRITE_M;
    end case;
end process;

```

```

when WRITE_M =>
    a_r_in<='0';
    Am_w<='1';
    Am_r<='0';
    A_inc<='0';
    ok<='0';

    next_state<=CHECK;

when CHECK=>
    a_r_in<='0';
    Am_w<='0';
    Am_r<='0';
    A_inc<='0';
    ok<='1';

    if(req='1') then
        next_state<=CHECK;
    else
        if(req='0' and Address_B="11") then
            next_state<=FINE;
        else
            if(req='0' and Address_B!="11") then
                next_state<=INC_CONT;
            end if;
        end if;
    end if;

when INC_CONT =>
    a_r_in<='0';
    Am_w<='0';
    Am_r<='0';
    A_inc<='1';
    ok<='1';

    next_state<=WAIT_REQUEST;

when FINE =>
    ok<='0';--abilità A a finire
end case;
end process;

end Behavioral;

```

Anche in B abbiamo scelto di terminare con uno stato di FINE, dal quale si può uscire, e quindi ritornare in WAIT_REQUEST, solo con il segnale di reset.

7.2.2.6: Nodo B

```
entity B is
generic (
    I: integer :=2;
    M: integer :=8
);
Port (
    clk_b: in std_logic;
    rst: in std_logic;
    req: in std_logic;
    X: in std_logic_vector((M-1) downto 0);
    ok: out std_logic;
    y: out std_logic_vector((M-1) downto 0);
    som: out std_logic_vector((M-1) downto 0);

    a_r_in1: out std_logic;
    Am_w1: out std_logic;
    Am_r1: out std_logic;
    A_inc1: out std_logic
);
end B;

architecture Structural of B is

component R_IN is
generic(
    M : integer:=8
);
Port (
    clk_b: in std_logic;
    rst: in std_logic;
    X: in std_logic_vector((M-1) downto 0);
    a_r_in: in std_logic;
    X_u: out std_logic_vector((M-1) downto 0)
);
end component;
```

```

component MemoriaB is
generic(
    I: integer :=2;
    N : integer:= 4;--n di stringhe
    M : integer:=8--lunghe stringa
);
port(
    clk_b: in std_logic;
    rst: in std_logic;
    S: in std_logic_vector((M-1) downto 0);
    Address_B: in std_logic_vector((I-1) downto 0);
    Am_w: in std_logic;
    Am_r: in std_logic;
    Y: out std_logic_vector((M-1) downto 0);

    S1: out std_logic_vector((M-1) downto 0)
);
end component;

component Add is Port (
    X: in std_logic_vector(7 downto 0);
    Y: in std_logic_vector(7 downto 0);
    S: out std_logic_vector(7 downto 0)
);
end component;

component cont is
generic (
    I: integer :=2
);
Port ( clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        inc : in STD_LOGIC; --questo è l'enable del clock, insieme danno l'impulso di conteggio
        Address : out STD_LOGIC_VECTOR ((I-1) downto 0));
end component;

component RC_B is
generic (
    I: integer :=2
);
Port (
    clk_b: in std_logic;
    rst: in std_logic;
    req: in std_logic;
    Address_B: in std_logic_vector((I-1) downto 0);
    a_r_in: out std_logic;
    Am_w: out std_logic;
    Am_r: out std_logic;
    A_inc: out std_logic;
    ok: out std_logic
);
end component;

```

```

signal ab_reg: std_logic;
signal xi: std_logic_vector((M-1) downto 0);
signal u_add: std_logic_vector((M-1) downto 0);
signal u_cont: std_logic_vector((I-1) downto 0);
signal ab_l: std_logic;
signal ab_s: std_logic;
signal ab_r: std_logic;
signal yi: std_logic_vector((M-1) downto 0);
signal ab_cont: std_logic;

begin

reg: R_IN Port map(
    clk_b=>clk_b,
    rst=>rst,
    X=>X,
    a_r_in=>ab_reg,
    X_u => xi
);

mem2: memoriaB port map(
    clk_b=>clk_b,
    rst=>rst,
    S => u_add,
    Address_B => u_cont,
    Am_w=> ab_s,
    Am_r => ab_r,
    Y=>yi,
    S1=>som
);

```

```

add_1: Add Port map (
    X=>xi,
    Y=>yi,
    S=>u_add
);

contB: cont Port map(
    clk =>clk_b,
    rst => rst,
    inc=> ab_cont,
    Address => u_cont
);

reteB: RC_B Port map (
    clk_b=>clk_b,
    rst => rst,
    req => req,
    Address_B => u_cont,
    a_r_in => ab_reg,
    Am_w => ab_s,
    Am_r => ab_r,
    A_inc => ab_cont,
    ok=>ok
);

a_r_in1<=ab_reg;
Am_w1 <= ab_s;
Am_r1 <= ab_r;
A_inc1 <= ab_cont;
Y<= yi;
end Structural;

```

L'implementazione del nodo B segue un approccio strutturale. Tutti i componenti che lo costituiscono, e le loro interconnessioni, sono rappresentati nella figura 7.4.

In uscita al nodo B sono riportate le abilitazioni del registro, della memoria e del contatore, per poterle visualizzare, successivamente, nel testbench. Inoltre, visualizzeremo l'uscita della memoria $Y(i)$ e il risultato della somma.

7.2.3: AB

```
entity AB is
generic (
    M: integer :=8
);
Port (
    clk_a: in std_logic;
    clk_b: in std_logic;
    rst: in std_logic;
    start: in std_logic;
    som: out std_logic_vector((M-1) downto 0);
    y: out std_logic_vector((M-1) downto 0);
    X1: out std_logic_vector((M-1) downto 0);
    req1: out std_logic;
    ok1: out std_logic;
    AInc_C1: out std_logic;
    a_r_in1: out std_logic;
    Am_w1: out std_logic;
    Am_r1: out std_logic;
    Am_rA: out std_logic;
    A_inc1: out std_logic
);
end AB;
```

Una volta realizzati i componenti A e B, possiamo creare la struttura globale che li racchiude entrambi. Realizziamo, attraverso il port map, le connessioni tra i due nodi.

In uscita al componente AB, riportiamo tutte le uscite di A e di B, al fine di verificare il corretto funzionamento dell'handshaking (riportando i segnali di req e di ok) e dei due nodi.

```
architecture Structural of AB is

component A is
generic (
    I: integer :=2;
    M : integer:=8--lunghe stringa
);
Port (
    clk_a: in std_logic;
    rst: in std_logic;
    start: in std_logic;
    ok: in std_logic;
    X: out std_logic_vector((M-1) downto 0);
    req: out std_logic;

    AInc_C1: out std_logic;
    Am_rA: out std_logic
);
end component;
```

```

component B is
generic (
    I: integer :=2;
    M: integer :=8
);
Port (
    clk_b: in std_logic;
    rst: in std_logic;
    req: in std_logic;
    X: in std_logic_vector((M-1) downto 0);
    ok: out std_logic;
    y: out std_logic_vector((M-1) downto 0);
    som: out std_logic_vector((M-1) downto 0);

    a_r_in1: out std_logic;
    Am_w1: out std_logic;
    Am_r1: out std_logic;
    A_inc1: out std_logic
);
end component;

signal okk: std_logic;
signal xx: std_logic_vector((M-1) downto 0);
signal reqq: std_logic;

begin

en_A: A Port map (
    clk_a=> clk_a,
    rst => rst,
    start => start,
    ok=> okk,
    X => xx,
    req => reqq,

    AInc_C1=>AInc_C1,
    Am_rA => Am_rA
);

X1<=xx;
req1<=reqq;
ok1<=okk;

enB: B port map(
    clk_b=> clk_b,
    rst => rst,
    req => reqq,
    X => xx,
    ok => okk,
    y => Y,
    som=> som,

```

```

    a_r_in1=>a_r_in1,
    Am_w1=> Am_w1,
    Am_r1 => Am_r1,
    A_inc1=>A_inc1
);

```

```
end Structural;
```

7.3: SIMULAZIONE

Per il testbench del componente AB, riportiamo il port map e il process, nel quale necessitiamo unicamente di dare il segnale di start (inizio), per far partire il funzionamento della macchina A e, quindi, dare il via a tutte le operazioni.

```

comp: AB Port map (
    clk_a=>clk,
    clk_b=> clk,
    rst => reset,
    start=> inizio,
    som=> somma,
    y=>y,
    x1=>x1,
    req1=>req1,
    ok1=>ok1,
    AIInc_C1=>AIInc_C1,
    a_r_in1=>a_r_in1,
    Am_w1=>Am_w1,
    Am_r1=>Am_r1,
    Am_rA=>Am_rA,
    A_inc1=>A_inc1
);

```

```

uut: process
begin

    reset<='1';
    wait for 100 ns;
    reset<='0';

    wait for 10 ns;

    inizio<='1';

    wait;
end process;

```

```
end Behavioral;
```

Lanciando la simulazione possiamo osservare il funzionamento dell'intero sistema.

Come primo passaggio, il nodo A alza l'abilitazione della lettura della memoria e al successivo colpo di clock possiamo notare in uscita X(i), ovvero il valore contenuto in memoria all'indirizzo specificato (all'inizio il valore all'address zero). Il nodo A procede anche ad alzare la richiesta verso B. Si pone, poi, in attesa dell'ok di B. Il nodo B alza l'abilitazione di load del registro e di lettura della memoria

(al colpo di clock successivo vediamo l'uscita $Y(i)$) e al successivo colpo di clock alza l'abilitazione in scrittura della memoria. Al prossimo colpo di clock alza l'ok. Il nodo A vedendo l'ok abbassa la sua richiesta. Infine B alza l'abilitazione dell'incremento del contatore e abbassa l'ok. Questo procedimento si ripete per tutti gli altri $N-1$ valori contenuti nella memoria.

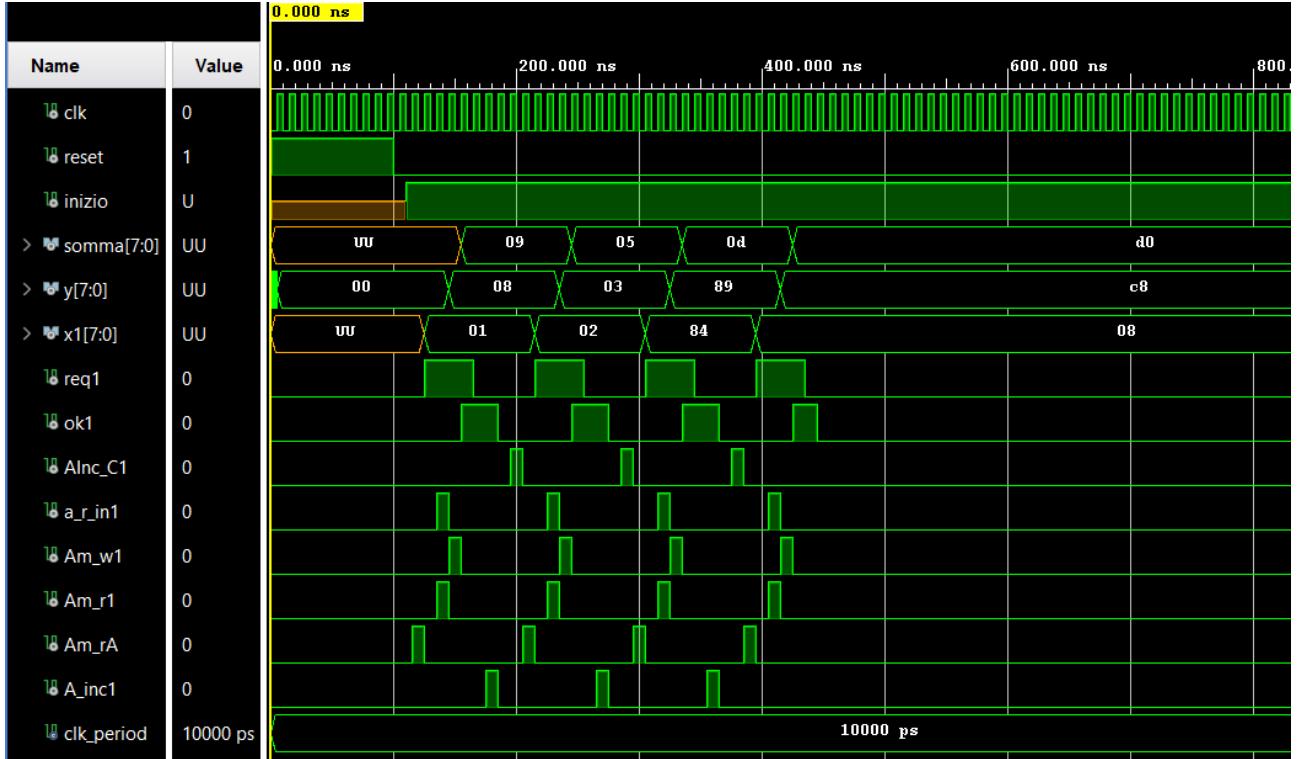


Figura 7.6: Simulazione

Notiamo come i valori di $X(i)$ e $Y(i)$ rispecchino quanto memorizzato nelle memorie di A e B, rispettivamente. Infine, il risultato della somma coincide con quanto atteso, dato il comportamento dell'addizionatore da noi descritto.

CAPITOLO 8: PROCESSORE

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

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

8.1: PROCESSORE MIC-1

Il processore Mic-1 nasce come interprete hardware del bytecode. Esso presenta un'architettura a stack e una logica microprogrammata. Il livello della microarchitettura descrive come le istruzioni ISA (Instruction Set Architecture) vengono interpretate ed eseguite dall'hardware. Ogni istruzione è dotata di 1 o 2:

- Il primo campo dell'istruzione è l'opcode che identifica l'istruzione specificando se si tratta di ADD, LOAD o altro.
- Il secondo campo, se presente, specifica gli operandi.

L'istruzione è tratta dal programma assembler scritto da un programmatore. Per eseguire l'istruzione bisogna implementare le microistruzioni necessarie.

8.1.1: Parte operativa

Il processore Mic-1 è composto da parte operativa (PO) e unità di controllo (UC). La PO è formata da dieci registri (di cui quattro per l'accesso alla memoria), due bus (tramite cui possono essere scambiati i dati tra i registri), l'ALU (per eseguire operazioni logico-aritmetiche) e uno shift register.

Riportiamo di seguito, in figura 8.1, l'unità operativa del processore e ne analizziamo i componenti.

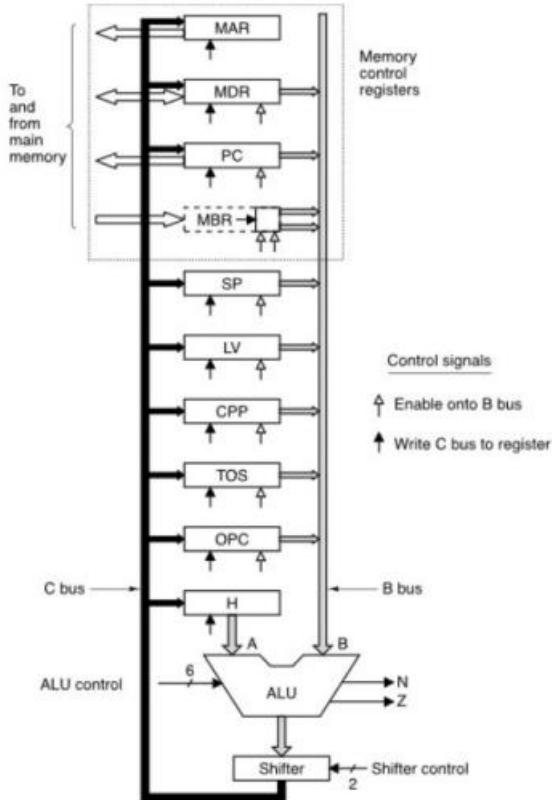


Figura 8.1: Parte operativa

Ci sono quattro registri utilizzati per comunicare con la memoria: la coppia MAR (Memory Address Register) - MDR (Memory Data Register) per la lettura/scrittura dei dati e la coppia PC (Program Counter) - MBR (Memory Byte Register) per leggere il programma eseguibile permettendo il fetch delle istruzioni ISA. MAR specifica l'indirizzo di memoria in cui si desidera leggere o scrivere una parola. MDR contiene il dato a 32 bit che sarà letto o scritto all'indirizzo di memoria specificato da MAR. Il PC è un registro a 32 bit che indica l'indirizzo di memoria della prossima istruzione ISA da caricare (fetch) e il MBR (Memory Byte Register) contiene il byte letto dalla memoria durante il fetch che viene caricato negli 8 bit meno significativi dei 32 bit disponibili.

I registri stack pointer (SP) e Top of stack (TOS) ci permettono di accedere alla testa dello stack, precisamente al suo valore e al suo indirizzo.

Il registro LV serve per puntare alla base attuale dello stack; CPP è un puntatore a valori costanti; OPC permette di appoggiare qualcosa di utile nella microistruzione. Il registro H contiene l'altro dato che viene posto in ingresso all'ALU.

I dati contenuti sulla maggior parte dei registri possono essere scritti sul bus B, ma solo uno alla volta può essere abilitato sul bus B, che quindi agisce da multiplexer. Il dato presente sul bus attraversa l'ALU. L'ALU è caratterizzata da sei linee di controllo per determinare le operazioni da svolgere. Non tutte le $2^6 = 64$ combinazioni (delle linee di controllo F0, F1, ENA, ENB, INVA, INC) sono utili; la tabella in figura 8.2 riporta quelle significative:

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

Figura 8.2: Tabella ALU

L'uscita dell'ALU, viene posta in ingresso ad uno shift register che a suo volta produce il suo output sul bus C. Dal bus C i dati possono essere caricati anche in più registri contemporaneamente.

Il comportamento dello shift register è controllato da due linee: SLL8 (Shift Left Logical) che consente di shiftare il contenuto di un byte a sinistra (ponendo gli zeri negli otto bit meno significativi) e SRA1 (Shift Right Arithmetic) che shifta il contenuto di un bit a destra mantenedo il bit più significativo invariato.

Questi componenti sono pilotati dalle microistruzioni contenute nella Control Store presente nell'unità di controllo.

8.1.2: UNITA' DI CONTROLLO

A ciascuna istruzione ISA corrisponde una microprocedura costituita da microistruzioni memorizzate all'interno della micromemoria.

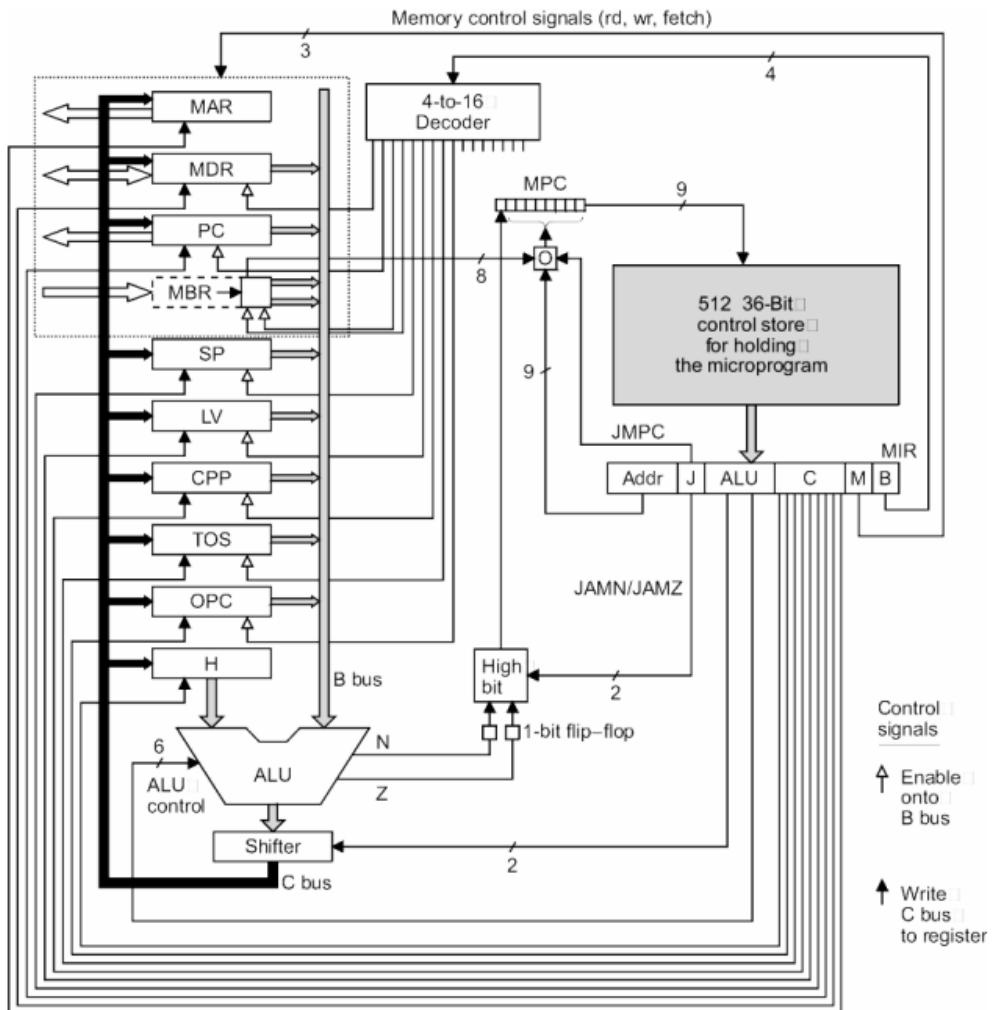


Figura 8.3 : Schema completo

La control store contiene 512 parole di 36 bit ciascuna, i quali corrispondono alla lunghezza della microistruzione. Notiamo come la control word non è di 41 bit essendo il segnale di controllo del bus B codificato su 4 bit invece che 9.

Nell'unità sono presenti 2 registri fondamentali: il MPC (MicroProgram Counter) che specifica l'indirizzo della prossima microistruzione da eseguire e il MIR (MicroInstruction Register) che memorizza la microistruzione corrente i cui bit pilotano i segnali di controllo per gestire l'unità operativa. La struttura della control word è la seguente:

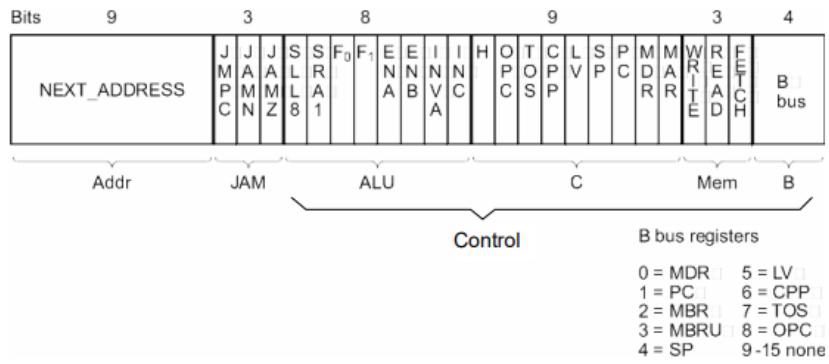


Figura 8.4 : Control word

- Next address: indica qual è la successiva microistruzione da eseguire; nasce per ottimizzare la memoria, evitando così di duplicare microistruzioni comuni a più istruzioni.
- JAM : viene usato per gestisce i salti
- ALU: per decidere quale operazioni l'ALU deve effettuare
- C: 9 bit per abilitare i registri in lettura dal bus C
- Mem: per specificare le operazioni verso la memoria
- B: per abilitare i registri in scrittura sul bus B

8.2: ANALISI ISTRUZIONI

Per la risoluzione del primo punto dell'esercizio abbiamo deciso di analizzare i codici operativi BIPUSH e IF_ICMPEQ.

8.2.1: BIPUSH

Per poter caricare i dati sullo stack il processore necessita di un'istruzione fondamentale, il BIPUSH.

La sequenza di micro istruzioni è visibile in figura 8.5

```

GNU nano 5.4          prezter@10: ~
ajvm.mal

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

```

Figura 8.5 : bipush

In particolare:

- $SP = MAR = SP + 1$: incrementiamo lo stack pointer così che possa puntare alla cella che conterrà il nuovo byte
- $PC = PC + 1$; *fetch*: eseguiamo il fetch per caricare in MBR l'opcode successivo
- $MDR = TOS = MBR$; *wr*; *goto main*: carichiamo il contenuto di MBR in TOS e MDR, eseguiamo il comando "wr" per la memorizzazione all'interno dell'indirizzo $SP+1$ caricato all'interno del MAR e ritorniamo al main

Questa istruzione prevede 2 byte, uno che specifica il codice operativo e l'altro corrisponde all'operando da caricare sullo stack.

Nella microistruzione “ $PC = PC + 1$; fetch” il byte operando è già stato pre-caricato da Main.

Riportiamo di seguito la riga di main:

$PC = PC + 1; fetch; goto (MBR)$

Essa prevede l'incremento del Program Counter ($PC = PC + 1$), il fetch del prossimo byte (opcode successivo o operando istruzione corrente) e il salto all'indirizzo dell'istruzione presente in MBR.

Eseguiamo bipush, la cui istruzione è riportata in figura 8.6, e ne riportiamo il risultato della simulazione in figura 8.7. In particolare abbiamo un primo bipush, il cui operando è 0xa (10 in decimale) e un secondo bipush, il cui operando è 0xe (14 in decimale). Essendo già presenti le due bipush nel programma, non abbiamo avuto la necessità di modificare questo file, ma ne abbiamo solo studiato il comportamento.

```

GNU nano 5.4
-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000001000000000000000000000000",
1 => "00001110000100000000101000010000",
2 => "10100111000000010011011001100101",
3 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
);

```

Figura 8.6 : *dp_ar_ram*

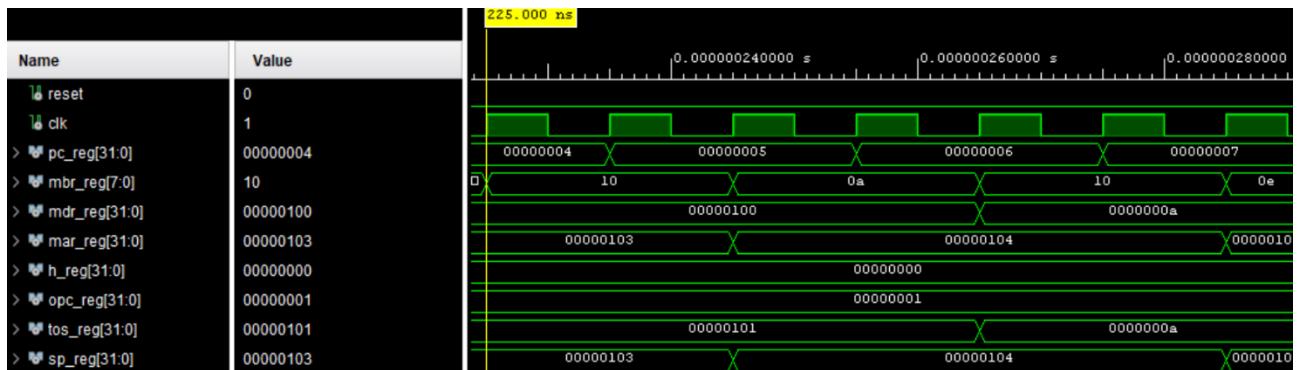


Figura 8.7: Simulazione Bipush

8.2.2: IF_ICMPEQ

La seconda istruzione analizzata è IF_ICMPEQ. Questo comando esegue, a fronte di un confronto effettuato sui due ultimi operandi caricati sullo stack, un salto condizionato.

Questa istruzione prevede tre byte, uno per l'opcode e due sono utilizzati come offset rispetto al PC per calcolare il nuovo indirizzo a cui saltare. Riportiamo di seguito la sequenza di microistruzioni.

```

if_icmpEQ = 0xA1:
    MAR = SP = SP - 1; rd
    MAR = SP = SP - 1
    H = MDR; rd
    OPC = TOS
    TOS = MDR
    Z = OPC - H; if (Z) goto T; else goto F

T:
    OPC = PC - 1; fetch; goto goto_cont

F:
    PC = PC + 1
    PC = PC + 1; fetch
    goto main

goto_cont:
    PC = PC + 1; fetch
    H = MBR << 8
    H = MBRU OR H
    PC = OPC + H; fetch
    goto main

```

Figura 8.8: *if_icmpEQ*

In particolare:

- ***MAR = SP = SP - 1; rd***: il primo operando (quello in cima allo stack) è già in TOS e quindi avvia la lettura del secondo che si trova a SP -1
- ***MAR = SP = SP - 1***: viene decrementato ulteriormente l'SP e assegnato il nuovo valore al MAR. Con questa operazione eliminiamo i primi due operandi dallo stack.
- ***H = MDR; rd***: viene copiato in H il secondo operando dello stack
- ***OPC = TOS***: il valore contenuto in TOS, che contiene già il primo operando, viene memorizzato in OPC
- ***TOS = MDR***: viene aggiornato il nuovo valore del top dello stack a quello attuale, cioè quello sotto i due operandi, a seguito dell'operazione di lettura.
- ***Z = OPC - H; if (Z) goto T; else goto F***: viene effettuata la differenza tra i due operandi e caricata in Z; in base al valore, se è alto (e quindi i due operandi sono uguali) si procede con il ramo T, altrimenti si va in F.
- T:
 - ***OPC = PC - 1; goto goto_cont***: dato che il main incrementa anticipatamente il PC, per prendere l'indirizzo dell'istruzione corrente, viene decrementato il valore del PC e memorizzato in OPC. A questo punto viene effettuato un salto incondizionato alla locazione corrente più offset.
- F:
 - ***PC = PC + 1***: viene incrementato il PC
 - ***PC = PC + 1; fetch***: viene nuovamente incrementato il PC, in modo da ignorare i due byte corrispondenti all'offset. Viene effettuata la fetch così che, ritornando nel main, l'MBR sia già pronto.

- **goto main**
- **goto_cont:**
 - **$PC = PC + 1; fetch$:** poiché nel main la lettura della prima parte dell'offset è già stata effettuata, si procede, incrementando il PC ed eseguendo la fetch, con la lettura della seconda
 - **$H = MBR \ll 8$:** la prima parte dell'offset, viene shiftata di 8 posizioni e caricata in H
 - **$H = MBRU OR H$:** viene fatta la OR tra la prima e la seconda parte dell'offset e il risultato è caricato in H
 - **$PC = OPC + H; fetch$:** il valore del PC viene aggiornato alla somma tra OPC (che contiene l'indirizzo dell'istruzione corrente) e H (che contiene l'offset). Con la fetch viene inizializzata la lettura del valore puntato dal PC
 - **goto main**

Per testare queste seconde comandi, abbiamo sfruttato due valori caricati tramite due bipush sullo stack. I due valori caricati sono 3 e 4, essendo diversi viene eseguito il ramo F.

```
-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000001000000000000000010000000",
1 => "000001000010000000001100010000",
2 => "10100111000000110000001010100001",
3 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
);
```

Figura 8.9: Caricamento valori diversi

La simulazione è visibile in figura 8.10.

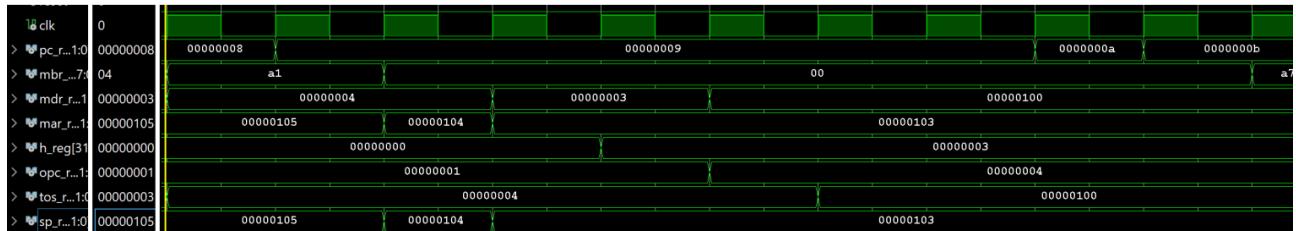


Figura 8.10: IF_CMPEQ per valori diversi

Per testare con due valori uguali, abbiamo caricato due 3 in cima allo stack, in questo modo viene eseguito il ramo T.

```
-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000001000000000000000010000000",
1 => "0000001100010000000000001100010000",
2 => "10100111000000110000001010100001",
3 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
);
```

Figura 8.11: Caricamento valori uguali

La simulazione è visibile in figura 8.12.

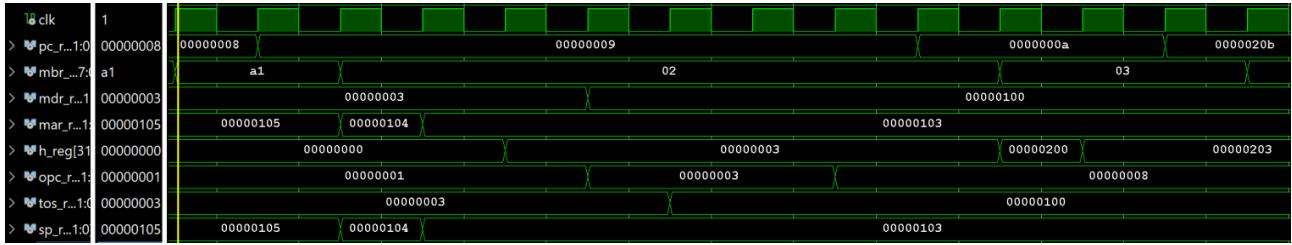


Figura 8.12: IF_CMPEQ per valori uguali

8.3: MODIFICA DEL CODICE OPERATIVO

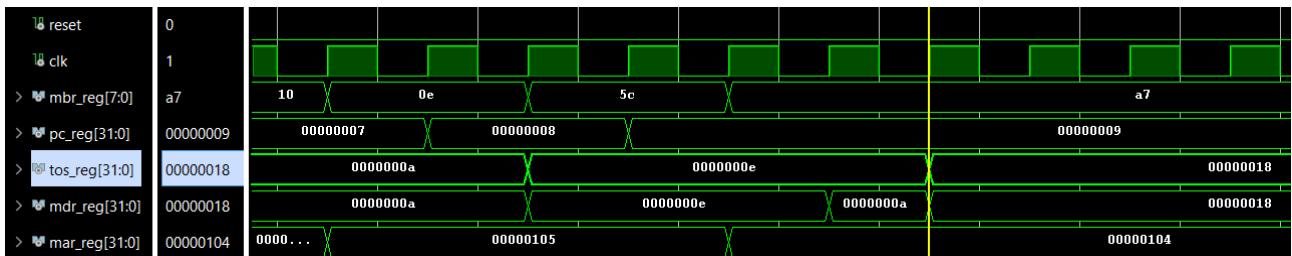
Il codice operativo che abbiamo scelto da modificare è ISUB. La sequenza di microistruzioni è riportata di seguito.

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

Per eseguire la modifica e quindi fare in modo che anziché eseguire la sottrazione esegua la somma, è stata modificata come di seguito riportato.

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

Dati i valori già presenti sullo stack, caricati dalle due bipush, ovvero "a" (10 in decimale) ed "e" (14 in decimale) vediamo il risultato del codice ISUB modificato, inserito nel programma, che dovrà quindi effettuare una somma invece di una sottrazione.



Possiamo osservare che il risultato è quanto atteso, ovvero 18 (24 in decimale), cioè la somma di "a+e" (10+14 in decimale).

CAPITOLO 9: INTERFACCIA SERIALE

Esercizio 9.1

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

Esercizio 9.2

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

9.1: PERIFERICA UART

Lo UART è un dispositivo hardware che prevede sia funzioni di trasmettitore che di ricevitore, in particolare il trasmettitore prende un byte di informazioni in parallelo sulla porta DBIN e le trasmette in serie sulla porta TXD, mentre il ricevitore prende le informazioni in serie sulla porta RXD e le trasmette in parallelo sulla porta DBOUT.

Per poter comunicare i dispositivi devono conoscere la struttura del pacchetto trasmesso, che nel caso dell'UART prevede: un bit di START, uno di STOP e, tra questi due bit, ci sarà il byte di dato da dover trasmettere. Nonostante venga definito il baud rate (cioè il numero di transizioni che avvengono sulla linea), affinché i dispositivi possano comunicare correttamente è fondamentale che il ricevitore sovraccampioni la linea, lavorando solitamente ad una frequenza 8-16 volte maggiore di quella del trasmettitore. In particolare, il ricevitore appena legge il bit di START incomincia a campionare i primi 8 campioni, posizionandosi al centro del bit, e per poter restare sempre al centro anche dei successivi bit, da quel momento in poi, comincerà a campionare ogni 16. D'altro canto, il trasmettitore è molto più semplice: l'inizio della trasmissione è sancito dall'unità di controllo. Il trasmettitore dovrà gestire il bit di START e al termine della comunicazione dovrà alzare il flag TBE per indicare che il bus è stato svuotato, mentre il ricevitore alzerà il bit RDA per indicare che il dato è disponibile per la lettura.

Esistono tre diverse modalità di trasmissione:

- Simplex: comunicazione unidirezionale, solo il trasmettitore può comunicare con il ricevitore
- Half-Duplex: trasmettitore e ricevitore possono comunicare, sullo stesso canale, ma solo uno per volta
- Full-Duplex: comunicazione bidirezionale, sullo stesso canale, che può avvenire anche nello stesso momento.

Sulla board, è presente lo UART in modalità full-duplex.

9.1.1: TRASMETTITORE

Per poter comprendere come lavora questo componente dobbiamo scendere più nel dettaglio ed analizzarlo sia lato trasmettitore che ricevitore.

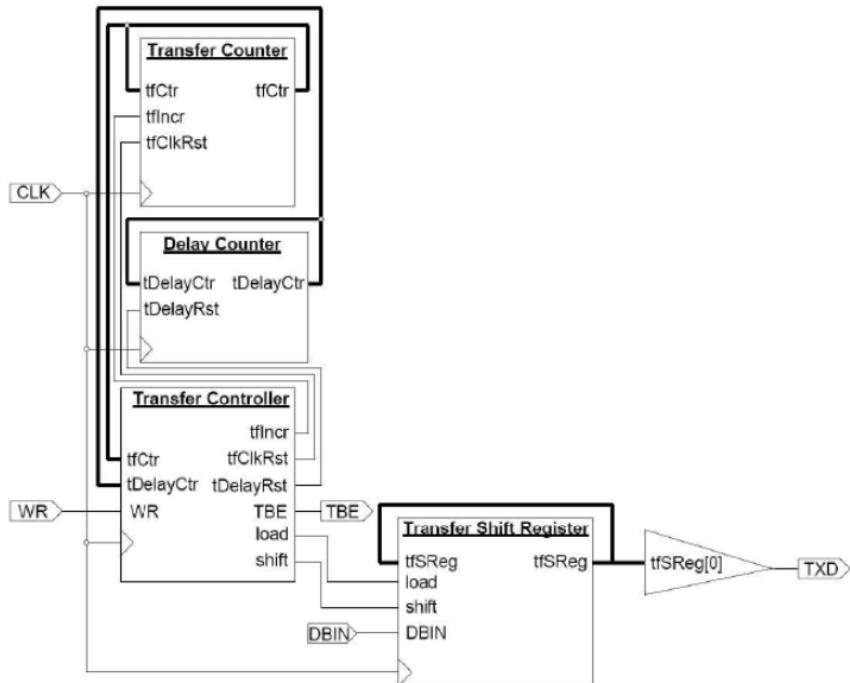


Figura 9.1: Schema a blocchi del trasmettitore

Questo blocco contiene un controller di trasferimento, due contatori per la sincronizzazione e un registro di spostamento del trasferimento. In particolare, il controller utilizza i due contatori per controllare la velocità con cui il byte di dato deve essere trasmesso attraverso la porta TXD e il numero di bit trasmessi. Un contatore viene utilizzato per ritardare il controller di trasferimento tra le trasmissioni, mentre l'altro viene utilizzato per tener traccia di quante trasmissioni sono state inviate.

Il trasmettitore, prima di iniziare la trasmissione dei bit, attende il segnale di write per poter evolvere in un nuovo stato e caricare il dato sul registro a scorrimento. Dopodiché si preoccuperà di inviare tutti e dieci i bit definiti nella comunicazione UART, al termine della quale attenderà che il segnale di write torni a zero, così che non si possa incorrere in trasmissioni multiple. L'automa del trasmettitore dell'UART è visibile in figura 9.2.

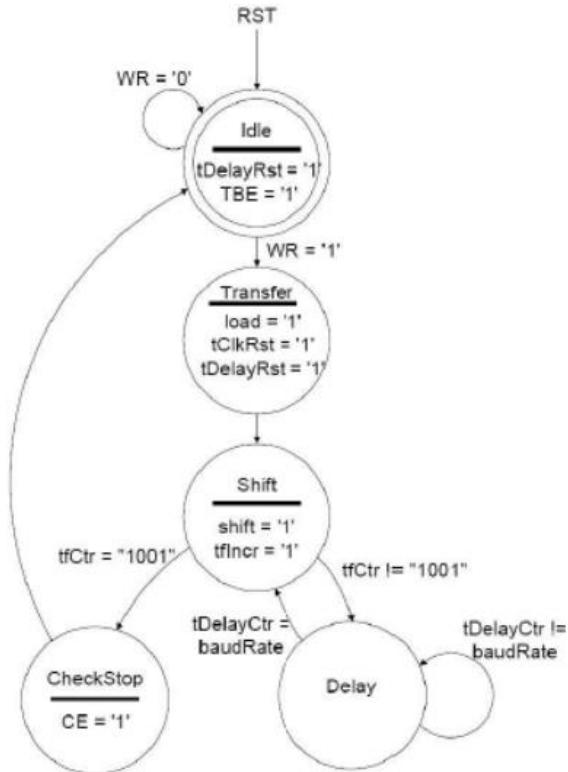


Figura 9.2: Automa trasmettitore

9.1.2: RICEVITORE

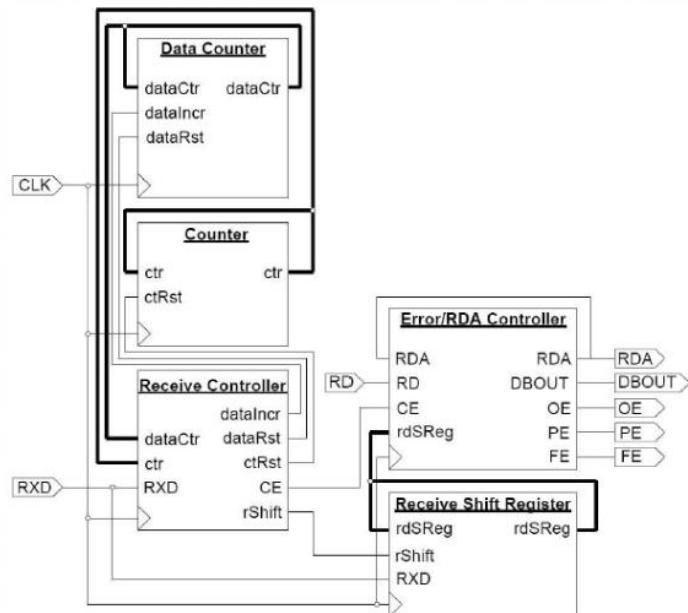


Figura 9.3: Schema a blocchi del ricevitore

Questo blocco presenta un controller dei dati seriali, due contatori per la sincronizzazione, un controller per i bit di errore e un registro a scorrimento.

Così come per il trasmettitore, anche il ricevitore per poter lavorare attende un segnale di ingresso, in particolare che il valore di RXD sia uguale a zero; infatti, essendo RXD collegato a TXD, quest'ultimo

assume valore alto fin tanto che non si inizia una nuova trasmissione, in questo modo la line passa da un livello logico alto ad un livello logico basso e il ricevitore comprende che sta iniziando una nuova trasmissione.

Ricevuto questo segnale, il contatore inizia a contare 8 colpi di clock, in modo che possa posizionarsi al centro del primo bit, a questo punto avanza in un nuovo stato: qui inizierà a contare altri 8 colpi di clock, in modo da terminare il primo bit, passa allora al successivo, attenderà ancora una volta che il contatore abbia raggiunto un valore pari a 8 in modo da essere certo di essersi posizionato al centro del nuovo bit, e solo a questo punto, incrementerà il numero di bit ricevuti e abiliterà lo shift register a shiftare. Una volta ricevuti i 10 bit prestabiliti dalla comunicazione UART, il ricevitore avanza in uno stato in cui si effettuerà un controllo dell'errore, per ritornare solo a questo punto nello stato di partenza, in attesa di ricevere un nuovo frame. L'automa del ricevitore è visibile in figura 9.4.

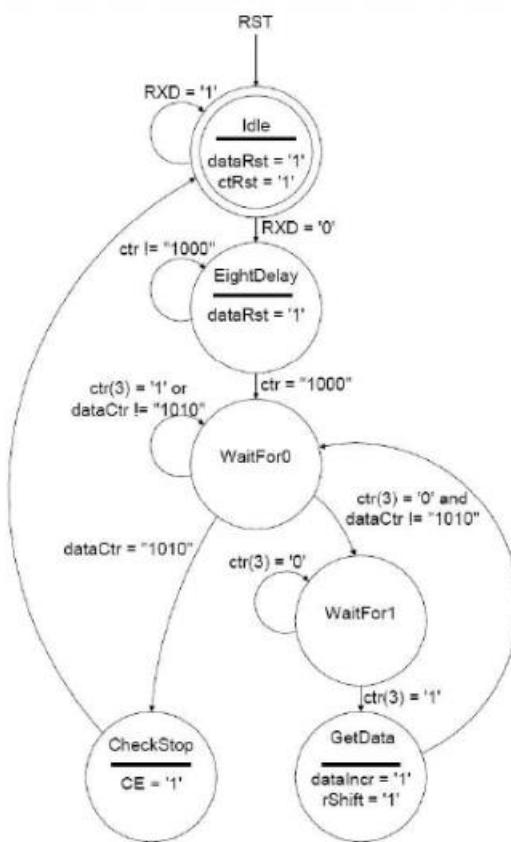


Figura 9.4: Automa ricevitore

9.2: UART IN CONFIGURAZIONE TAPPO

Compresa a questo punto la struttura del componente hardware utilizzato, possiamo spiegare come è stato realizzato il primo punto di questo esercizio; in particolare è stato utilizzato un unico componente in configurazione “tappo”, collegando l’uscita del trasmettitore all’ingresso del ricevitore, così da avere gli ingressi (cioè gli switch) collegati all’ingresso DBIN e le uscite (i led) collegati a DBOUT.

Abbiamo inoltre sfruttato un bottone per fornire l’ingresso WR, in modo da far partire la trasmissione, e tre led per visualizzare, eventualmente, i bit di errore. In particolare, il led su cui è

mappato OE diventerà luminoso così da segnalare che il dato inviato al ricevitore e presente sul registro non è stato letto.

In fine, avendo selezionato un baud rate di 48Mhz, e avendo sulla scheda un clock di 100MHz, abbiamo usato un divisore di frequenza così da dividere la frequenza per due, in modo da essere più vicini al valore del baud rate.

```

entity scheda is port (
    CLK: in std_logic;
    RST: in std_logic;
    DBIN: in std_logic_vector(7 downto 0);
    WR: in std_logic;
    PE: out std_logic;
    FE: out std_logic;
    OE: out std_logic;
    DBOUT: out std_logic_vector(7 downto 0)
);
end scheda;

architecture structural of scheda is
COMPONENT UARTcomponent is
    Generic (
        --@48MHz
        BAUD_DIVIDE_G : integer := 26;    --115200 baud
        BAUD_RATE_G   : integer := 417

        --@26.6MHz
        --BAUD_DIVIDE_G : integer := 14;    --115200 baud
        --BAUD_RATE_G   : integer := 231
    );
    Port (
        TXD      : out  std_logic  := '1';
        RXD      : in   std_logic;
        CLK      : in   std_logic;
        DBIN     : in   std_logic_vector (7 downto 0);
        DBOUT    : out  std_logic_vector (7 downto 0);
        RDA      : inout std_logic;
        TBE      : out  std_logic  := '1';
        RD       : in   std_logic;
        WR       : in   std_logic;
        PE       : out  std_logic;
        FE       : out  std_logic;
        OE       : out  std_logic;
        RST      : in   std_logic  := '0');
    end component;

    signal C: std_logic;
    signal clk_d: std_logic;

begin

clk_div: process(clk)
    variable count: integer range 0 to 1 :=0;
    begin
        if(rising_edge (clk)) then
            if(rst='1') then
                count:=0;
                clk_d<='0';
            else
                if(count=1) then
                    clk_d<='1';
                else
                    clk_d<='0';
                end if;
                count:=count+1;
            end if;
        end if;
    end process;

```

```

        else
            if(count=1) then
                clk_d<='1';
                count:=0;
            else
                clk_d<='0';
                count:=count+1;
            end if;
        end if;
    end if;
end process;

```

```

UART: UARTcomponent
Port map (
    TXD =>c,
    RXD =>c,
    CLK =>clk_d,
    DBIN =>DBIN,
    DBOUT =>DBOUT,
    RDA =>OPEN,
    TBE =>OPEN,
    RD =>'0',
    WR =>WR,
    PE =>PE,
    FE =>FE,
    OE =>OE,
    RST =>RST
);
end structural;

```

9.3: 2_UART_MEM

Per la risoluzione del secondo punto è stata realizzata l'architettura visibile in figura 9.5.

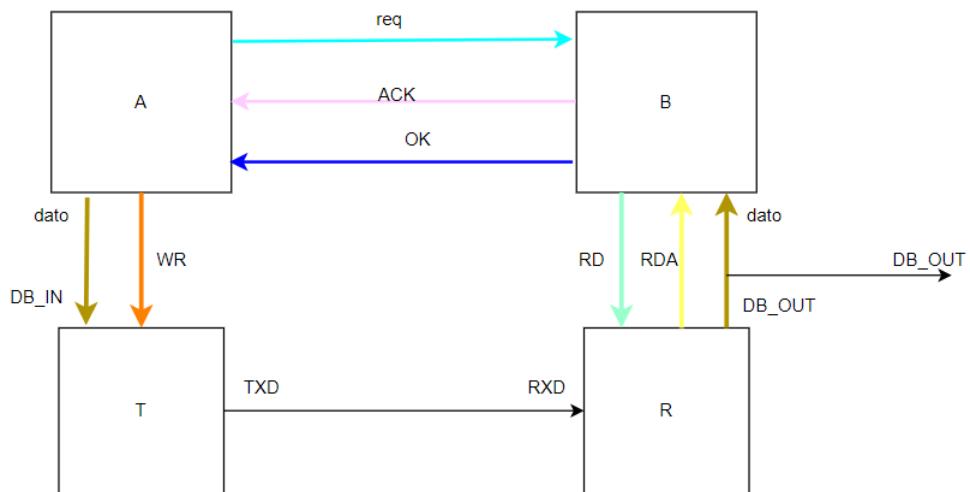


Figura 9.5: Architettura 2 UART

Sono state realizzate due entità, tra le quali è stato definito un protocollo di handshaking completo; in questo modo riusciamo ad evitare l'errore di overrun, che altrimenti sarebbe inevitabile. Ciascuna entità comunica con un'entità UART, la prima (T), ovvero quella che comunica con A è stata usata dal lato del trasmettitore, mentre la seconda entità R, viene utilizzata lato del ricevitore: in particolare A comunica con T inviando a quest'ultima dati in parallelo e un segnale di WR per iniziare la trasmissione; d'altro canto R riceve i dati da R in parallelo insieme al dato RDA, per indicare che i dati

sono disponibili per la lettura, in risposta a questi segnali B invia ad R il segnale di RD per simulare la lettura dal buffer, così da evitare l'errore di overrun.

L'entità A è stata realizzata in maniera strutturale da una ROM un contatore e una rete di controllo (figura 9.6); la ROM contiene le stringhe che devono essere inviate in parallelo all'entità T, per poter scorrere le celle di questa memoria è stato utilizzato un contatore che venisse incrementato in modo da inviare di volta in volta, tutte le stringhe presenti in memoria. Il contatore realizzato è un contatore modulo 5, essendo che la ROM contiene 5 stringhe da dover inviare.

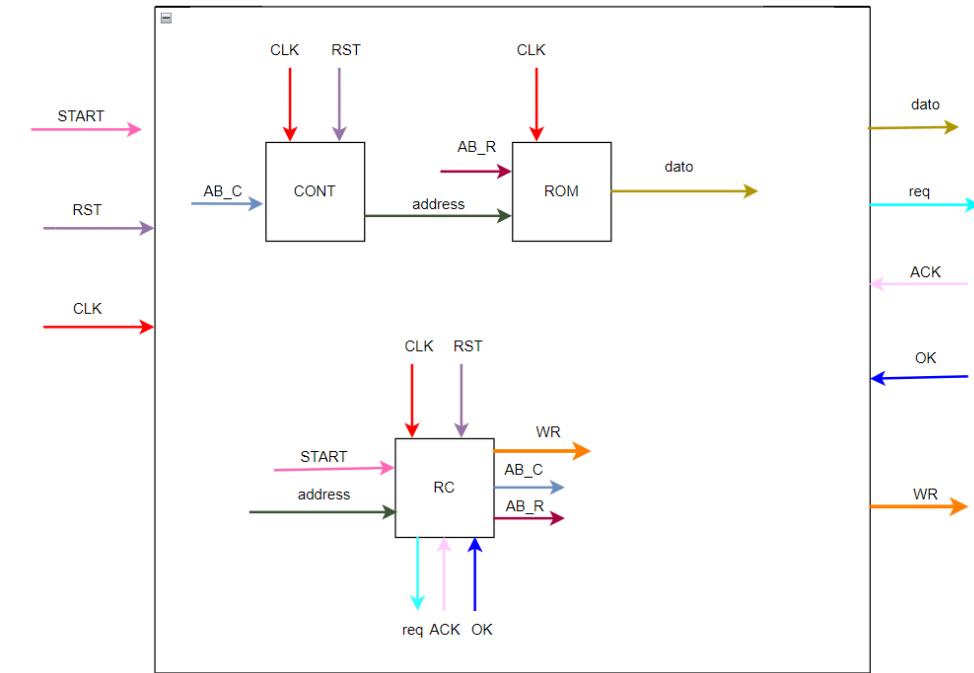


Figura 9.6: Architettura di A

La rete di controllo è naturalmente la parte che gestisce sia la comunicazione con B, che la trasmissione dei dati a T. Abbiamo deciso di realizzarla in logica cablata e l'automa è riportato in figura 9.7.

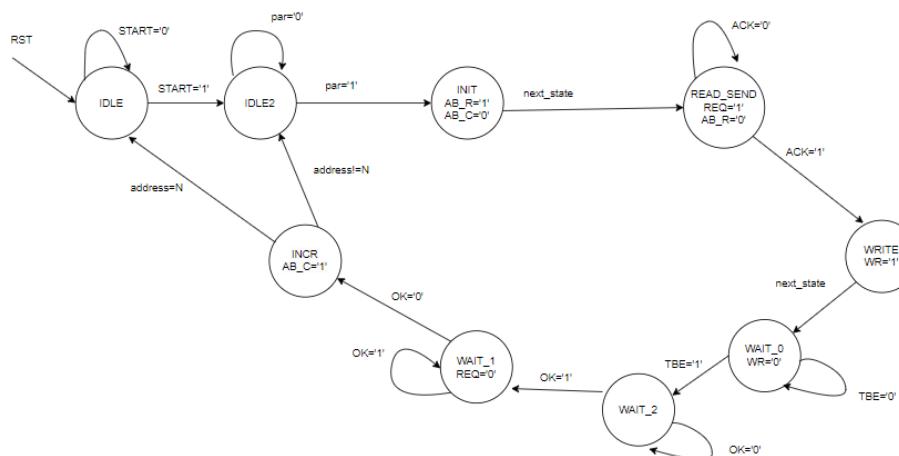


Figura 9.7 : Automa della RC di A

Più nel dettaglio, A attende di ricevere il bit di START in modo da iniziare, tuttavia, prima che possa trasmettere le diverse stringhe presenti nella ROM, l'entità A dovrà ricevere ogni volta un segnale in ingresso (par) che istaurare effettivamente la connessione e trasmettere le stringhe. Una volta ricevuto, inizia le sue operazioni abilitando in primis la lettura dalla ROM così che possa essere letto il dato in esso contenuto e di conseguenza possa essere inviato a T. A questo punto invierà a B la richiesta di voler comunicare ed attenderà di ricevere da quest'ultima un messaggio di risposta contenente l'ack. Una volta ricevuto l'ack e consapevole, dunque, della volontà di B di voler comunicare, alza il segnale di WR verso il trasmettitore così da sancire l'inizio della trasmissione. A questo punto non resterà che attendere il messaggio TBE dal trasmettitore, che quando si alzerà segnalera ad A di aver terminato la trasmissione, A resterà allora in attesa di ricevere l'ok da B, in modo da notificare l'avvenuta esecuzione e ricezione del dato. Solo a questo punto sarà possibile abbassare la richiesta e attendere che anche il segnale di ok si azzeri. A termine di queste operazioni abiliterà l'incremento del contatore che a seconda del conteggio stabilirà se l'operazione deve essere ripetuta, ovvero quando abbiamo ancora stringhe da inviare e quindi il valore del conteggio è minore di N (numero delle stringhe), oppure se l'automa debba ritornare nello stato di IDLE, e quindi attendere di ricevere un nuovo start per ricominciare l'esecuzione.

Di seguito riportiamo il codice della RC di A:

```

entity RC_A is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    start: in std_logic;
    Address_A: in std_logic_vector(2 downto 0);
    ack: in std_logic;
    ok: in std_logic;
    PAR: in std_logic;
    TBE: in std_logic;
    WR: out std_logic;
    AB_C: out std_logic;
    AB_R: out std_logic;
    REQ: out std_logic
  );
end RC_A;

architecture Behavioral of RC_A is
type state is(
  IDLE, IDLE2, INIT, READ_SEND, WRITE, WAIT_0, WAIT_1, WAIT_2, INCR
);
signal current_state, next_state: state;

begin

```

```

ag: process(clk)
begin
    if(rising_edge(clk)) then
        if(rst='1') then
            current_state<=IDLE;
        else
            current_state<=next_state;
        end if;
    end if;
end process;

c: process(current_state, start, ok, ack, Address_A)
begin

case current_state is

when IDLE =>
    ab_c<='0';
    ab_r <='0';
    req<='0';
    wr<='0';

    if(start='0') then
        next_state <=IDLE;
    else
        if(start='1') then
            next_state <= IDLE2;
        end if;
    end if;

when IDLE2 =>
    ab_c<='0';
    ab_r <='0';
    req<='0';
    wr<='0';

    if(par='0') then
        next_state <=IDLE2;
    else
        if(par='1') then
            next_state <= INIT;
        end if;
    end if;

when INIT =>
    ab_c<='0';
    ab_r <='1';
    req<='0';
    wr<='0';

    next_state<=READ_SEND;

when READ_SEND =>
    ab_c<='0';
    ab_r <='0';
    req<='1';
    wr<='0';

```

```

if(ACK='0') then
    next_state<=READ_SEND;
else
    if(ACK='1') then
        next_state<=WRITE;
    end if;
end if;

when WRITE =>
    ab_c<='0';
    ab_r <='0';
    req<='1';
    wr<='1';

    next_state<=WAIT_0;

when WAIT_0 =>
    ab_c<='0';
    ab_r <='0';
    req<='1';
    wr<='0';

    if(tbe='0') then
        next_state<=WAIT_0;
    else
        if(tbe='1') then
            next_state<=WAIT_2;
        end if;
    end if;

when WAIT_2 =>
    ab_c<='0';
    ab_r <='0';
    req<='1';
    wr<='0';

    if(ok='0') then
        next_state<=WAIT_2;
    else
        if(ok='1') then
            next_state<=WAIT_1;
        end if;
    end if;

when WAIT_1 =>
    ab_c<='0';
    ab_r <='0';
    req<='0';
    wr<='0';

    if(OK='1') then
        next_state<=WAIT_1;
    else
        if(ok='0') then
            next_state<=INCR;
        end if;
    end if;

```

```

when INCR =>
    ab_c<='1';
    ab_r <='0';
    req<='0';
    wr<='0';

    if(Address_A!="100") then
        next_state<=IDLE2;
    else
        if(Address_A=="100") then
            next_state<=IDLE;
        end if;
    end if;

end case;
end process;

end Behavioral;

```

In maniera analoga anche B è stata realizzata in modo strutturale tramite un contatore, un registro, una memoria e naturalmente l'unità di controllo (figura 9.8). In particolare, il registro è stato utilizzato per memorizzare la stringa proveniente dall'entità R in parallelo, alla fine della memorizzazione, grazie al conteggio di un contatore interno a B, questa stringa veniva memorizzata nell'opportuna cella della memoria.

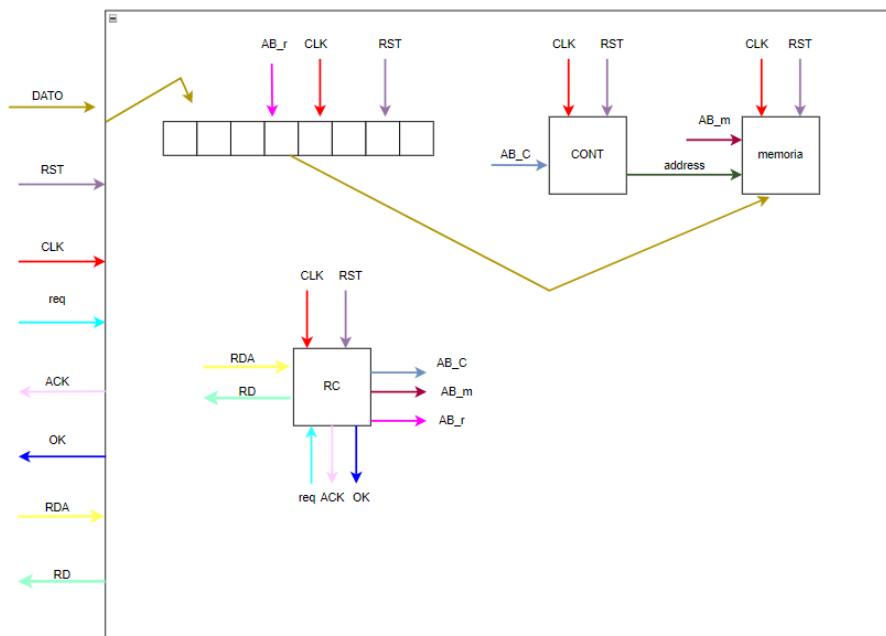


Figura 9.8: Architettura di B

Anche l'automa di B è stato realizzato in logica cablata ed è visibile in figura 9.9.

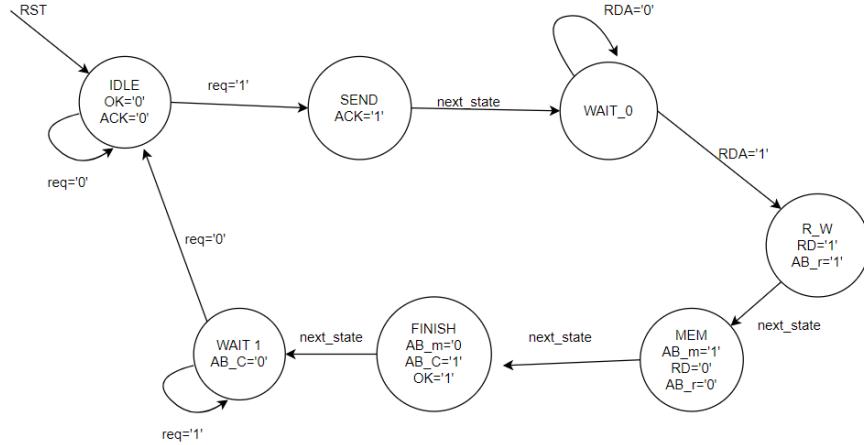


Figura 9.9: Automa della RC di B

La rete di controllo di B, realizzata anche in questo caso in logica cablata, attraversa diversi stati, partendo da quello di IDLE: resta in questo stato fin tanto che non riceve la richiesta da parte di A che gli comunica la volontà di comunicare. A questo punto invia l'ack così da notificare ad A l'avvenuta recezione della richiesta, dopodiché non gli resterà che attendere di ricevere il segnale RDA dal ricevitore. Quest'ultimo alzerà il flag per segnalare la disponibilità in lettura dei dati. B, simulerà allora il comportamento del processore alzando il bit di RD, farà in modo da notificare a R che il dato è stato letto, così facendo non si verificherà l'errore di overrun; difatti, se questo segnale non si alzasse mai, il ricevitore non potrebbe sapere che il dato sia stato effettivamente letto e dunque penserà che si sia verificata una sovrascrittura del dato causando così l'errore di overrun. Oltre ad alzare questo flag, B abiliterà anche la memorizzazione sul registro e successivamente in memoria. Solo a questo punto potrà effettivamente alzare l'ok e attendere che la richiesta si abbassi, in modo da poter poi ritornare in IDLE e attendere l'inizio di una nuova trasmissione.

Di seguito riportiamo il codice della RC di B:

```

entity RC_B is
    Port (
        clk: in std_logic;
        rst: in std_logic;
        req: in std_logic;
        rda: in std_logic;
        ab_m: out std_logic;
        ab_c: out std_logic;
        ab_r: out std_logic;
        ack: out std_logic;
        ok: out std_logic;
        rd: out std_logic
    );
end RC_B;

architecture Behavioral of RC_B is
    type state is(
        IDLE, SEND, WAIT_0, R_W, MEM, FINISH, WAIT_1
    );
    signal current_state, next_state: state;
begin

```

```

ag: process(clk)
begin
    if(rising_edge(clk)) then
        if(rst='1') then
            current_state<=IDLE;
        else
            current_state<=next_state;
        end if;
    end if;
end process;

c: process(current_state, req, rda)
begin

case current_state is

when IDLE =>
    ab_m<='0';
    ab_c<='0';
    ab_r<='0';
    ack<='0';
    ok<='0';
    rd<='0';

    if(req='0') then
        next_state <=IDLE;
    else
        if(req='1') then
            next_state <= SEND;
        end if;
    end if;

when SEND =>
    ab_m<='0';
    ab_c<='0';
    ab_r<='0';
    ack<='1';
    ok<='0';
    rd<='0';

    next_state<=WAIT_0;

when WAIT_0 =>
    ab_m<='0';
    ab_c<='0';
    ab_r<='0';
    ack<='1';
    ok<='0';
    rd<='0';

```

```

        if(rda='0') then
            next_state<=WAIT_0;
        else
            if(rda='1') then
                next_state<=R_W;
            end if;
        end if;

when R_W =>
    ab_m<='0';
    ab_c<='0';
    ab_r<='1';
    ack<='1';
    ok<='0';
    rd<='1';

    next_state<=MEM;

when MEM =>
    ab_m<='1';
    ab_c<='0';
    ab_r<='0';
    ack<='1';
    ok<='0';
    rd<='0';

    next_state<=FINISH;

when FINISH =>
    ab_m<='0';
    ab_c<='1';
    ab_r<='0';
    ack<='1';
    ok<='1';
    rd<='0';

    next_state<=WAIT_1;

when WAIT_1 =>
    ab_m<='0';
    ab_c<='0';
    ab_r<='0';
    ack<='1';
    ok<='1';
    rd<='0';

    if(req='1') then
        next_state<=WAIT_1;
    else
        if(req='0') then
            next_state<=IDLE;
        end if;
    end if;

end case;
end process;
end Behavioral;

```

L'interconnessione dei 4 componenti (entità A, trasmettitore della prima UART, ricevitore della seconda UART ed entità B) è stata realizzata come di seguito riportata:

```

entity TOT is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    start: in std_logic;
    par: in std_logic;
    dato_in: out std_logic_vector(7 downto 0);
    dato_out: out std_logic_vector(7 downto 0);
    PE: out std_logic;
    FE: out std_logic;
    OE: out std_logic;

    divisore: out std_logic;
    wr: out std_logic;
    tyy: out std_logic;
    ab_c: out std_logic
  );
end TOT;

architecture structural of TOT is

component A is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    start: in std_logic;
    ack: in std_logic;
    ok: in std_logic;
    par: in std_logic;
    the: in std_logic;
    wr: out std_logic;
    req: out std_logic;
    ab_c: out std_logic;
    dato: out std_logic_vector(7 downto 0)
  );
end component;

component B is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    dato: in std_logic_vector(7 downto 0);
    req: in std_logic;
    rda: in std_logic;
    ack: out std_logic;
    ok: out std_logic;
    rd: out std_logic;
    dato_out: out std_logic_vector(7 downto 0)
  );
end component;

```

```

component UARTcomponent is
  Generic (
    --@48MHz
    BAUD_DIVIDE_G : integer := 26; --115200 baud
    BAUD_RATE_G   : integer := 417
  );
  Port (
    TXD      : out  std_logic  := '1';
    RXD      : in   std_logic;
    CLR      : in   std_logic;
    DBIN     : in   std_logic_vector(7 downto 0);
    DBOUT    : out  std_logic_vector(7 downto 0);
    RDA      : inout std_logic;
    TBE      : out  std_logic  := '1';
    RD       : in   std_logic;
    WR       : in   std_logic;
    PE       : out  std_logic;
    FE       : out  std_logic;
    OE       : out  std_logic;
    RST      : in   std_logic  := '0'
  );
end component;

signal ackk: std_logic;
signal okk: std_logic;
signal wrx: std_logic;
signal reqq: std_logic;
signal datoo: std_logic_vector(7 downto 0);

signal ty: std_logic;

signal dbout: std_logic_vector(7 downto 0);
signal rdःaa: std_logic;
signal rdd: std_logic;

signal clk_d: std_logic;
signal d: std_logic_vector(7 downto 0);

-- 
signal tbel: std_logic;
begin

  aa: A Port map (
    clk=>clk_d,
    rst=>rst,
    start=>start,
    ack=>ackk,
    ok=>okk,
    par=>par,
    tbe => tbel,
    wr=>wr,
    ab_c=>ab_c,
    req=>reqq,
    datoo=>datoo
  );

```

```

UART_T: UARTcomponent Port map (
    TXD =>ty,
    RXD =>'1',
    CLK => clk_d,
    DBIN => datoo,
    DBOUT=>OPEN,
    RDA=>OPEN,
    TBE=>tbel,
    RD=>'0',
    WR=>wrr,
    PE=>open,
    FE=>open,
    OE=>open,
    RST=>rst
);

UART_R: UARTcomponent Port map (
    TXD =>open,
    RXD =>ty,
    CLK => clk_d,
    DBIN => datoo,
    DBOUT=>dbout,
    RDA=>rdaa,
    TBE=>OPEN,
    RD=>rdd,
    WR=>'0',
    PE=>PE,
    FE=>FE,
    OE=>OE,
    RST=>rst
);

bb: B port map (
    clk=>clk_d,
    rst=>rst,
    dato=>dbout,
    req=>reqq,
    rda=>rdaa,
    ack=>ackk,
    ok=>okk,
    rd=>rdd,
    dato_out=>d
);

dato_in<=datoo;
dato_out<=dbout;
divisore<=clk_d;
wr<=wrr;
tty<=ty;

clk_div: process(clk)
    variable count: integer range 0 to 1 :=0;
begin
    if(rising_edge (clk)) then
        if(rst='1') then
            count:=0;
            clk_d<='0';
        else
            if(count=1) then
                clk_d<='1';
                count:=0;
            else
                clk_d<='0';
                count:=count+1;
            end if;
        end if;
    end if;
end process;

end structural;

```

Per analizzarne il comportamento, ancora una volta è stato eseguito il testbench, in figura 9.10 è possibile osservare la simulazione.

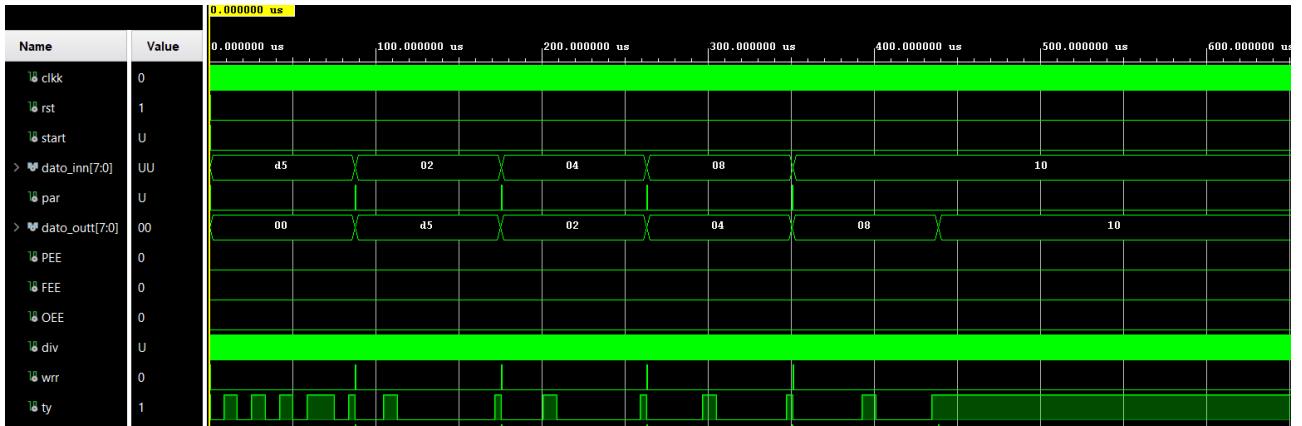


Figura 9.10 : Simulazione

Come si può osservare, tutte le stringhe vengono effettivamente trasferite da A a B, il segnale ty permette di osservare il trasferimento bit a bit tra T ed R, al termine del quale il dato è visibile su "DATO_OUT", quando si alza nuovamente il segnale di PAR inizia la trasmissione della stringa successiva e così per tutte quelle contenute nella ROM.

CAPITOLO 10: SWITCH MULTISTADIO

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 una rete con 4 nodi, implementando uno schema a priorità fissa fra i nodi (ed. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).
- b) (Opzionale) rimuovendo l'ipotesi di lavorare secondo uno schema a priorità fra i nodi e considerando una rete di 8 nodi, lo switch deve gestire eventuali conflitti generati da collisioni secondo un meccanismo a scelta (ad es. perdendo uno dei messaggi in conflitto).
- c) (Opzionale) Si implementi un protocollo di handshaking semplice regolato da una coppia di segnali (pronto a inviare/pronto a ricevere) per l'invio di ciascun messaggio fra due nodi.

10.1: COS’È LO SWITCH MULTISTADIO

Lo switch multistadio è un dispositivo che serve per interconnettere N nodi sorgente con N nodi destinazione. In particolare, a differenza dello switch a singolo stadio, che consente la comunicazione di una coppia di nodi alla volta, quello multistadio consente la comunicazione in parallelo, e quindi a più coppie di nodi di comunicare contemporaneamente. Questa soluzione architettonale, si basa sull'utilizzo di più livelli intermedi per stabilire la comunicazione tra i nodi. In particolare, vengono utilizzati $\log N$ stadi, ciascuno di essi composto da elementi semplici, l'insieme dei quali crea il blocco di base; tale blocco è composto da un multiplexer a due ingressi, che prevede come abilitazione il bit di sorgente, e un demultiplexer, che prevede invece come abilitazione i bit della destinazione. In questa architettura multistadio, infatti, l'indirizzo della sorgente e della destinazione sono definiti su k bit, divisi in tante parti quanti sono gli stadi, in modo da pilotare opportunamente i blocchi. Così facendo, la logica di controllo è distribuita sui vari livelli di cui è composta. Un modello topologico per questo tipo di interconnessione è definito Omega Network, che si basa sul concetto del perfect shuffling. Ovviamente però, questa architettura presenta dei problemi, in quanto, sebbene consenta la comunicazione parallela di più nodi, qualora questi si ritrovassero a voler comunicare con lo stesso nodo di uscita, o semplicemente a dover attraversare contemporaneamente uno stesso stadio, si verificherebbe il fenomeno della collisione che dovrebbe, dunque, essere opportunamente gestito.

10.2: SVOLGIMENTO

Il punto a di questo esercizio chiedeva di realizzare e di implementare lo scambio di messaggi da 2 bit tra 4 nodi stabilendo una priorità tra questi.

Per realizzare questo progetto, abbiamo deciso di scomporlo in due blocchi: parte operativa e parte di controllo.

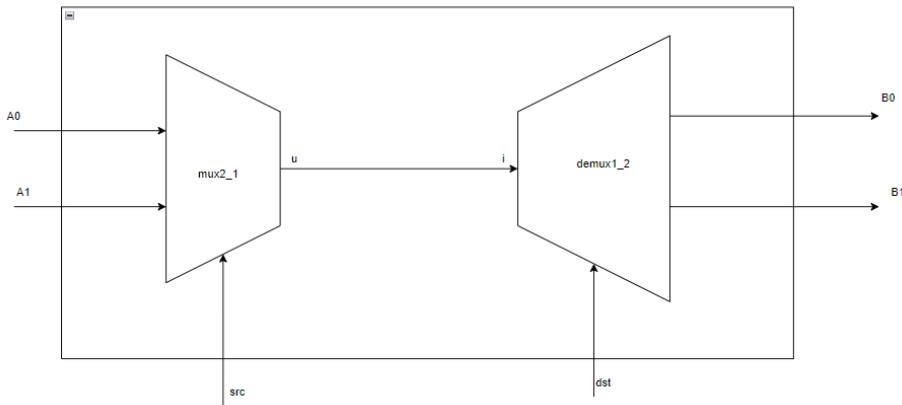


Figura 10.1 : Switch

Dovendo realizzare uno switch multistadio, siamo allora partite dalla realizzazione del blocco elementare (visibile in figura 10.1), e quindi dei singoli stadi. In particolare, come già accennato, abbiamo sfruttato un multiplexer 2:1 e un demultiplexer 1:2, che abbiamo interconnesso come di seguito riportato:

```

entity switch is
generic(
  N: integer :=2
);
Port (
  A0: in std_logic_vector((N-1) downto 0);
  A1: in std_logic_vector((N-1) downto 0);
  src: in std_logic;
  dst: in std_logic;
  B0:out std_logic_vector((N-1) downto 0);
  B1: out std_logic_vector((N-1) downto 0)
);
end switch;

architecture Structural of switch is

component mux2_1 is
generic(
  N: integer :=2
);
Port (
  A0: in std_logic_vector((N-1) downto 0);
  A1: in std_logic_vector((N-1) downto 0);
  src: in std_logic;
  U: out std_logic_vector((N-1) downto 0)
);
end component;

component demux1_2  is
generic(
  N: integer :=2
);

```

```

Port (
  I: in std_logic_vector((N-1) downto 0);
  dst: in std_logic;
  B0:out std_logic_vector((N-1) downto 0);
  B1: out std_logic_vector((N-1) downto 0)
);
end component;

signal c: std_logic_vector((N-1) downto 0);

begin

m: mux2_1 port map(
  A0=>A0,
  A1=>A1,
  src=>src,
  U=>c
);

d: demux1_2 Port map(
  I=>c,
  dst=>dst,
  B0=>B0,
  B1=>B1
);

end Structural;

```

In particolare: gli ingressi A0 e A1 dello switch rappresentano i due bit del dato che deve essere trasferito, così come le uscite. A0 e A1 rappresentano dunque gli ingressi del multiplexer, che in base al segnale di abilitazione (rappresentato dalla sorgente) decide quale dei due portare in uscita e quindi in ingresso al demultiplexer, che a sua volta, grazie al segnale di abilitazione (rappresentato dal bit della destinazione) definisce su quale uscita portare e quindi di conseguenza su quale uscita dello switch sarà disponibile il dato.

Osservazione: abbiamo definito la dimensione del pacchetto dati con un generic, in modo che, volendo modificare questo aspetto, e volendo, ad esempio, trasmettere tre bit di dati, anziché due, sarà necessario semplicemente modificare questo campo, modificando il valore della N.

Realizzato il componente elementare non restava che interconnettere questi blocchi per implementare la parte operativa (figura 10.2).

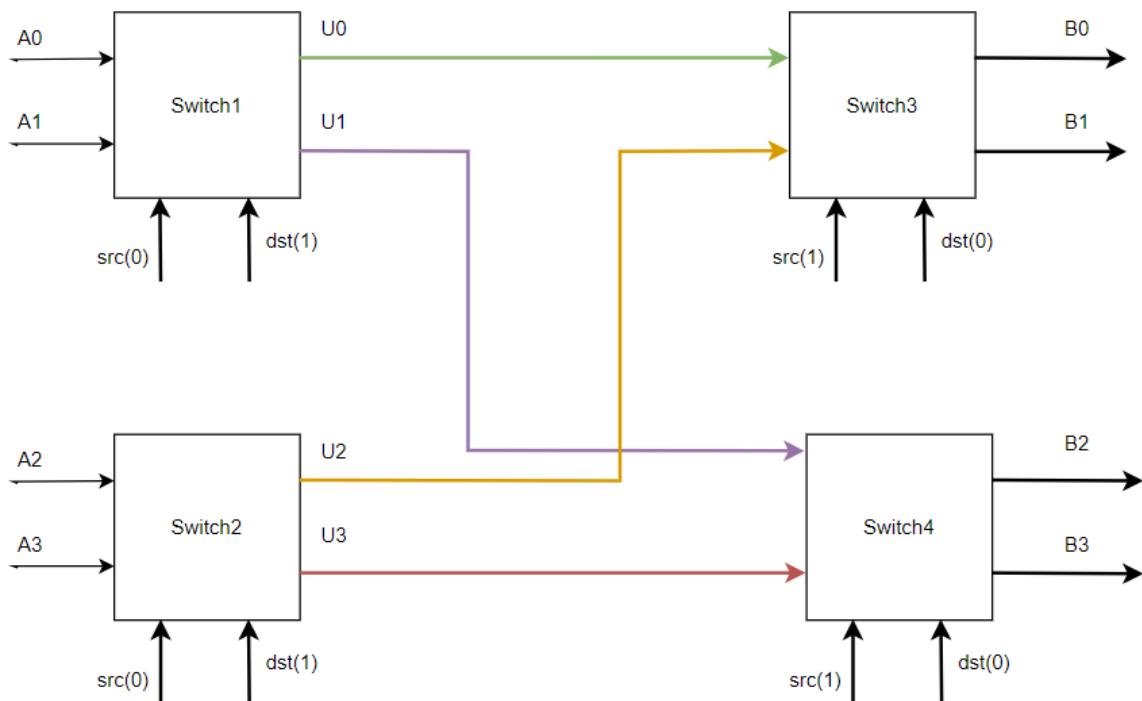


Figura 10.2: Parte operativa

Come si evince dalla figura 10.2, la parte operativa è implementata sfruttando 4 switch collegati tra di loro secondo la tecnica del perfect shuffling.

Per determinare il percorso da uno specifico nodo sorgente a uno di destinazione si valuta ad ogni stadio, il singolo bit i-esimo associato all'indirizzo destinazione, e viene selezionato il collegamento da percorrere utilizzando il seguente criterio:

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

Per realizzare tale criterio è necessario considerare come bit di selezione dei mux al primo stadio il bit meno significativo del vettore sorgente e per il demultiplexer il bit più significativo della destinazione; per gli switch al secondo stadio invece, viene considerato il secondo bit meno significativo del vettore "src", (che di fatti corrisponde al più significativo, essendo src di soli due bit) e il secondo bit più significativo per la destinazione (che di fatti corrisponde al meno significativo, essendo "dst" di soli due bit). Il codice è riportato di seguito:

```

| entity PO is
| generic(
|   N: integer :=2
| );
| Port (
|   A0, A1, A2, A3: in std_logic_vector((N-1) downto 0);
|   src: in std_logic_vector(1 downto 0);
|   dst: in std_logic_vector(1 downto 0);
|   B0, B1, B2, B3:out std_logic_vector((N-1) downto 0)
| );
| end PO;

| architecture Structural of PO is

| component switch is
| generic(
|   N: integer :=2
| );
| Port (
|   A0: in std_logic_vector((N-1) downto 0);
|   A1: in std_logic_vector((N-1) downto 0);
|   src: in std_logic;
|   dst: in std_logic;
|   B0:out std_logic_vector((N-1) downto 0);
|   B1: out std_logic_vector((N-1) downto 0)
| );
| end component;

| begin

| s1: switch Port map (
|   A0=>A0,
|   A1=>A1,
|   src=>src(0),
|   dst=>dst(1),
|   B0=>U0,
|   B1=>U1
| );

| s2: switch Port map (
|   A0=>A2,
|   A1=>A3,
|   src=>src(0),
|   dst=>dst(1),
|   B0=>U2,
|   B1=>U3
| );

| s3: switch Port map (
|   A0=>U0,
|   A1=>U2,
|   src=>src(1),
|   dst=>dst(0),
|   B0=>B0,
|   B1=>B1
| );

| s4: switch Port map (
|   A0=>U1,
|   A1=>U3,
|   src=>src(1),
|   dst=>dst(0),
|   B0=>B2,
|   B1=>B3
| );

| end Structural;

```

10.2.1: PARTE DI CONTROLLO

Per gestire la comunicazione tra i vari nodi, la parte di controllo è stata realizzata in maniera strutturale, incorporando un multiplexer, un demultiplexer e un arbitro.

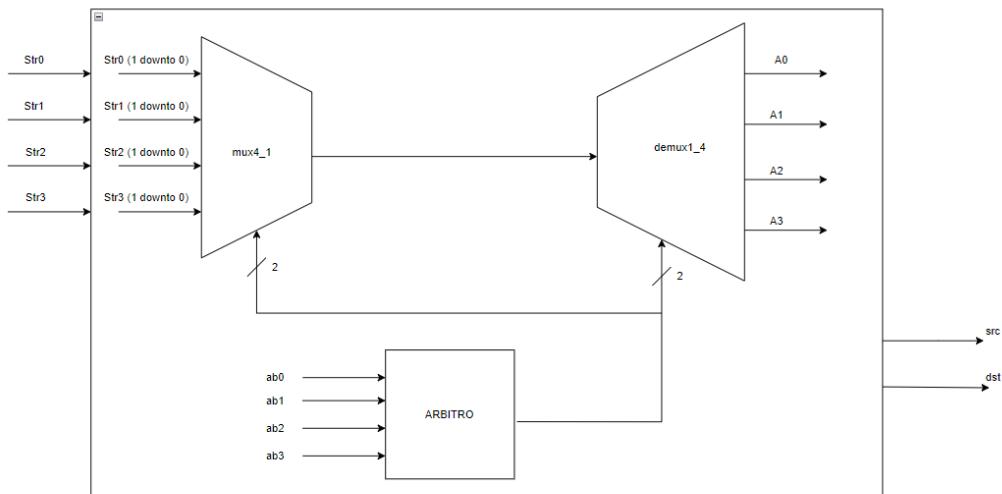


Figura 10.3 : Parte di controllo

Il multiplexer e il demultiplexer servono per gestire i bit di dato che devono essere trasmessi, perché in ingresso alla rete di controllo, il segnale che entra è un messaggio che prevede sia 2 bit per la destinazione che i due bit di dato (figura 10.4).

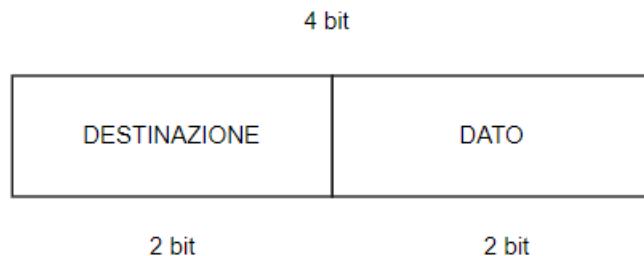


Figura 10.4: Struttura pacchetto

Ciascun nodo comunica all'unità di controllo, tramite un apposito segnale, la sua eventuale volontà di voler trasmettere. Tali segnali, sono i segnali in base al quale l'arbitro decide chi deve trasmettere. L'arbitro è stato realizzato a livello data-flow e consente di definire la priorità dei nodi che trasmettono, in questo caso il nodo con la priorità più alta è il nodo 0, seguito dal 1, e così via fino al 3. Quindi sulla base di questa priorità, verrà riportato in uscita il valore codificato in binario del nodo che intende comunicare e che ha la priorità più alta. Quindi, ad esempio, in uscita avremo 01 solo se il nodo 1 intende comunicare e il nodo 0 no, perché se anche il nodo zero volesse comunicare, avendo una priorità più alta, farà in modo che l'uscita dell'arbitro diventi 00.

```

library IEEE ;
use IEEE . STD_LOGIC_1164 . ALL ;

entity arbitro is
port (
    en0 , en1 , en2 , en3 : in std_logic ;
    y : out std_logic_vector (1 downto 0));
end arbitro ;

architecture dataflow of arbitro is

begin

y <= "00" when en0='1' else
"01" when en1='1' else
"10" when en2 ='1' else
"11" when en3 ='1' else
"00";

end dataflow;

```

Nella rete di controllo quindi viene fatto entrare tutto il messaggio, di cui, i bit meno significativi di tutti i messaggi di ingresso, rappresentano gli ingressi del multiplexer, le abilitazioni in ingresso sono le abilitazioni che vengono fornite all'arbitro in modo da poter effettuare una scelta del nodo che dovrà comunicare, sulla base della priorità implementata; le uscite sono invece composte dall'uscita del demux e dall'indirizzo della sorgente e della destinazione che viene passato poi alla parte operativa, in particolare l'indirizzo della destinazione viene valutato sulla base dell'uscita dell'arbitro che mi indicherà il nodo che deve comunicare e di conseguenza la destinazione connessa a quel nodo presente nella stringa.

```

entity CU is
generic(
    N: integer :=2
);
Port(
    Str0,Str1,Str2,Str3: in std_logic_vector((2+(N-1))  downto 0);
    ab0,ab1,ab2,ab3: in std_logic;
    src: out std_logic_vector(1 downto 0);
    dst: out std_logic_vector(1 downto 0);
    A0, A1, A2, A3: out std_logic_vector((N-1)  downto 0)
);
end CU;

architecture Structural of CU is

component mux4_1 is
generic(
    N: integer :=2
);
port (
    A0, A1, A2, A3: in std_logic_vector((N-1)  downto 0);
    s: in std_logic_vector(1  downto 0);
    U: out std_logic_vector((N-1)  downto 0)
);
end component;
component demux1_4 is
generic(
    N: integer :=2
);

```

```

Port (
    I: in std_logic_vector((N-1) downto 0);
    dst: in std_logic_vector(1 downto 0);
    B0, B1, B2, B3:out std_logic_vector((N-1) downto 0)
);
end component;

component arbitro is
port (
    en0 , en1 , en2 , en3 : in std_logic ;
    y : out std_logic_vector (1 downto 0)
);
end component;

signal u_a: std_logic_vector (1 downto 0);
signal u_mux: std_logic_vector ((N-1) downto 0);

begin

    mux1: mux4_1 port map(
        A0=>str0(1 downto 0),
        A1=>str1(1 downto 0),
        A2=>str2(1 downto 0),
        A3=>str3(1 downto 0),
        s=>u_a,
        U=> u_mux
    );
    dem1: demux1_4 port map(
        I=>u_mux,
        dst=>u_a,
        B0=>A0,
        B1=>A1,
        B2=>A2,
        B3=>A3
    );
    a: arbitro port map (
        en0=>ab0,
        en1=>ab1,
        en2=>ab2,
        en3=>ab3,
        y =>u_a
    );
    src <= u_a;
    dst <= str0 (3 downto 2) when u_a ="00" else
        str1 (3 downto 2) when u_a ="01" else
        str2 (3 downto 2) when u_a ="10" else
        str3 (3 downto 2) when u_a ="11" else
        "--";
end Structural;

```

10.2.2: INTERCONNESSIONE

Il blocco complessivo Omega_N è stato realizzato seguendo un approccio strutturale connettendo unità di controllo e unità operativa.

```

entity Omega_N is
generic(
  N: integer :=2
);
Port(
  Str0,Str1,Str2,Str3: in std_logic_vector((2+(N-1)) downto 0);
  ab0,ab1,ab2,ab3: in std_logic;
  U0, U1, U2, U3: out std_logic_vector((N-1) downto 0);

  sorg: out std_logic_vector(1 downto 0);
  dest: out std_logic_vector(1 downto 0)
);
end Omega_N;

architecture Structural of Omega_N is

component CU is
generic(
  N: integer :=2
);
Port(
  Str0,Str1,Str2,Str3: in std_logic_vector((2+(N-1)) downto 0);
  ab0,ab1,ab2,ab3: in std_logic;
  src: out std_logic_vector(1 downto 0);
  dst: out std_logic_vector(1 downto 0);
  A0, A1, A2, A3: out std_logic_vector((N-1) downto 0)
);
end component;

component PO is
generic(
  N: integer :=2
);
Port (
  A0, A1, A2, A3: in std_logic_vector((N-1) downto 0);
  src: in std_logic_vector(1 downto 0);
  dst: in std_logic_vector(1 downto 0);
  B0, B1, B2, B3:out std_logic_vector((N-1) downto 0)
);
end component;

signal sorgente: std_logic_vector(1 downto 0);
signal destinaz: std_logic_vector(1 downto 0);
signal us0: std_logic_vector((N-1) downto 0);
signal us1: std_logic_vector((N-1) downto 0);
signal us2: std_logic_vector((N-1) downto 0);
signal us3: std_logic_vector((N-1) downto 0);

begin

```

```

partC: CU Port map(
    Str0=>Str0,
    Str1=>Str1,
    Str2=>Str2,
    Str3=>Str3,
    ab0=>ab0,
    ab1=>ab1,
    ab2=>ab2,
    ab3=>ab3,
    src=> sorgente,
    dst=> destinaz,
    A0=> us0,
    A1=> us1,
    A2=> us2,
    A3=> us3
);

parteO:PO Port map(
    A0=> us0,
    A1=> us1,
    A2=> us2,
    A3=> us3,
    src=> sorgente,
    dst=> destinaz,
    B0=>U0,
    B1=>U1,
    B2=>U2,
    B3=>U3
);

sorg<=sorgente;
dest<=destinaz;

end Structural;

```

10.2.3: TESTBENCH

Per testare il componente ancora una volta è stato realizzato il testbench; in particolare abbiamo specificato i 4 messaggi per i 4 nodi e abbiamo effettuato 4 casi di test.

```

test: process
begin

    str_0<="1011";
    str_1<="0100";
    str_2<="1001";
    str_3<="1111";

    ab_0<='1';
    ab_1<='0';
    ab_2<='1';
    ab_3<='1';

    wait for 20ns;
    ab_0<='0';
    ab_1<='0';
    ab_2<='1';
    ab_3<='1';

    wait for 20ns;
    ab_0<='0';
    ab_1<='0';
    ab_2<='0';
    ab_3<='1';

    wait for 20ns;
    ab_0<='0';
    ab_1<='1';
    ab_2<='0';
    ab_3<='1';

```

Nel primo caso di test, abbiamo alzato l'abilitazione del primo, terzo e quarto nodo (che sono rispettivamente il nodo 0,2 e 3), ovvero abbiamo indicato tramite questa abilitazione la loro volontà di voler comunicare. A causa della presenza dell'arbitro, l'unico nodo che può effettivamente comunicare è però solo il primo, che dovrà dunque inviare il messaggio "1011", dove gli ultimi 2 bit rappresentano il dato da trasmettere (e quindi 3 in decimale/esadecimale) e i primi due bit, rappresentano la destinazione, vale a dire il nodo 2. Infatti, come è possibile vedere dal test riportato in figura 10.5, nei primi 20 ns, sul valore di uscita U_2 (cioè appunto il nodo 2) è riportato il valore 3.

In maniera del tutto analoga abbiamo poi effettuato gli altri 3 casi di test, visibili tutti in figura 10.5.

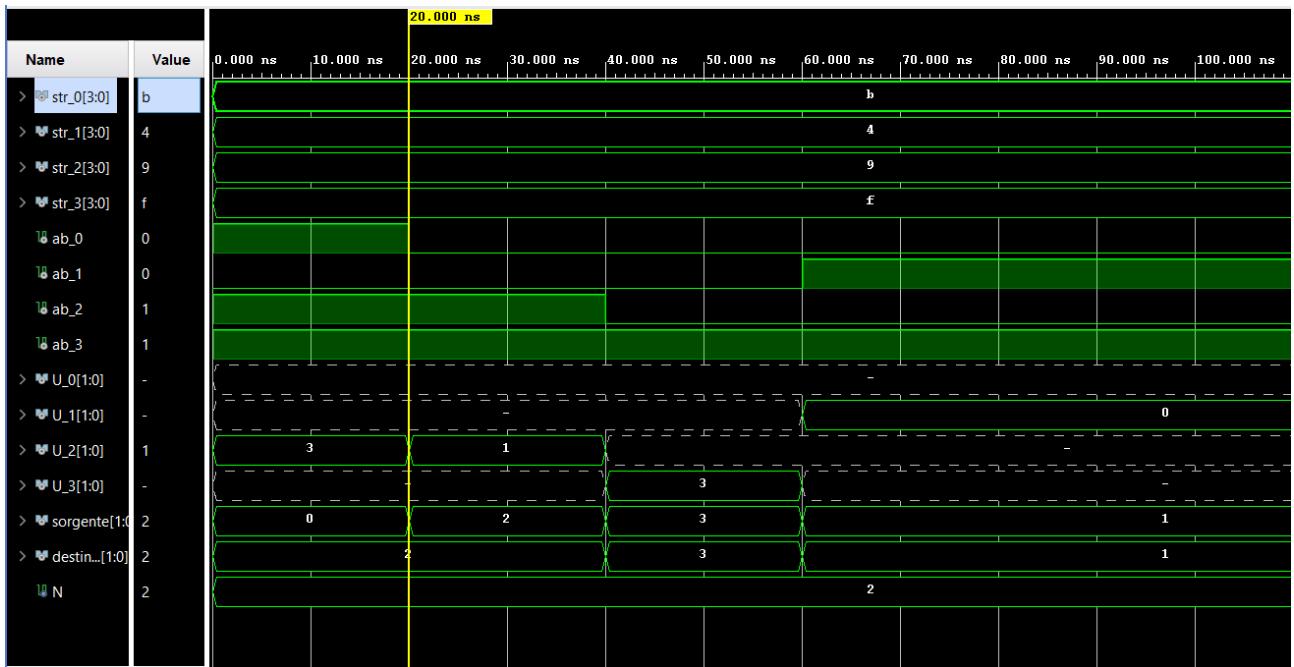


Figura 10.5: Testbench

10.3: RETE DI 8 NODI

Per la risoluzione del punto b dell'esercizio abbiamo modificato lo switch, introducendo all'interno del componente una logica di gestione dei messaggi. Il componente è stato realizzato in maniera strutturale, dalla composizione di un multiplexer, un demultiplexer e un arbitro. In questo caso, dovendo gestire 8 nodi, erano necessari 3 bit per la sorgente e 3 per la destinazione; quindi, il messaggio inviato è stato definito come visibile in figura 10.6:



Figura 10.6: Dimensione del pacchetto

L'intero pacchetto entra nello switch, e quindi attraversa il mux per uscire poi dal demux.

Ancora una volta il multiplexer e il demultiplexer sono stati utilizzati per instradare il pacchetto in base al valore della sorgente e della destinazione, l'arbitro invece è stato il componente realizzato per gestire i conflitti. In questo caso, si è deciso di gestire il conflitto, causato da una collisione, con la perdita di uno dei messaggi; la logica adottata è stata quella di concatenare su un segnale, il bit più significativo della destinazione dei messaggi in arrivo allo switch così da definire quale dei messaggi dovesse essere effettivamente trasmesso. In particolare si è deciso di mettere in uscita il messaggio della prima linea se il valore concatenato era uno dei seguenti : "00" or x="01" or x="0-" or x="1-" (dove il don't care serviva per gestire anche quei switch in cui non si verificavano collisioni, e dunque il quel caso il messaggio che doveva passare era l'unico che effettivamente arrivava al

componente), oppure, in caso contrario, ovvero in uno dei seguenti casi "11" or x="10" or x="-0" or x="-1", il messaggio messo in uscita era quello sulla seconda linea.

Per gestire l'indirizzo della destinazione, che è quello che determina il percorso che il messaggio deve compiere per giungere al nodo destinazione, è stato effettuata una rotazione sui bit che definiscono la destinazione, in modo che ad ogni nuovo switch che si incontra lungo il percorso, il valore del bit, preso in esame, cambia, analizzando così tutti i bit della destinazione, da quello più significativo a quello meno significativo. Alla fine del percorso, quando si verifica l'ultima rotazione, il pacchetto ritorna nella forma di partenza, e quindi è esattamente identico a quello inviato dal nodo sorgente.

```

architecture Dataflow of arb is

signal x: std_logic_vector(1 downto 0);

begin

x<= A0(4) & A1(4);--bit più significativi della destinazioni

src<= '0' when x="00" or x="01" or x="0-" or x="1-" else
'1' when x="11" or x="10" or x="-0" or x="-1" else
'-';

dst<= A0(4) when x=="00" or x=="01" or x=="0-" or x=="1-" else
A1(4) when x=="11" or x=="10" or x=="-0" or x=="-1" else
'-';

U0<= A0(7 downto 5) & A0(3 downto 2) & A0(4) & A0(1 downto 0);
U1<= A1(7 downto 5) & A1(3 downto 2) & A1(4) & A1(1 downto 0);

end Dataflow;

```

Lo switch è stato così realizzato:

```

entity switch is
generic(
  N: integer :=2
);
Port (
  A0: in std_logic_vector(6+(N-1) downto 0);
  A1: in std_logic_vector(6+(N-1) downto 0);
  B0:out std_logic_vector(6+(N-1) downto 0);
  B1: out std_logic_vector(6+(N-1) downto 0)
);
end switch;

architecture Structural of switch is

component mux2_1 is
generic(
  N: integer :=2
);
Port (
  A0: in std_logic_vector(6+(N-1) downto 0);
  A1: in std_logic_vector(6+(N-1) downto 0);
  src: in std_logic;
  U: out std_logic_vector(6+(N-1) downto 0)
);
end component;

```

```

component demux1_2  is
generic(
  N: integer :=2
);
Port (
  I: in std_logic_vector(6+(N-1) downto 0);
  dst: in std_logic;
  B0:out std_logic_vector(6+(N-1) downto 0);
  B1: out std_logic_vector(6+(N-1) downto 0)
);
end component;

component arb is
generic(
  N: integer :=2
);
Port (
  A0: in std_logic_vector(6+(N-1) downto 0);
  A1: in std_logic_vector(6+(N-1) downto 0);
  src: out std_logic;
  dst: out std_logic;
  U0: out std_logic_vector(6+(N-1) downto 0);
  U1: out std_logic_vector(6+(N-1) downto 0)
);
end component;

signal c: std_logic_vector(6+(N-1) downto 0);
signal U0_arb: std_logic_vector(6+(N-1) downto 0);
signal U1_arb: std_logic_vector(6+(N-1) downto 0);
signal src_arb: std_logic;
signal dst_arb: std_logic;

begin

a: arb port map(
  A0=>A0,
  A1=>A1,
  src=>src_arb,
  dst=> dst_arb,
  U0=>U0_arb,
  U1=>U1_arb
);

m: mux2_1 port map(
  A0=>U0_arb,
  A1=>U1_arb,
  src=>src_arb,
  U=>c
);

d: demux1_2 Port map(
  I=>c,
  dst=> dst_arb,
  B0=>B0,
  B1=>B1
);

end Structural;

```

Per completare la realizzazione, e quindi realizzare l'omega network che interconnettesse gli 8 nodi della sorgente con quelli della destinazione, sono stati sfruttati 4 switch interconnessi secondo lo schema del perfect shuffling.

```

entity rete_di_switch is
generic(
  N: integer :=2
);
Port (
  S0, S1, S2, S3, S4, S5, S6, S7: in std_logic_vector(6+(N-1) downto 0);
  B0, B1, B2, B3, B4, B5, B6, B7:out std_logic_vector(6+(N-1) downto 0)
);
end rete_di_switch;

architecture Structural of rete_di_switch is

component switch is
generic(
  N: integer :=2
);
Port (
  A0: in std_logic_vector(6+(N-1) downto 0);
  A1: in std_logic_vector(6+(N-1) downto 0);
  B0:out std_logic_vector(6+(N-1) downto 0);
  B1: out std_logic_vector(6+(N-1) downto 0)
);
end component;

signal U0,U1,U2,U3, U4, U5, U6, U7:std_logic_vector(6+(N-1) downto 0);
signal X0,X1,X2,X3, X4, X5, X6, X7:std_logic_vector(6+(N-1) downto 0);

begin

SW1_1: switch Port map (
  A0=>S0,
  A1=>S1,
  B0=>U0,
  B1=>U1
);

SW1_2: switch Port map (
  A0=>S2,
  A1=>S3,
  B0=>U2,
  B1=>U3
);

SW1_3: switch Port map (
  A0=>S4,
  A1=>S5,
  B0=>U4,
  B1=>U5
);

SW1_4: switch Port map (
  A0=>S6,
  A1=>S7,
  B0=>U6,
  B1=>U7
);


```

```

SW2_1: switch Port map (
    A0=>U0,
    A1=>U4,
    B0=>X0,
    B1=>X1
);

SW2_2: switch Port map (
    A0=>U1,
    A1=>U5,
    B0=>X2,
    B1=>X3
);

SW2_3: switch Port map (
    A0=>U2,
    A1=>U6,
    B0=>X4,
    B1=>X5
);

SW2_4: switch Port map (
    A0=>U3,
    A1=>U7,
    B0=>X6,
    B1=>X7
);

SW3_1: switch Port map (
    A0=>X0,
    A1=>X4,
    B0=>B0,
    B1=>B1
);

SW3_2: switch Port map (
    A0=>X1,
    A1=>X5,
    B0=>B2,
    B1=>B3
);

SW3_3: switch Port map (
    A0=>X2,
    A1=>X6,
    B0=>B4,
    B1=>B5
);

SW4_4: switch Port map (
    A0=>X3,
    A1=>X7,
    B0=>B6,
    B1=>B7
);

end Structural;

```

CAPITOLO 11: MOLTIPLICATORE DI BOOTH

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

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

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

11.1: DESCRIZIONE

Realizzeremo in questo capitolo il moltiplicatore di Booth per effettuare la moltiplicazione tra due valori. Questo moltiplicatore, in particolare, usa una propria codifica che ci permette di rappresentare un numero binario in complementi a due in un numero le cui cifre sono terne.

La notazione in complemento a due viene usata per rappresentare numeri con segno. Nei numeri negativi in complemento a due, ogni cifra del numero ha un peso dato dalla notazione posizionale, ma la cifra più significativa (che viene moltiplicata per 2^{n-1}) ha peso negativo, ovvero -2^{n-1} .

Dato un numero X, per ottenere $-X$ in complemento a due dobbiamo complementare tutti i bit e aggiungere 1. Se il numero è positivo (la cifra di peso più significativo vale 0) possiamo scriverlo attraverso la sommatoria:

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

Se il numero è negativo abbiamo la seguente espressione:

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

Riportiamo di seguito la costruzione della successione y, che risulterà ovviamente una successione non binaria. Dato un numero intero X la cui rappresentazione in complementi a 2 è:

$$x_{n-1} x_{n-2} \dots x_0$$

e ponendo

$$y_0 = -x_0$$

$$y_1 = -x_1 + x_0$$

$$y_{n-1} = -x_{n-1} + x_{n-2}$$

Moltiplicando la prima equazione per 2^0 , la seconda per 2^1 e così via e poi sommandole otteniamo:

$$y_{n-1}2^{n-1} + y_{n-2}2^{n-2} + \dots + y_02^0 = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_02^0$$

dove l'espressione, alla sinistra dell'uguale nell'equazione, è la rappresentazione in notazione posizionale e quella a destra è la rappresentazione in complementi a due.

Di seguito scriviamo:

$$Y = \sum_{i=0}^{n-1} y_i 2^i = -2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i = X$$

La rappresentazione di Booth di un numero X si può ottenere prendendo ciascuna coppia di bit adiacenti $x_j x_{j-1}$ e: alla coppia “00” oppure “11” corrisponde una codifica di 0, alla coppia “01” corrisponde +1 e a “10” corrisponde -1.

In corrispondenza di ogni coppia abbiamo un diverso comportamento, al fine di ottenere il risultato corretto della moltiplicazione:

- 00,11: effettuiamo solo lo shift a destra di una posizione
- 01: effettuiamo prima l'addizione e poi lo shift a destra
- 10: effettuiamo prima la sottrazione e poi lo shift a destra

11.2: FUNZIONAMENTO

In questo paragrafo rappresentiamo lo schema realizzato per il moltiplicatore di booth. Divideremo l'architettura in parte operativa e di controllo (realizzata attraverso un automa).

Riportiamo di seguito lo schema generale:

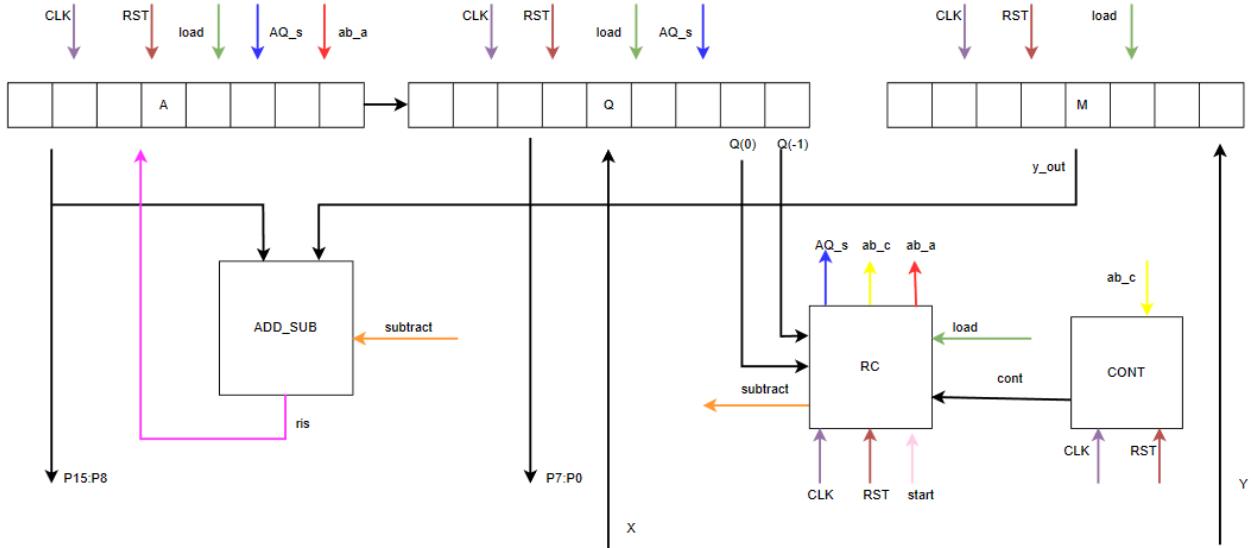


Figura 11.1: Moltiplicatore di Booth

Per realizzare il moltiplicatore usufruiremo di un registro M(8 bit), in cui inseriamo il valore del moltiplicando, due shift register A(8 bit) e Q(9 bit). Un blocco che realizza l'addizione o la somma, un contatore e una rete di controllo.

Il registro A, inizialmente, viene inizializzato a 0. In esso verranno memorizzati i risultati parziali dell'add_sub e, alla fine dell'operazione, conterrà una parte del risultato, precisamente la metà. Nel registro Q, all'inizio viene posto il valore del moltiplicatore. A questo registro viene aggiunto un bit in più rispetto al numero dei bit usati per rappresentare il moltiplicatore. Questo bit aggiuntivo lo poniamo a 0.

Il contatore ha il compito di indicare alla rete di controllo la fine del procedimento. Dato la moltiplicazione, nel nostro esempio, di vettori di 8 bit, il termine della procedura si verifica quando il conteggio è pari a 8. Tutte le coppie Q(0)Q(-1) sono state analizzate e sono stati effettuati 8 shift.

L'addizionatore/sottrattore calcola la somma o la sottrazione sulla base del segnale di subtract proveniente dalla rete di controllo. Il blocco è costituito dal componente RCA (Ripple Carry Adder), secondo lo schema in figura 11.2. Nel componente RCA entrano due segnali, X e Y. In realtà, come possiamo notare, il segnale Y viene messo in xor con il segnale di subtract e poi posto in ingresso al ripple carry adder. Questo segnale ci permette di porre il carry entrante a uno e di fare il negato di Y. Se vale zero allora la xor tra 0 e Y fa Y, mentre se vale 1 la xor tra Y e 1 fornisce il suo complemento. Se subtract vale uno allora vuol dire che vogliamo fare la sottrazione, se vale zero vogliamo fare la somma.

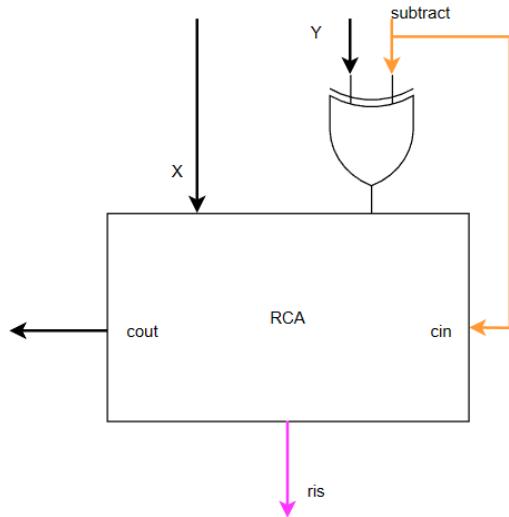


Figura 11.2 ADD_SUB

Il RCA è composto dai Full Adder. Riportiamo di seguito lo schema su 8 bit:

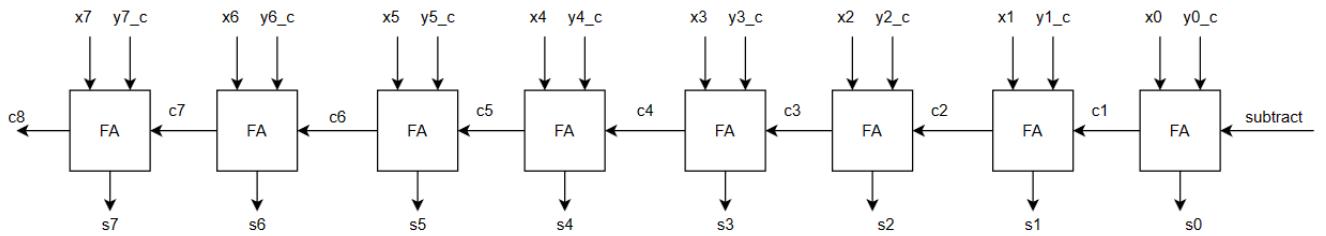


Figura 11.3: RCA

11.2.1: Automa

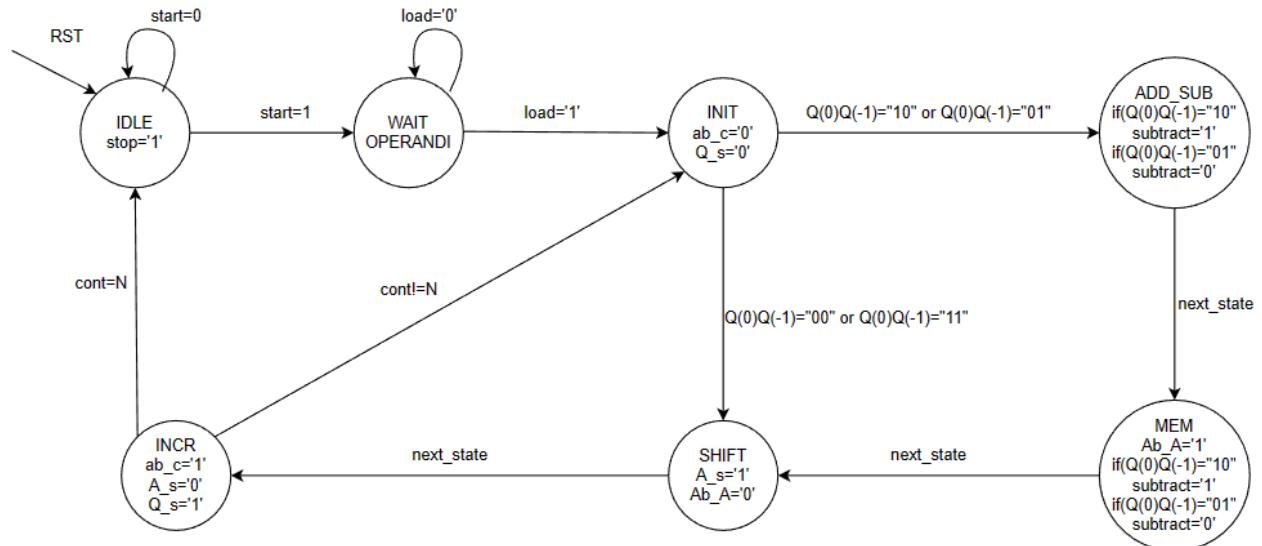


Figura 11.4: Automa_RC

L'unità di controllo prevede un primo stato di IDLE, in cui rimaniamo in attesa finché non riceviamo lo start della macchina. Una volta ricevuto, aspettiamo (nello stato WAIT OPERANDI) che venga dato il segnale di load, necessario per caricare il moltiplicatore e il moltiplicando.

Dallo stato di INIT: se $Q(0)Q(-1)$ è uguale a "00" oppure a "11" andiamo nello stato di SHIFT direttamente, altrimenti passiamo prima per lo stato ADD_SUB. In quest'ultimo stato, bisogna alzare il segnale di subtract oppure metterlo a zero a seconda di sé dobbiamo fare la somma o la sottrazione. Il caso di sottrazione si verifica quando gli ultimi due bit del registro Q sono "10", mentre quando la coppia è "01" si effettua la somma. Dopodiché passiamo nello stato MEM, in cui abilitiamo lo shift register "A" a memorizzare il valore della somma. In questo stato manteniamo ancora invariato il segnale di subtract rispetto allo stato precedente, per evitare che prima di memorizzare il dato in A possa cambiare l'uscita del sommatore e caricare nel registro un valore diverso da quello atteso. Infatti, se il segnale di subtract viene posto a '1' nello stato ADD_SUB e nello stato successivo si riazzerasse, il sommatore vedrebbe il cambiamento del segnale di ingresso e provvederebbe a calcolare la somma. L'uscita "ris" sarebbe cambiata prima che il valore corretto venga memorizzato in A. A partire da MEM il prossimo stato sarà SHIFT.

Nello stato SHIFT alziamo l'abilitazione per lo shift di A, il cui valore shiftato sarà posto in ingresso allo shift register Q. Nello stato INCR, abilitiamo lo shift di Q e alziamo l'abilitazione del contatore. A questo punto, se il valore del conteggio è diverso da N ripetiamo il procedimento a partire da INIT, altrimenti, se siamo arrivati a N ritorniamo in IDLE nell'attesa di un eventuale start, per procedere con una nuova moltiplicazione.

11.3: IMPLEMENTAZIONE VHDL

11.3.1: SHIFT REGISTER A

```

entity A is port(
    CLK, RST: in std_logic;
    X : in std_logic_vector(7 downto 0);
    load : in std_logic;
    Ab_a : in std_logic;--abilito A a memorizzare il risultato di add_sub
    A_s : in std_logic;--abilito A a shiftare
    P : out std_logic_vector(7 downto 0);
    usc_shift: out std_logic
);
end A;

architecture Behavioral of A is

signal tmp: std_logic_vector(7 downto 0);
signal c: std_logic;

```

```

begin
  process(CLK)
  begin
    if (rising_edge(CLK)) then
      if (RST='1' or load ='1') then
        tmp <= (others=>'0');
        c<=tmp(0);
      else if(Ab_a='1') then
        tmp <= X;
        c<=tmp(0);
      else
        if(A_s='1') then
          c<=tmp(0);
          tmp(7 downto 0)<= tmp(7) & tmp(7 downto 1);
        end if;
      end if;
    end if;
  end process;
  P <= tmp;
  usc_shift<=c;
end Behavioral;

```

Questo shift register (descritto in maniera comportamentale), come possiamo notare, risulta essere a caricamento parallelo. Prevede, poi, due uscite. Un'uscita seriale che viene posta in ingresso a Q e, un'uscita parallela riportante tutti gli 8 bit.

Il segnale X in ingresso corrisponde al valore che il registro A deve memorizzare quando riceve l'abilitazione "AB_a" di caricamento. Il segnale di load, come già detto in precedenza, consente di inizializzare A con tutti zeri.

Notiamo che, quando viene effettuato lo shift verso destra, poniamo come bit in posizione 7, il bit che c'era precedentemente, ovvero prima di effettuare lo shift. È questo, quindi, il valore che poniamo in testa al registro durante l'operazione.

11.3.2: Implementazione Q

```

entity Q is port(
  CLK, RST: in std_logic;
  X : in std_logic_vector(7 downto 0);
  usc_a: in std_logic; --uscita del registro A
  load : in std_logic;
  Q_s : in std_logic;--abilito Q a shiftare
  P : out std_logic_vector(8 downto 0)
);
end Q;

```

```

architecture Behavioral of Q is

signal tmp: std_logic_vector(8 downto 0);

begin
    process(CLK)
    begin
        if (rising_edge(CLK)) then
            if (RST='1') then
                tmp <= (others=>'0');
            else if(load='1') then
                tmp <= X & '0';
            else
                if(Q_s='1') then
                    tmp(8 downto 0)<= usc_a & tmp(8 downto 1);
                end if;
            end if;
        end if;
    end process;
    P <= tmp;
end Behavioral;

```

Il segnale X è il segnale di ingresso con cui lo shift register viene caricato quando viene attivato il segnale di load. X viene concatenato con lo '0' e poi posto nel segnale tmp. Il signal tmp sarà poi posto nel segnale di uscita dello shift register.

Quando il segnale di abilitazione allo shift si alza, viene posto, come detto in precedenza, l'uscita di A nel bit più significativo.

11.3.3: Implementazione M

```

entity M is Port (
    clk: in std_logic;
    rst: in std_logic;
    Y: in std_logic_vector(7 downto 0);
    load: in std_logic;
    y_out: out std_logic_vector(7 downto 0)
);
end M;

architecture Behavioral of M is

begin
    process(clk)
    begin
        if(rising_edge(clk)) then
            if(rst='1') then
                y_out<="00000000";
            else
                if(load='1') then
                    y_out<=Y;
                end if;
            end if;
        end if;
    end process;
end Behavioral;

```

11.3.4: Implementazione Full Adder

Per il full adder, abbiamo usato un'implementazione di tipo dataflow:

```

entity FA is
  port(
    a: in std_logic;
    b: in std_logic;
    cin: in std_logic;
    cout: out std_logic;
    s: out std_logic
  );
end FA;

architecture dataflow of FA is

begin

  s<= a xor b xor cin;
  cout<= (a and b) or (cin and (a xor b));

end dataflow;

```

11.3.5: Implementazione RCA

```

entity rca is
  port(
    X: in std_logic_vector(7 downto 0);
    Y: in std_logic_vector(7 downto 0);
    c_in: in std_logic;
    c_out: out std_logic;
    ris: out std_logic_vector(7 downto 0)
  );
end rca;

architecture structural of rca is
  component FA is
  port(
    a: in std_logic;
    b: in std_logic;
    cin: in std_logic;
    cout: out std_logic;
    s: out std_logic
  );
  end component;

  signal temp: std_logic_vector(7 downto 0);

```

```

begin

FA_0: FA port map(
    a=>X(0),
    b=> Y(0),
    cin=> c_in,
    cout=> temp(0),
    s=> ris(0)
);

FA_1_to_7: FOR i IN 1 TO 7 GENERATE
    RA: FA port map(
        a=>X(i),
        b=> Y(i),
        cin=>temp(i-1),
        cout=> temp(i),
        s=> ris(i)
    );
END GENERATE;

c_out<=temp(7);

end structural;

```

Il signal temp che abbiamo dichiarato servirà a contenere tutti i riporti uscenti dai Full Adder.

Esso è composto, in modo strutturale, da 8 full adder. Per il primo diamo in ingresso i bit meno significativi di X e Y e poniamo la sua uscita nel bit meno significativo dell'uscita "ris" dell'RCA. L'uscita "cout" viene posta anch'essa nel bit meno significativo del segnale temp.

Per i successivi FA (da 1 a 7) abbiamo usato un ciclo for per realizzare i rispettivi port map.

11.3.6: Implementazione ADD_SUB

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ADD_SUB is
    port(
        X: in std_logic_vector(7 downto 0);
        Y: in std_logic_vector(7 downto 0);
        cin: in std_logic;--subtract
        ris: out std_logic_vector(7 downto 0);
        cout: out std_logic
    );
end ADD_SUB;

```

```

architecture structural of ADD_SUB is
    component rca is
        port(
            X: in std_logic_vector(7 downto 0);
            Y: in std_logic_vector(7 downto 0);
            c_in: in std_logic;
            c_out: out std_logic;
            ris: out std_logic_vector(7 downto 0)
        );
    end component;

    signal complemento: std_logic_vector(7 downto 0);

begin

comp: FOR i IN 0 TO 7 GENERATE
    complemento(i)<=Y(i) xor cin;
END GENERATE;

RA: rca port map(
    X=>X,
    Y=> complemento,
    c_in=>cin,
    c_out=> cout,
    ris=>ris
);

end structural;

```

Il blocco ADD_SUB è stato realizzato seguendo lo schema in figura 11.2.

Esso prevede in ingresso il segnale X, Y, il segnale cin di subtract e in uscita la somma “ris” e il riporto “cout”.

È composto in modo strutturale dal blocco RCA. Notiamo che, prima di effettuare il port map è necessario effettuare il complemento di Y attraverso la xor bit a bit con “cin”. Ovviamente, come già affermato in precedenza, ricordiamo che se il subtract è 0, il signal “complemento” sarà uguale al segnale Y.

11.3.7: Implementazione rete di controllo

```
entity RC is

Port (
    clk: in std_logic;
    rst: in std_logic;
    cont: in std_logic_vector(2 downto 0);
    Q: in std_logic_vector(1 downto 0);
    start: in std_logic;
    load: in std_logic;
    A_s: out std_logic;
    Q_s: out std_logic;
    ab_c: out std_logic;
    ab_a: out std_logic;
    subtract: out std_logic;
    stop: out std_logic
);
end RC;

architecture Behavioral of RC is
type state is(
    IDLE, WAIT_OPERANDI, INIT, ADD_SUB, MEM, SHIFT, INCR
);
signal current_state, next_state: state;

begin

ag: process(clk)
begin
    if(rising_edge(clk)) then
        if(rst='1') then
            current_state<=IDLE;
        else
            current_state<=next_state;
        end if;
    end if;
end process;

c: process(current_state, start, load, Q, cont)
begin

A_s<='0';
Q_s<='0';
ab_c<='0';
ab_a<='0';
subtract<='0';
stop<='0';


```

```

case current_state is

when IDLE =>
    stop<='1';

    if(start='0') then
        next_state <=IDLE;
    else
        if(start='1') then
            next_state <= WAIT_OPERANDI;
        end if;
    end if;

when WAIT_OPERANDI =>

    if(load='0') then
        next_state <=WAIT_OPERANDI;
    else
        if(load='1') then
            next_state <= INIT;
        end if;
    end if;

when INIT =>

    if(Q="00" or Q="11") then
        next_state<=SHIFT;
    else
        if(Q="10" or Q="01") then
            next_state<=ADD_SUB;
        end if;
    end if;

when ADD_SUB =>
    if(Q="10") then
        subtract<='1';
    else
        if(Q="01") then
            subtract<='0';
        end if;
    end if;

    next_state<=MEM;

when MEM =>
    ab_a<='1';
    if(Q="10") then
        subtract<='1';
    else
        if(Q="01") then
            subtract<='0';
        end if;
    end if;

```

```

    next_state<=SHIFT;

when SHIFT =>
  A_s<='1';

  next_state<=INCR;

when INCR=>
  ab_c<='1';
  Q_s<='1';

  if(cont!="111") then
    next_state<=INIT;
  else
    next_state<=IDLE;
  end if;

end case;
end process;

end Behavioral;

```

L'implementazione della rete di controllo procede come quanto descritto in precedenza.

Come ulteriore uscita dalla rete di controllo, oltre ai segnali già citati, possiamo notare la presenza di un segnale di stop. Questo segnale viene alzato nello stato di IDLE. Esso servirà a gestire l'uscita che andremo a visualizzare.

L'uscita del moltiplicatore di Booth sarà data dal concatenamento degli 8 bit di A e degli 8 bit di Q (il bit meno significativo non farà parte del prodotto finale). Per una maggiore chiarezza nella visualizzazione, preferiremmo far visualizzare il solo risultato della moltiplicazione e non tutti i risultati intermedi. Per questa ragione, alla fine dell'operazione, ritorniamo nello stato di IDLE e alziamo il flag di stop. In questo modo l'uscita da visualizzare sarà aggiornata al risultato corrente. Se avviamo una nuova moltiplicazione, con nuovi valori, alla fine del prodotto verrà visualizzato il nuovo risultato ottenuto.

11.3.8: Implementazione Moltiplicatore di Booth

Il moltiplicatore di Booth riceve in ingresso: clock, reset, start e load (segnale, rispettivamente, di inizio e caricamento, che verranno posti in ingresso alla rete di controllo), X_q (valore del moltiplicatore), Y_m (valore del moltiplicando), Pt (prodotto finale su 16 bit), stop (segnale di uscita della rete di controllo).

Utilizzando un approccio strutturale, dichiariamo nell'architettura tutti i componenti della parte operativa e la rete di controllo.

```
entity Multiplier_Booth is Port (
    clk: in std_logic;
    rst: in std_logic;
    load : in std_logic;
    start: in std_logic;
    X_q: in std_logic_vector(7 downto 0);--inizializzazione di Q
    Y_m: in std_logic_vector(7 downto 0);--inizializzazione di M
    Pt: out std_logic_vector(15 downto 0);
    stop: out std_logic
);
end Multiplier_Booth;

architecture Structural of Multiplier_Booth is

component A is port(
    CLK, RST: in std_logic;
    X : in std_logic_vector(7 downto 0);
    load : in std_logic;
    Ab_a : in std_logic;--abilito A a memorizzare il risultato di add_sub
    A_s : in std_logic;--abilito A a shiftare
    P : out std_logic_vector(7 downto 0);
    usc_shift: out std_logic
);
end component;

component Q is port(
    CLK, RST: in std_logic;
    X : in std_logic_vector(7 downto 0);
    usc_a: in std_logic; --uscita del registro A
    load : in std_logic;
    Q_s : in std_logic;--abilito Q a shiftare
    P : out std_logic_vector(8 downto 0)
);
end component;

component M is Port (
    clk: in std_logic;
    rst: in std_logic;
    Y: in std_logic_vector(7 downto 0);
    load: in std_logic;
    Y_out: out std_logic_vector(7 downto 0)
);
end component;

component cont is
    Port (
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        ab_c : in STD_LOGIC; --questo è l'enable del clock, insieme danno l'impulso di conteggio
        conteggio : out STD_LOGIC_VECTOR (2 downto 0)
    );
end component;
```

```

component ADD_SUB is
  port(
    X: in std_logic_vector(7 downto 0);
    Y: in std_logic_vector(7 downto 0);
    cin: in std_logic; --subtract
    ris: out std_logic_vector(7 downto 0);
    cout: out std_logic
  );
end component;

component RC is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    cont: in std_logic_vector(2 downto 0);
    Q: in std_logic_vector(1 downto 0);
    start: in std_logic;
    load: in std_logic;
    A_s: out std_logic;
    Q_s: out std_logic;
    ab_c: out std_logic;
    ab_a: out std_logic;
    subtract: out std_logic;
    stop: out std_logic
  );
end component;

signal carica_a: std_logic;
signal shift_a: std_logic;
signal shift_q: std_logic;
signal u_shift: std_logic;
signal u_M: std_logic_vector(7 downto 0);
signal ab_cont: std_logic;
signal u_cont: std_logic_vector(2 downto 0);
signal pp: std_logic_vector(7 downto 0);
signal qq: std_logic_vector(8 downto 0);
signal subtract1: std_logic;

begin

A1: A port map(
  CLK=>CLK,
  RST=>RST,
  X => prod_parz,
  load => load,
  Ab_a =>carica_a,
  A_s => shift_a,
  P => pp,
  usc_shift=> u_shift
);

```

```

Q1: Q port map(
    CLK=>CLK,
    RST=>RST,
    X => X_q,
    usc_a => u_shift,
    load =>load,
    Q_s => shift_q,
    P=> qq
);
M1: M Port map (
    CLK=>CLK,
    RST=>RST,
    Y=> Y_m,
    load=> load,
    y_out => u_M
);
c: cont Port map (
    CLK=>CLK,
    RST=>RST,
    ab_c => ab_cont,
    conteggio => u_cont
);
as: ADD_SUB port map(
    X => pp,
    Y =>u_M,
    cin=> subtract1,
    ris => prod_parz,
    cout=> open
);
rc1: RC Port map(
    clk=> clk,
    rst=>rst,
    cont=>u_cont,
    Q=> qq(1 downto 0),
    start=> start,
    load =>load,
    A_s=> shift_a,
    Q_s=> shift_q,
    ab_c =>ab_cont,
    ab_a =>carica_a,
    subtract =>subtract1,
    stop=>stop
);
Pt<=pp & qq(8 downto 1);

end Structural;

```

11.3.9: Implementazione gestione_uscita

```
entity gestione_uscita is
  Port (
    CLK: in std_logic;
    rst: in std_logic;
    AQ: in std_logic_vector(15 downto 0);
    stop: in std_logic;
    AQ_out: out std_logic_vector(15 downto 0)
  );
end gestione_uscita;

architecture Behavioral of gestione_uscita is

begin
  process(clk)
  begin
    if(stop='1') then
      AQ_out<=AQ;
    end if;
  end process;

end Behavioral;
```

L'entità gestione_uscita, prevede come segnale di ingresso il clock, il reset, l'ingresso AQ (che rappresenta il risultato della moltiplicazione in uscita dal Moltiplicatore di Booth) e l'ingresso di stop.

Il funzionamento, descritto in maniera behavioral, di questa entità prevede, come già accennato nel paragrafo precedente, di aggiornare la sua uscita solo in corrispondenza di un segnale di stop. In questo modo, quando le operazioni di moltiplicazione saranno concluse verrà mostrato in uscita il risultato finale.

11.3.10: Implementazione tot

```
entity tot is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    load: in std_logic;
    start: in std_logic;
    X_q: in std_logic_vector(7 downto 0);--inizializzazione di Q
    Y_m: in std_logic_vector(7 downto 0);--inizializzazione di M
    Pt1: out std_logic_vector(15 downto 0)
  );
end tot;

architecture Behavioral of tot is

component Multiplier_Booth is Port (
  clk: in std_logic;
  rst: in std_logic;
  load : in std_logic;
  start: in std_logic;
  X_q: in std_logic_vector(7 downto 0);--inizializzazione di Q
  Y_m: in std_logic_vector(7 downto 0);--inizializzazione di M
  Pt: out std_logic_vector(15 downto 0);
  stop: out std_logic
);
end component;
```

```

component gestione_uscita is
  Port (
    CLK: in std_logic;
    rst: in std_logic;
    AQ: in std_logic_vector(15 downto 0);
    stop: in std_logic;
    AQ_out: out std_logic_vector(15 downto 0)
  );
end component;

signal uscita: std_logic_vector(15 downto 0);
signal s: std_logic;

begin

mb: Multiplier_Booth Port map(
  clk=>clk,
  rst=>rst,
  load=>load,
  start=> start,
  X_q=>X_q,
  Y_m=>y_m,
  Pt=>uscita,
  stop=>s
);

gu: gestione_uscita Port map (
  clk=>clk,
  rst=>rst,
  AQ=>uscita,
  stop=>s,
  AQ_out=>Pt1
);

end Behavioral;

```

Abbiamo creato l'entità tot composta dal moltiplicatore e da gestione_uscita. Esso, sostanzialmente, prevede gli stessi ingressi e le stesse uscite del multiplier, ad eccezione dell'uscita di stop. Questa uscita viene gestita con un signal "s", per effettuare il collegamento tra uscita di stop del moltiplicatore e ingresso stop dell'altra entità. Anche l'uscita "Pt" del multiplier viene associato a un signal, per effettuarne il collegamento con gestione_uscita.

11.4: SIMULAZIONE

Riportiamo di seguito il testbench, in particolare i test effettuati:

```
test: process
begin
    rst <= '1';
    wait for 100ns; --global reset
    rst <='0';

    wait for 10ns;

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

    l<='1';
    xq<="00000010";
    ym<="00000001";
    wait for 10ns;
    l<='0';

    wait for 400 ns;

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

    l<='1';
    xq<="00001010";
    ym<="00000110";
    wait for 10ns;
    l<='0';

    wait for 400 ns;

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

    l<='1';
    xq<="11111011";
    ym<="00000011";
    wait for 10ns;
    l<='0';

    wait;
end process;
end Behavioral;
```

Il primo test effettuato prevede la moltiplicazione 2*1. Il risultato che ci aspettiamo è ovviamente 2.

Il secondo test ha come moltiplicatore 10 e come moltiplicando 6. Il risultato dovrà essere 60.

Il terzo test prevede una moltiplicazione del numero -5 per il numero 3. Il numero -5 sarà espresso in complementi a 2. Data la rappresentazione su 8 bit del numero 5 (00000101), per ottenere -5 in complemento a 2 dobbiamo complementare i bit e aggiungere 1:

00000101 -> 11111010 -> 11111011

Il risultato di quest'ultima moltiplicazione dovrà essere -15.

La rappresentazione di -15 in complemento a 2 è la seguente:

0000000000001111 (15 in binario su 16 bit) -> 111111111110000 -> 111111111110001

In notazione esadecimale, -15 risulta essere "fff1".

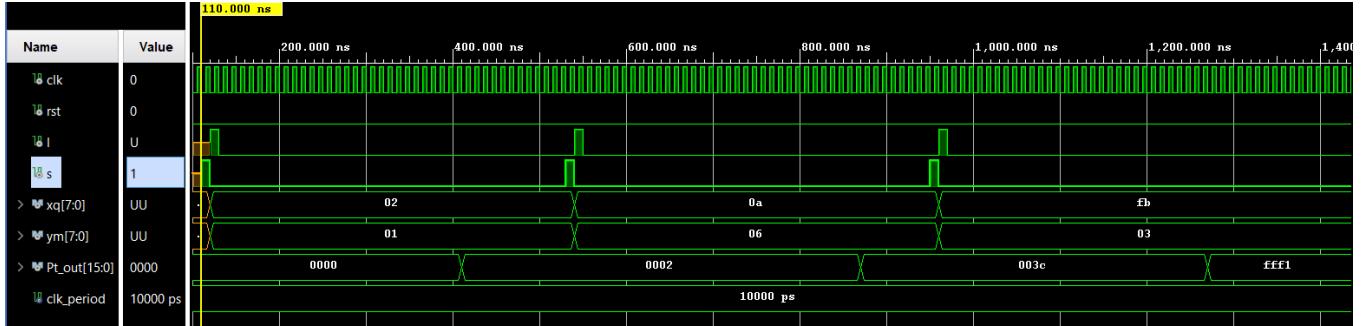


Figura 11.5: Testbench

Dati i numeri espressi in notazione esadecimale, possiamo notare dalla figura 11.5, il risultato della simulazione, che risulta essere quello atteso:

- La moltiplicazione di 02 e 01 dà come risultato 0002(su 16 bit)
- La moltiplicazione 0a(10 in decimale) * 06 dà 003c(60 in decimale)
- La moltiplicazione di fb(-5) e 03 dà fff1(-15)

11.5: Scheda

Per sintetizzare su FPGA questo componente, abbiamo deciso di utilizzare 3 bottoni(per il reset, lo start e il load), i 16 switch(8 per il moltiplicatore e 8 per il moltiplicando) e i 16 led(per visualizzare il risultato del prodotto).

Abbiamo aggiunto al componente totale realizzato e mostrato precedentemente l'automa dei debouncer per i rispettivi bottoni.

Riportiamo di seguito la struttura totale per questo punto dell'esercizio, che risulta essere, solo di poco, modificato:

```
entity tot_pt2 is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    load2: in std_logic;
    start2: in std_logic;
    X_q2: in std_logic_vector(7 downto 0);--inizializzazione di Q
    Y_m2: in std_logic_vector(7 downto 0);--inizializzazione di M
    Pt2: out std_logic_vector(15 downto 0)
  );
end tot_pt2;
```

```

architecture Behavioral of tot_pt2 is

component Multiplier_Booth is Port (
    clk: in std_logic;
    rst: in std_logic;
    load : in std_logic;
    start: in std_logic;
    X_q: in std_logic_vector(7 downto 0);--inizializzazione di Q
    Y_m: in std_logic_vector(7 downto 0);--inizializzazione di M
    Pt: out std_logic_vector(15 downto 0);
    stop: out std_logic
);
end component;

component gestione_uscita is
    Port (
        CLK: in std_logic;
        rst: in std_logic;
        AQ: in std_logic_vector(15 downto 0);
        stop: in std_logic;
        AQ_out: out std_logic_vector(15 downto 0)
    );
end component;

component B_DB is
    generic (
        CLK_period: integer := 10; -- periodo del clock in nanosec
        btn_noise_time: integer := 10000000 --durata dell'oscillazione in nanosec
    );
    Port ( RST : in STD_LOGIC;
            CLK : in STD_LOGIC;
            BTN : in STD_LOGIC;
            CLEARED_BTN : out STD_LOGIC);
end component;

signal uscita: std_logic_vector(15 downto 0);
signal s: std_logic;
signal load_b: std_logic;
signal reset_b: std_logic;
signal start_b: std_logic;

begin

bott1: B_DB Port map(
    RST =>'0',
    CLK=>clk,
    BTN =>load2,
    CLEARED_BTN=>load_b
);

bott2: B_DB Port map(
    RST =>'0',
    CLK=>clk,
    BTN =>rst,
    CLEARED_BTN=>reset_b
);

```

```

bott3: B_DB Port map(
    RST =>'0',
    CLK=>clk,
    BTN =>start2,
    CLEARED_BTN=>start_b
);

mb: Multiplier_Booth Port map(
    clk=>clk,
    rst=>reset_b,
    load=>load_b,
    start=> start_b,
    X_q=>X_q2,
    Y_m=>y_m2,
    Pt=>uscita,
    stop=>s
);

gu: gestione_uscita Port map (
    clk=>clk,
    rst=>reset_b,
    AQ=>uscita,
    stop=>s,
    AQ_out=>Pt2
);

end Behavioral;

```

CAPITOLO 12: ESERCIZIO LIBERO

Progettare ed implementare in VHDL un sistema composto da tre unità: A, B, C. All'interno dell'unità A è presente una ROM di N=8 locazioni di k=7 bit. Per ogni locazione:

- Se il bit meno significativo è 1, A invia i restanti k-1 bit al sistema B;
- Se il bit meno significativo è 0, A invia i restanti k-1 bit al sistema C.

Il componente C ha un contatore tramite cui conta il numero di messaggi di k-1 bit ricevuti. Al termine della trasmissione, A invia un segnale di terminazione ad entrambi i sistemi (rappresentato da k-1 bit alti). Ricevuto tale segnale, il sistema C invia a B il numero di messaggi ricevuti dal sistema A. Tutte le comunicazioni avvengono mediante un protocollo di handshaking completo. Si implementi il sistema A utilizzando la logica microprogrammata per l'unità di controllo.

12.1: SOLUZIONE

Per la risoluzione di questo esercizio, riportiamo, di seguito, lo schema generale delle interconnessioni (con handshaking) dei nodi A, B e C.

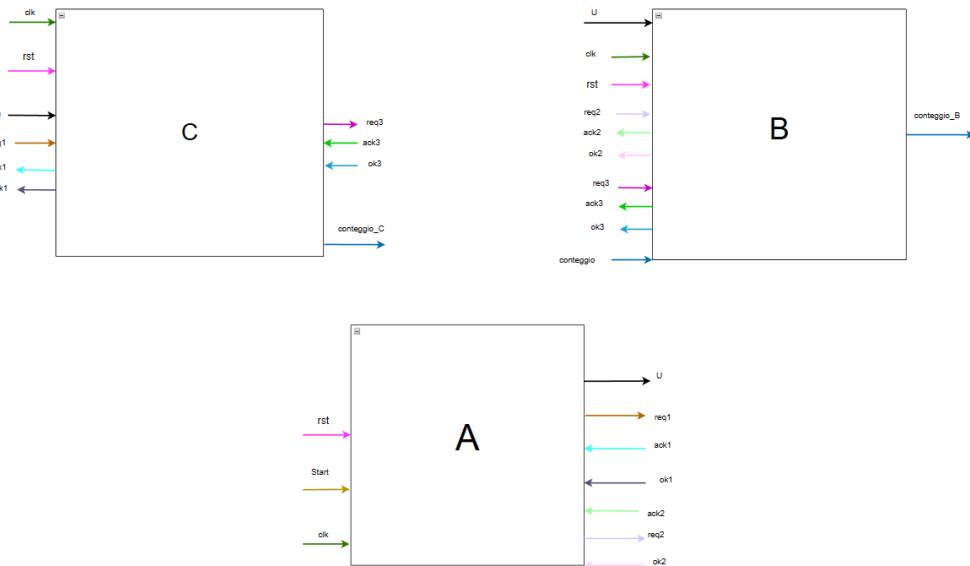


Figura 12.1: Struttura globale

I tre nodi comunicano attraverso un protocollo di handshaking completo. In particolare, il nodo A ha in uscita due richieste verso i nodi B e C (quale richiesta inviare sarà poi deciso dalla rete di controllo di A, sulla base del bit meno significativo) e riceve da quest'ultimi i rispettivi segnali di ack e ok. Inoltre, anche il nodo C instaura una comunicazione, anch'essa gestita da un protocollo di handshaking completo, verso il nodo B per comunicargli il conteggio del numero di stringhe ricevute da A.

Procediamo con l'analisi in dettaglio dei singoli componenti che costituiscono questo sistema.

12.1.1: A

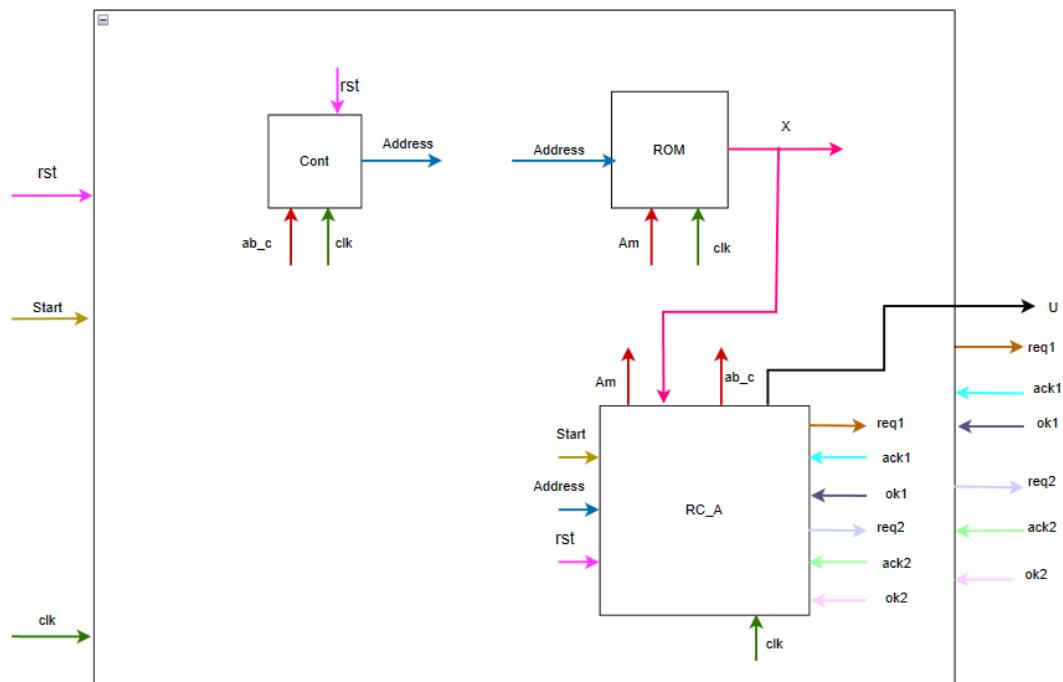


Figura 12.2: A

Il nodo A riceve come segnali di ingresso il clock, il reset, e un segnale di start per dare il via alle operazioni. Pone in uscita i segnali necessari per la comunicazione con B e C, nonché il dato U su k-1 bit.

Esso è costituito da tre elementi in particolare: una Rom, un contatore e una rete di controllo.

La rete di controllo riceve il segnale di start per iniziare la sua elaborazione, pilota le abilitazioni del contatore e della ROM, gestisce l'handshaking con i restanti 2 nodi e riceve in ingresso il valore del conteggio del contatore per terminare le sue operazioni e ritornare nello stato di IDLE, in attesa di un nuovo segnale di start.

Viene richiesto di realizzare la suddetta unità di controllo in logica microprogrammata. Riportiamo, per facilitare la spiegazione della logica usata nel risolvere questo problema, l'automa della rete per poi tradurla in logica microprogrammata nei successivi paragrafi.

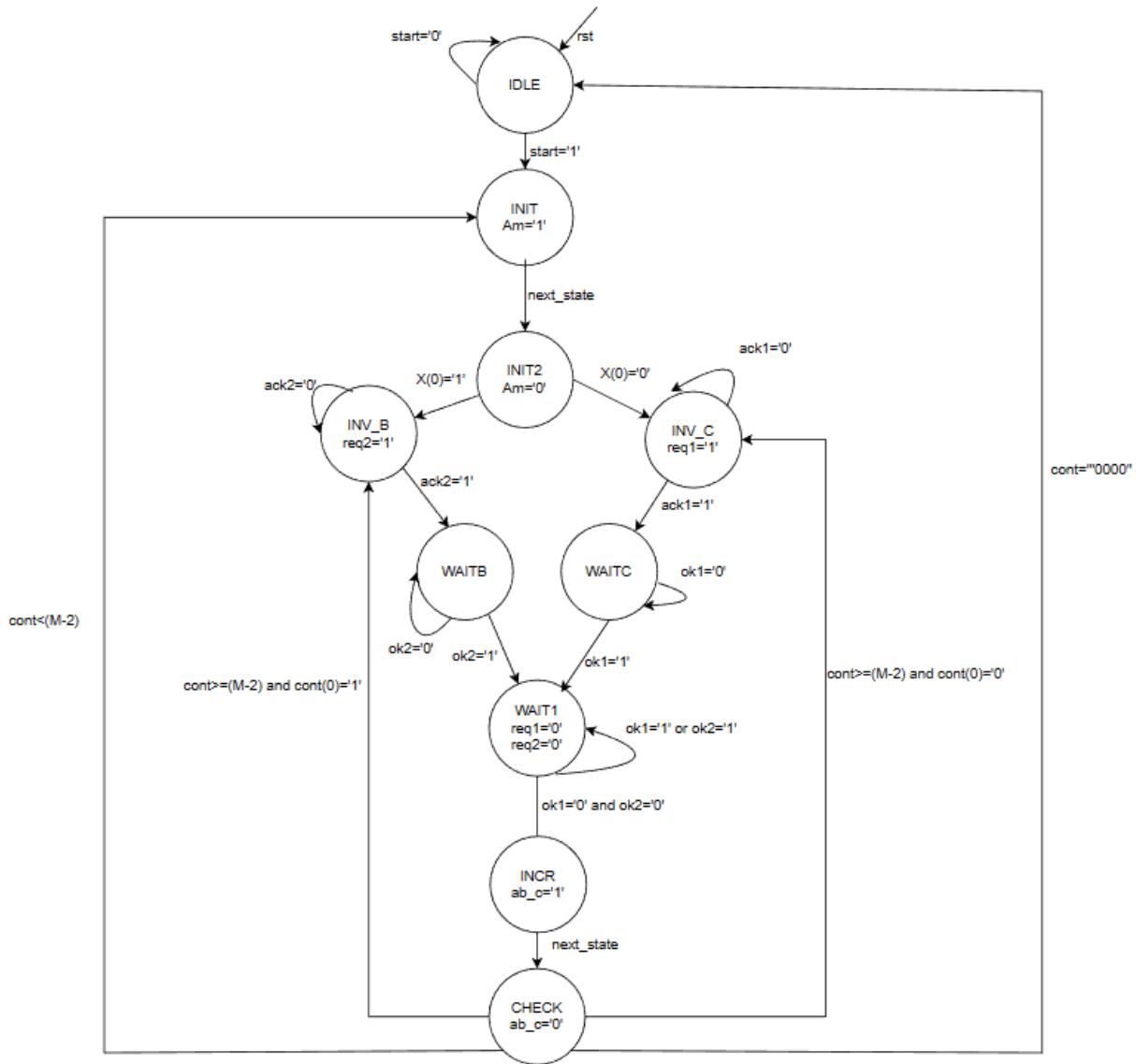


Figura 12.3: Automa di A

Nello stato di IDLE attendiamo l'arrivo del segnale di start. Una volta ricevuto, passiamo nello stato di INIT, dove abilitiamo la Rom alla lettura. Il prossimo stato in cui transitiamo è INIT2 dove abbassiamo l'abilitazione della memoria, precedentemente alzata, e effettuiamo un confronto con il bit meno significativo della stringa da inviare. Se quest'ultimo è pari a 1 allora inviamo la stringa al nodo B, alzando la rispettiva richiesta req2 e ponendoci nello stato INV_B, altrimenti inviamo il dato al nodo C, alzando la richiesta req1 e ponendoci nello stato INV_C. Negli stati INV_B e INV_C, attendiamo l'arrivo dei rispettivi ack. Nei stati WAITB, WAITC attendiamo i rispettivi ok. Nello stato WAIT1 attendiamo, invece, che gli ok siano ritornati a zero (basta che solo uno dei due sia alto per rimanere in questo stato. Notiamo che la stringa viene inviata o al nodo B o al nodo C e mai contemporaneamente. Di conseguenza solo uno degli ok, alla volta, sarà alto). Lo stato INCR alza l'abilitazione per il contatore.

Per gestire l'invio della stringa di terminazione (sia al nodo B che al nodo C, tramite handshaking) abbiamo optato di considerare un contatore modulo 9. Ponendo M=10, dobbiamo distinguere l'invio dei dati presenti nella ROM (nel nostro caso N=8) e le due stringhe di terminazione. Per fare ciò abbiamo creato uno stato di CHECK. Da questo stato, se il contatore è minore di M-2 (ovvero 10-2=8)

stiamo considerando l'invio delle 8 stringhe (da 0 a 7) della memoria di A, il cui invio deve essere scandito dal bit meno significativo del dato. Si ritorna, infatti, nello stato di INIT procedendo con la lettura dalla ROM. In caso contrario, andiamo nello stato INV_B o INV_C sulla base del bit meno significativo del conteggio del contatore. Infatti, questo bit ci assicura che le due stringhe vengano inviate ad entrambi i nodi. Esso varrà 0 quando il conteggio sarà pari a 1000 (ovvero 8), inviando la terminazione a C, e varrà 1 quando l'uscita del contatore è 1001 (ovvero 9), inviando la terminazione a B. Successivamente il contatore si resetta e si torna nello stato di IDLE.

12.1.2: B

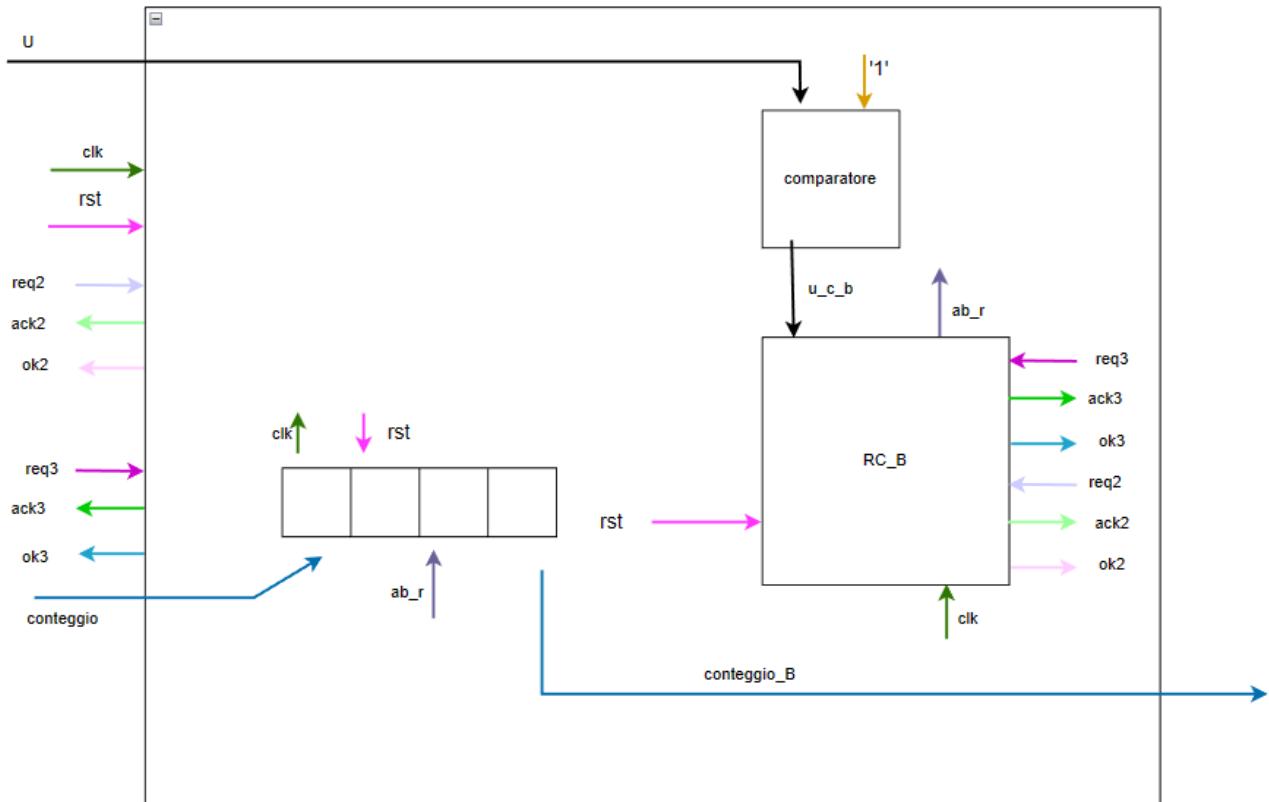


Figura 12.4: B

Il nodo B riceve, in ingresso, il clock, il reset, i segnali per la comunicazione con i nodi A e C, la stringa inviata dal nodo A e il conteggio inviato dal nodo C.

Esso è costituito da un registro (in cui memorizzare il valore ricevuto dal conteggio), da un comparatore (per riconoscere la stringa di terminazione) e dalla rete di controllo. Notiamo che, l'ingresso U prima di porlo nel comparatore, avremmo potuto salvare il suo valore in un registro e poi porlo in ingresso alla macchina per effettuare il confronto con la stringa "111111". In questo caso, abbiamo posto direttamente, come si osserva in figura 12.4, l'uscita di A nel comparatore ma è buona norma fermare il dato usando un registro che viene abilitato dalla rete di controllo.

Il valore del conteggio viene posto in uscita al nodo B e visualizzato, come successivamente mostreremo, nella simulazione. La rete di controllo gestisce l'abilitazione per il registro e i segnali per l'handshaking.

Mostriamo, di seguito, l'automa della rete di controllo:

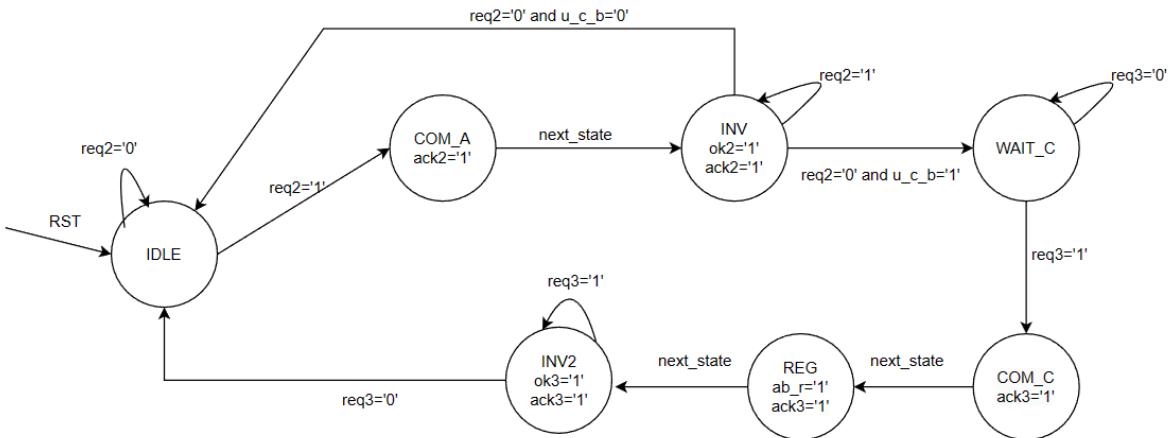


Figura 12.5: Automa_di_B

Proseguiamo col descrivere cosa succede negli stati dell'automa.

In IDLE, il nodo si pone in attesa delle stringhe provenienti dal nodo A. Una volta ricevuta si transita nello stato COM_A, in cui inviamo l'ack e, successivamente nello stato INV, in cui inviamo l'ok, avendo terminato le operazioni. Nello stato INV si attende che la richiesta di A si abbassi e a questo punto si analizza l'uscita del comparatore. Se è zero significa che ancora non abbiamo ricevuto la stringa di terminazione e, si prosegue in IDLE. Se invece è uno, significa che abbiamo ricevuto la stringa di terminazione e, prima di tornare in IDLE, ci poniamo in attesa della richiesta di comunicazione da C (che dovrà inviare il conteggio effettuato). Ricevuta “req3”, si invia il rispettivo “ack3” e si abilita il registro di B a memorizzare il valore ricevuto. Successivamente, inviamo l'ok3, aspettiamo che la richiesta si abbassi e ritorniamo in IDLE.

12.1.3: C

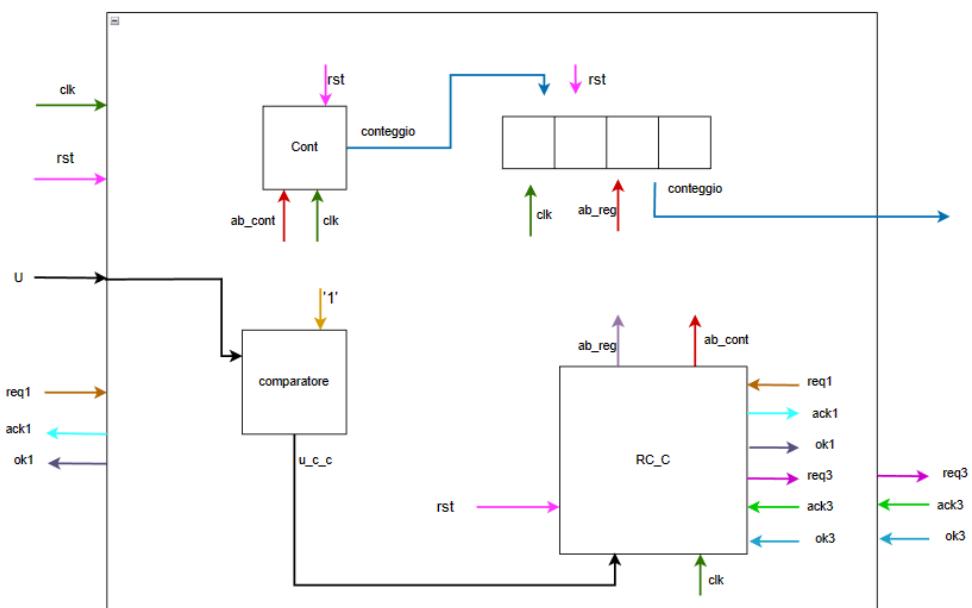


Figura 12.6: C

Il nodo C riceve in ingresso il clock, il reset, la stringa inviata da A e i segnali per gestire la comunicazione con il nodo A. In uscita pone il valore del conteggio calcolato e i segnali per effettuare l'handshaking con il nodo B, a cui deve inviare il numero di stringhe ricevute da A.

Esso è costituito dai seguenti componenti: un contatore, un comparatore, un registro e una rete di controllo. Come già detto per il nodo B, era possibile memorizzare l'ingresso U in un registro prima di porlo in ingresso al comparatore.

La rete di controllo abilita il contatore e il registro e gestisce le comunicazioni con A e B.

Anche per questo nodo procederemo con l'analizzare l'automa della rete di controllo, mostrato in figura 12.7.

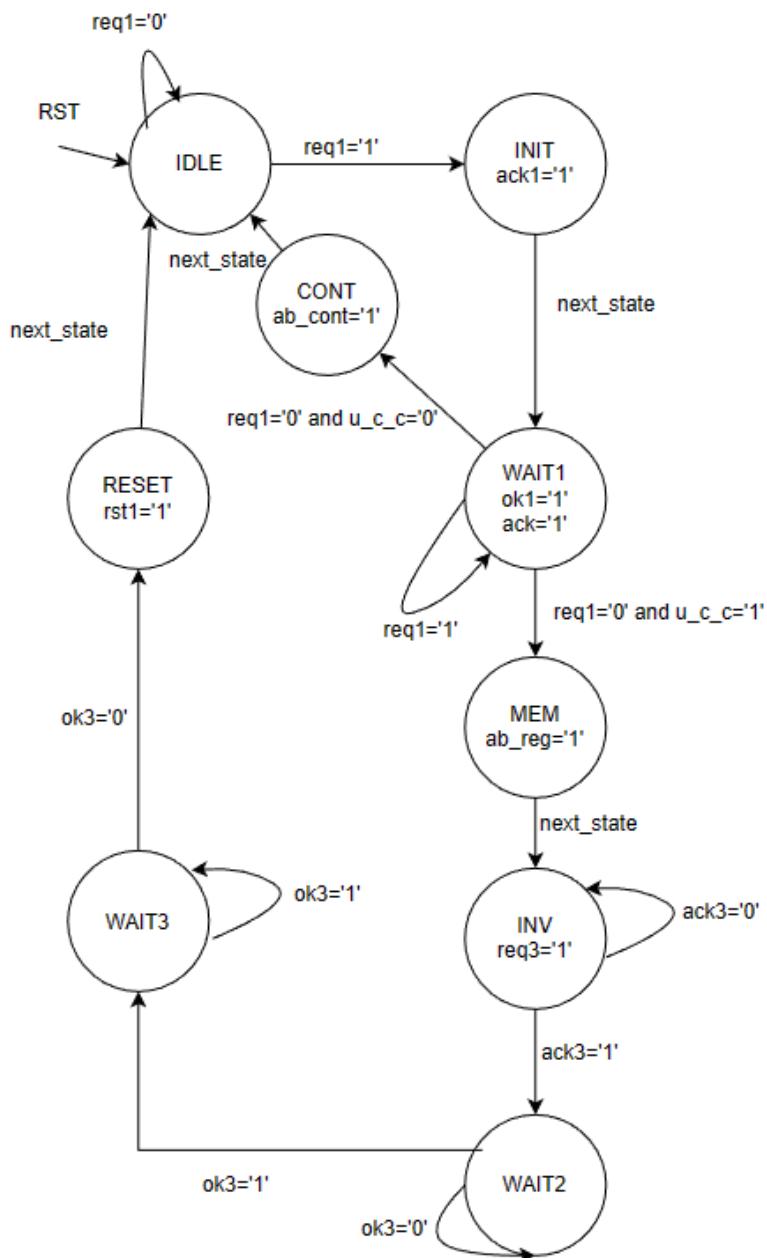


Figura 12.7: Automa_di_C

Nello stato di IDLE, C si pone in attesa della richiesta proveniente dal nodo A. Una volta ricevuta, il prossimo stato sarà INIT, in cui si alzerà il segnale di ack1. Successivamente, nello stato WAIT1, si alzerà il segnale di ok1 e si attenderà che la richiesta req1 si abbassi. Una volta che diviene 0, si può procedere in due stati diversi a seconda dell'uscita del comparatore. Se u_c_c è pari a 0 (ovvero la stringa ricevuta da A non è quella di terminazione) si procede con l'alzare l'abilitazione del conteggio (nello stato CONT) e infine si ritorna nello stato di IDLE, in attesa di una nuova richiesta. Se u_c_c è pari a 1 (la stringa ricevuta è quella di terminazione), si procede nello stato MEM, in cui abilitiamo il registro di C a memorizzare il conteggio effettuato. Successivamente si transita nello stato INV, in cui inviamo la richiesta per esprimere la volontà di voler comunicare con B e attendiamo il suo ack. Una volta ricevuto ack3 di B, C si pone in attesa del segnale ok3 con cui B esprime di aver terminato le sue operazioni. Ricevuto l'ok3, C attende ulteriormente prima di procedere, nello stato WAIT3, che B abbassi l'ok.

Lo stato RESET serve per resettare il contatore. Quest'ultimo, dovendo contare il numero di stringhe ricevute da A e non sapendo, quindi, a priori quante ne riceverà, può avere un valore di conteggio di 8, al massimo. Dopo aver ricevuto la stringa di terminazione e inviato il risultato, C può resettarlo per le future operazioni.

12.2: IMPLEMENTAZIONE VHDL

Procediamo col mostrare l'implementazione vhdl dei 3 nodi, con tutti i componenti che li costituiscono, e del sistema complessivo.

12.2.1: Nodo A

12.2.1.1: Contatore

```
entity cont is
    Port (
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        ab_c : in STD_LOGIC;
        rst1: in std_logic;
        address : out STD_LOGIC_VECTOR (3 downto 0));
end cont;

architecture Behavioral of cont is

signal c : std_logic_vector (3 downto 0) := (others => '0');

begin
```

```

address <= c;

counter_process: process(clk)
begin
    if (clk'event AND clk = '1') then
        if (rst = '1' or rstl='1') then
            c <= (others => '0');
        else
            if (ab_c='1') then
                if(c="1001") then
                    c <= (others => '0');
                else
                    c <= std_logic_vector(unsigned(c) + 1);
                end if;
            end if;
            end if;
        end if;
    end process;

end Behavioral;

```

Abbiamo realizzato un contatore modulo 9 a partire da un contatore modulo 16.

Per fare ciò, quando il componente vede alta l'abilitazione a contare, controlla prima se il conteggio è pari a nove e, nel caso, azzerà il valore di conteggio.

12.2.1.2: ROM

```

entity ROM is
    generic (
        N: integer :=8;
        K: integer :=7
    );
    port(
        CLK : in std_logic;
        Am : in std_logic;
        address : in std_logic_vector(3 downto 0);
        X: out std_logic_vector((K-1) downto 0)
    );
end ROM;

```

```

architecture behavioral of ROM is

type rom_type is array (0 to (N-1)) of std_logic_vector((K-1) downto 0);

signal ROM1 : rom_type := (
b"00000011",
b"00000010",
b"0000101",
b"1001000",
b"1010001",
b"0011001",
b"0110000",
b"0010000"
);

begin
process(CLK)
begin
  if rising_edge(CLK) then
    if (Am='1') then
      X <= ROM1(conv_integer(address));
    end if;
  end if;
end process;

end behavioral;

```

Nella ROM sono memorizzate N=8 stringhe da K=7 bit. In particolare, non considerando il bit meno significativo, abbiamo i seguenti valori:

- 1 → da inviare a B
- 1 → da inviare a C
- 2 → da inviare a B
- 36 (0x24) → da inviare a C
- 40 (0x28) → da inviare a B
- 12 (C) → da inviare a B
- 24(0x18) → da inviare a C
- 8 → da inviare a C

Verranno inviate 4 stringhe a C. Di conseguenza, ci aspetteremo di vedere nel valore di uscita del nodo B il valore 4.

12.2.1.3: MicroRom

```

entity microRom is
  port (
    pc_i: in std_logic_vector (3 downto 0); --3 bit per codificare numero degli stati
    pc_next_o : out std_logic_vector (3 downto 0);
    --tutte le uscite della rete di controllo
    Am : out std_logic;
    ab_c : out std_logic;
    req1 : out std_logic;
    req2 : out std_logic
  );
end microRom;

```

```

architecture synth of microRom is

TYPE control_word IS RECORD
    pc_next_o : std_logic_vector (3 downto 0);
    Am : std_logic;
    ab_c : std_logic;
    req1 : std_logic;
    req2 : std_logic;
END RECORD ;

CONSTANT IDLE : control_word := (
    pc_next_o => "0000",
    Am => '0',
    ab_c =>'0',
    req1 =>'0',
    req2 =>'0'
);
CONSTANT INIT : control_word := (
    pc_next_o => "0010",
    Am => '1',
    ab_c =>'0',
    req1 =>'0',
    req2 =>'0'
);
CONSTANT INIT2 : control_word := (
    pc_next_o => "0010",
    Am => '0',
    ab_c =>'0',
    req1 =>'0',
    req2 =>'0'
);

CONSTANT INV_B : control_word := (
    pc_next_o => "0011",
    Am => '0',
    ab_c =>'0',
    req1 =>'0',
    req2 =>'1'
);
CONSTANT INV_C : control_word := (
    pc_next_o => "0100",
    Am => '0',
    ab_c =>'0',
    req1 =>'1',
    req2 =>'0'
);
CONSTANT WAITB : control_word := (
    pc_next_o => "0101",
    Am => '0',
    ab_c =>'0',
    req1 =>'0',
    req2 =>'1'
);
CONSTANT WAITC : control_word := (
    pc_next_o => "0110",
    Am => '0',
    ab_c =>'0',
    req1 =>'1',
    req2 =>'0'
);

```

```

CONSTANT WAIT1 : control_word := (
  pc_next_o => "0111",
  Am => '0',
  ab_c =>'0',
  req1 =>'0',
  req2 =>'0'
);
CONSTANT INCR : control_word := (
  pc_next_o => "1001",
  Am => '0',
  ab_c =>'1',
  req1 =>'0',
  req2 =>'0'
);
CONSTANT CHECK : control_word := (
  pc_next_o => "1001",
  Am => '0',
  ab_c =>'0',
  req1 =>'0',
  req2 =>'0'
);

TYPE ROM_TYPE IS ARRAY (0 to 9) OF control_word ;
CONSTANT ROM : ROM_type := (
  0 => IDLE ,
  1 => INIT ,
  2 => INIT2 ,
  3 => INV_B ,
  4 => INV_C ,
  5 => WAITB,
  6 => WAITC,
  7 => WAIT1,
  8 => INCR,
  9 => CHECK
);

SIGNAL Controllo : control_word ;

BEGIN

  Controllo <= ROM(conv_integer(unsigned(pc_i)));
  pc_next_o <= Controllo.pc_next_o;
  Am <= Controllo.Am;
  ab_c <= Controllo.ab_c;
  req1 <= Controllo.req1;
  req2 <= Controllo.req2;

END synth ;

```

Abbiamo riportato il codice della microRom per realizzare la rete di controllo di A.

Nella logica microprogrammata, a ogni stato coincidono certi segnali di controllo. È un modello diverso, programmabile che permette l'aggiunta di uno stato estendendo una struttura. Questa logica necessita, quindi, di un componente che permette, in modo flessibile, di manipolare i dati contenuti nella struttura e che sia programmabile. Il componente di cui necessitiamo è la ROM.

La microRom prevede in uscita i segnali: req1 e req2, l'abilitazione alla lettura dalla memoria e all'incremento del contatore ed il prossimo stato. Nell'architettura si procede alla dichiarazione di un record, control_word, formato da tutti questi segnali di uscita (che piloteranno la parte operativa, le richieste per iniziare la comunicazione con gli altri nodi, e un'uscita che concorrerà a determinare la prossima parola da trarre). Dichiaro, poi, un array di control_word (Rom_type) con tanti elementi

quanti sono gli stati rappresentati nell'automa in figura 12.3. In ogni control_word setto i parametri d'uscita, relativi al determinato stato, che andranno a pilotare il mio sistema.

Infine, mi dichiaro una signal “Controllo” di tipo control_word, che corrisponderà alla control_word specificata da pc_i, a cui associo tutte le variabili di uscita del mio sistema. In questo modo, dato un pc_i di ingresso, si accede alla locazione dell'array specificata da questo parametro di ingresso e si associa la corrispondente control_word (tra le 10 dell'array) al “Controllo” settando tutti i parametri di uscita della MicroRom.

12.2.1.4: RC_A

In aggiunta alla microRom, la logica microprogrammata fa uso di una logica esterna di controllo, e di un microProgramCounter che ha il compito di determinare qual è la prossima parola da trarre dalla ROM. Abbiamo deciso di inglobare nel codice la logica di controllo e il PC in un unico process come segue:

```
entity RC_A is
generic (
    K: integer := 7
);
Port (
    clk : in STD_LOGIC;
    rst : in STD_LOGIC;
    start : in STD_LOGIC;
    ack1 : in STD_LOGIC;
    ok1 : in STD_LOGIC;
    ack2 : in STD_LOGIC;
    ok2 : in STD_LOGIC;
    X: in std_logic_vector((K-1) downto 0);
    address: in std_logic_vector(3 downto 0);

    req1 : out STD_LOGIC;
    req2 : out STD_LOGIC;
    Am : out STD_LOGIC;
    ab_c : out STD_LOGIC;
    U: out std_logic_vector((K-2) downto 0)
);
end RC_A;
```

```

architecture micro_controlled of RC_A is

component microRom is
port (
  pc_i: in std_logic_vector (3 downto 0);
  pc_next_o : out std_logic_vector (3 downto 0);
  Am : out std_logic;
  ab_c : out std_logic;
  req1 : out std_logic;
  req2 : out std_logic
);
end component;

signal ab_res_int : std_logic;
signal pc_next: std_logic_vector(3 downto 0);
signal pc_next_oo: std_logic_vector(3 downto 0);

begin

rom : microROM port map(
  pc_i => pc_next,
  pc_next_o => pc_next_oo,
  Am => Am,
  ab_c => ab_c,
  req1 => req1,
  req2 => req2
);

```

```

--inglobiamo in un unico processo la logica esterna di controllo e il microPC
reg_PC: PROCESS(clk)
BEGIN
    IF (clk'event and clk = '1') THEN
        IF (rst = '1' OR (address = "0000" and pc_next="1001")) THEN
            pc_next <= "0000";
        ELSIF ((start = '1' and pc_next="0000") OR (address <="0111" and pc_next="1001")) THEN
            pc_next <= "0001";
        ELSIF (address <= "0111" and X(0) = '1' and pc_next="0010") THEN
            pc_next <= "0011";
            U<=X(6 downto 1);
        ELSIF (address <= "0111" and X(0) = '0' and pc_next="0010") THEN
            pc_next <= "0100";
            U<=X(6 downto 1);
        ELSIF (address > "0111" AND address(0)='0' and pc_next="1001") THEN
            pc_next <= "0011";
            U<="111111";
        ELSIF (address > "0111" AND address(0)='1' and pc_next="1001") THEN
            pc_next <= "0100";
            U<="111111";
        ELSIF (ack2 = '1' and pc_next="0011") THEN
            pc_next <= "0101";
        ELSIF (ack1 = '1' and pc_next="0100") THEN
            pc_next <= "0110";
        ELSIF ((ok1 = '1' and pc_next="0110") OR (ok2='1' and pc_next="0101")) THEN
            pc_next <= "0111";
        ELSIF (ok1 = '0' AND ok2='0' and pc_next="0111") THEN
            pc_next <= "1000";
        ELSE
            pc_next <= pc_next_oo;
        END IF;
    END IF;

    END PROCESS;
end micro_controlled;

```

Dati i segnali di ingresso alla logica, sulla base di questi e sulla base dello stato attuale in cui ci si trova, viene determinato il prossimo indirizzo a cui saltare. Viene settato il valore del pc_next, in ingresso alla ROM, e al colpo di clock successivo la Rom andrà a selezionare la parola corretta da eseguire.

Se nessuna condizione è soddisfatta si procede nell'ultimo ramo else, in cui la prossima parola che la rom dovrà eseguire, sarà quella specificata all'interno della microRom stessa, nel “pc_next_o”.

Il ragionamento sulle condizioni e sugli eventuali salti da effettuare sono gli stessi già spiegati nella descrizione del nodo A nel paragrafo 12.1.1.

12.2.1.5: A

Riportiamo, di seguito, l'implementazione del nodo A, secondo un approccio strutturale. Evidenziamo, così, i componenti di cui è composto e le relative interconnessioni di cui sono caratterizzati:

```
entity A is
  generic (
    K: integer := 7
  );
  Port (
    clk : in STD_LOGIC;
    rst : in STD_LOGIC;
    start : in STD_LOGIC;
    ack1 : in STD_LOGIC;
    ok1 : in STD_LOGIC;
    ack2 : in STD_LOGIC;
    ok2 : in STD_LOGIC;

    req1 : out STD_LOGIC;
    req2 : out STD_LOGIC;
    U: out std_logic_vector((K-2) downto 0)
  );
end A;

architecture Structural of A is

component cont is
  Port (
    clk : in STD_LOGIC;
    rst : in STD_LOGIC;
    ab_c : in STD_LOGIC;
    rst1: in std_logic;
    address : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component ROM is
  generic (
    N: integer :=8;
    K: integer :=7
  );
  port(
    CLK : in std_logic;
    Am : in std_logic;
    address : in std_logic_vector(3 downto 0);
    X: out std_logic_vector((K-1) downto 0)
  );
end component;
```

```

component RC_A is
generic (
  K: integer := 7
);
Port (
  clk : in STD_LOGIC;
  rst : in STD_LOGIC;
  start : in STD_LOGIC;
  ack1 : in STD_LOGIC;
  ok1 : in STD_LOGIC;
  ack2 : in STD_LOGIC;
  ok2 : in STD_LOGIC;
  X: in std_logic_vector((K-1) downto 0);
  address: in std_logic_vector(3 downto 0);

  req1 : out STD_LOGIC;
  req2 : out STD_LOGIC;
  Am : out STD_LOGIC;
  ab_c : out STD_LOGIC;
  U: out std_logic_vector((K-2) downto 0)
);

end component;

signal ab_cont: std_logic;
signal ab_rom: std_logic;
signal cont_u: std_logic_vector(3 downto 0);
signal x_u: std_logic_vector((K-1) downto 0);

begin

c: cont port map(
  clk => clk,
  rst => rst,
  ab_c => ab_cont,
  rst1 =>'0',
  address => cont_u
);
rom_mem: ROM port map(
  clk => clk,
  Am => ab_rom,
  address => cont_u,
  X => x_u
);
ret_c: RC_A Port map (
  clk => clk,
  rst => rst,
  start => start,
  ack1 => ack1,
  ok1 => ok1,
  ack2 => ack2,
  ok2 => ok2,
  X => x_u,
  address => cont_u,

```

```

    req1 => req1,
    req2 => req2,
    Am => ab_rom,
    ab_c => ab_cont,
    U => U
);
end Structural;

```

12.2.2: Nodo B

12.2.2.1: Registro

```

entity reg is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    conteggio: in std_logic_vector(3 downto 0);
    ab_reg: in std_logic;
    conteggio_out: out std_logic_vector(3 downto 0)
  );
end reg;

architecture Behavioral of reg is

begin
  process(clk)
  begin
    if(rising_edge(clk)) then
      if(rst='1') then
        conteggio_out<="0000";
      else
        if(ab_reg='1') then
          conteggio_out<=conteggio;
        end if;
      end if;
    end if;
  end process;
end Behavioral;

```

12.2.2.2: Comparatore

L'entità "comp", compara l'ingresso X con la stringa di terminazione "111111". Descritto secondo un approccio comportamentale, se sono uguali pone come bit di uscita 1, altrimenti 0:

```

entity comp is
  generic(
    K: integer:= 7;
    uni: std_logic_vector(5 downto 0):="111111"
  );
  port(
    X: in std_logic_vector((K-2) downto 0);
    u_c_c : out std_logic);
end entity;

```

```

architecture behavioral of comp is

begin

process(X)
begin
    if X = uni then
        u_c_c <= '1';
    else
        u_c_c <= '0';
    end if;
end process;
end architecture behavioral;

```

12.2.2.3: RC_B

```

entity RC_B is

Port (
    clk: in std_logic;
    rst: in std_logic;
    req3: in std_logic;
    req2: in std_logic;
    u_c_b: in std_logic;

    ok3: out std_logic;
    ack3: out std_logic;
    ack2: out std_logic;
    ok2: out std_logic;
    ab_r: out std_logic
);
end RC_B;

architecture Behavioral of RC_B is

type state is(
    IDLE, COM_A, INV, WAIT_C, COM_C, REG, INV2
);
signal current_state, next_state: state;

begin

ag: process(clk)
begin
    if(rising_edge(clk)) then
        if(rst='1') then
            current_state<=IDLE;
        else
            current_state<=next_state;
        end if;
    end if;
end process;

```

```

c: process(current_state, req3, req2, u_c_b)
begin

    ok3<='0';
    ack3<='0';
    ok2<='0';
    ack2<='0';
    ab_r<='0';

    case current_state is

        when IDLE =>

            if(req2='0') then
                next_state <=IDLE;
            else
                if(req2='1') then
                    next_state <= COM_A;
                end if;
            end if;

        when COM_A =>

            ack2<='1';

            next_state <= INV;

        when INV =>
            ack2<='1';
            ok2<='1';

            if(req2='1') then
                next_state<=INV;
            else
                if(req2='0' and u_c_b='0') then
                    next_state<=IDLE;
                else
                    if(req2='0' and u_c_b='1') then
                        next_state<=WAIT_C;
                    end if;
                end if;
            end if;

        when WAIT_C =>

            if(req3='0') then
                next_state<=WAIT_C;
            else
                if(req3='1') then
                    next_state<=COM_C;
                end if;
            end if;

```

```

when COM_C =>
    ack3<='1';

    next_state<=REG;

when REG =>
    ack3<='1';
    ab_r<='1';

    next_state<=INV2;

when INV2=>
    ack3<='1';
    ok3<='1';

    if(req3='1') then
        next_state<=INV2;
    else
        if(req3='0') then
            next_state<=IDLE;

            end if;
        end if;

    end case;
end process;

end Behavioral;

```

L'implementazione della rete di controllo di B segue quanto già descritto nell'automa in figura 12.5, nel paragrafo 12.1.2.

12.2.2.4: B

```

entity B is
generic (
    K: integer := 7
);
Port (
    clk: in std_logic;
    rst: in std_logic;
    req2: in std_logic;
    req3: in std_logic;
    conteggio: in std_logic_vector(3 downto 0);
    X: in std_logic_vector((K-2) downto 0);

    ack2: out std_logic;
    ok2: out std_logic;
    ack3: out std_logic;
    ok3: out std_logic;
    conteg_o: out STD_LOGIC_VECTOR (3 downto 0)

);
end B;

```

```

architecture Structural of B is

component reg is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    conteggio: in std_logic_vector(3 downto 0);
    ab_reg: in std_logic;
    conteggio_out: out std_logic_vector(3 downto 0)
  );
end component;

component comp is
  generic(
    K: integer:= 7;
    uni: std_logic_vector(5 downto 0):="111111"
  );
  port(
    X: in std_logic_vector((K-2) downto 0);
    u_c_c : out std_logic);
end component;

component RC_B is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    req3: in std_logic;
    req2: in std_logic;
    u_c_b: in std_logic;

    ok3: out std_logic;
    ack3: out std_logic;
    ack2: out std_logic;
    ok2: out std_logic;
    ab_r: out std_logic
  );
end component;

signal ucb: std_logic;
signal ab_registro: STD_LOGIC;

begin

reg2: reg Port map (
  clk => clk,
  rst => rst,
  conteggio => conteggio,
  ab_reg => ab_registro,
  conteggio_out => conteg_o
);

```

```

comp2: comp port map (
    x => x,
    u_c_c => ucb
);

rete_cont: RC_B Port map (
    clk => clk,
    rst => rst,
    req3 => req3,
    req2 => req2,
    u_c_b => ucb,
    ok3 => ok3,
    ack3 => ack3,
    ack2 => ack2,
    ok2 => ok2,
    ab_r => ab_registro
);
end Structural;

```

12.2.3: Nodo C

12.2.3.1: REGISTRO, CONTATORE, COMPARATORE

Questi tre componenti sono stati già descritti precedentemente. In particolare, il contatore nella descrizione del nodo A e il registro e comparatore nella descrizione del nodo B. Non riportiamo dunque l'implementazione essendo la stessa.

12.2.3.2: RC_C

```

entity RC_C is

    Port (
        clk: in std_logic;
        rst: in std_logic;
        req1: in std_logic;
        ack3: in std_logic;
        ok3: in std_logic;
        u_c_c: in std_logic;

        ok1: out std_logic;
        ack1: out std_logic;
        req3: out std_logic;
        ab_reg: out std_logic;
        ab_cont: out std_logic;
        rst1: out std_logic
    );
end RC_C;

```

```

architecture Behavioral of RC_C is

type state is(
    IDLE, CONT, INIT, WAIT1, MEM, INV, WAIT2, WAIT3, RESET
);
signal current_state, next_state: state;

begin

ag: process(clk)
begin
    if(rising_edge(clk)) then
        if(rst='1') then
            current_state<=IDLE;
        else
            current_state<=next_state;
        end if;
    end if;
end process;

c: process(current_state, req1, ack3, ok3, u_c_c)
begin

ok1<='0';
ack1<='0';
req3<='0';
ab_reg<='0';
ab_cont<='0';
rst1<='0';

case current_state is

when IDLE =>

    if(req1='0') then
        next_state <=IDLE;
    else
        if(req1='1') then
            next_state <= INIT;
        end if;
    end if;

when INIT =>

    ack1<='1';

    next_state <= WAIT1;

```

```

when WAIT1 =>
    ack1<='1';
    ok1<='1';
    if(req1='1') then
        next_state<=WAIT1;
    else
        if(req1='0' and u_c_c='0') then
            next_state<=CONT;
        else
            if(req1='0' and u_c_c='1') then
                next_state<=MEM;
            end if;
        end if;
    end if;

when CONT =>
ab_cont<='1';

next_state<=IDLE;

when MEM =>
ab_reg<='1';

next_state<=INV;

when INV =>
req3<='1';

if(ack3='0') then
    next_state<=INV;
else
    if(ack3='1') then
        next_state<=WAIT2;
    end if;
end if;

when WAIT2=>

if(ok3='0') then
    next_state<=WAIT2;
else
    if(ok3='1') then
        next_state<=WAIT3;
    end if;
end if;

```

```

when WAIT3=>
    if(ok3='1') then
        next_state<=WAIT3;
    else
        if(ok3='0') then
            next_state<=RESET;
        end if;
    end if;

when RESET=>
    rst1<='1';

next_state<=IDLE;

end case;
end process;

```

end Behavioral;

12.2.3.3: C

```

entity C is
generic (
    K: integer := 7
);
Port (
    clk: in std_logic;
    rst: in std_logic;
    X: in std_logic_vector((K-2) downto 0);
    req1: in std_logic;
    ack3: in std_logic;
    ok3: in std_logic;

    req3: out std_logic;
    ack1: out std_logic;
    ok1: out std_logic;
    conteggio: out std_logic_vector(3 downto 0)
);
end C;

architecture Structural of C is

component comp is
generic(
    K: integer:= 7;
    uni: std_logic_vector(5 downto 0):="111111"
);
port(
    X: in std_logic_vector((K-2) downto 0);
    u_c_c : out std_logic);
end component;

```

```

component cont is
  Port (
    clk : in STD_LOGIC;
    rst : in STD_LOGIC;
    ab_c : in STD_LOGIC;
    rst1: in std_logic;
    address : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component reg is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    conteggio: in std_logic_vector(3 downto 0);
    ab_reg: in std_logic;
    conteggio_out: out std_logic_vector(3 downto 0)
  );
end component;

component RC_C is
  Port (
    clk: in std_logic;
    rst: in std_logic;
    req1: in std_logic;
    ack3: in std_logic;
    ok3: in std_logic;
    u_c_c: in std_logic;

    ok1: out std_logic;
    ack1: out std_logic;
    req3: out std_logic;
    ab_reg: out std_logic;
    ab_cont: out std_logic;
    rst1: out std_logic
  );
end component;

signal ucc: std_logic;
signal abc: std_logic;
signal conteg: STD_LOGIC_VECTOR (3 downto 0);
signal ab_registro: STD_LOGIC;
signal reset1: STD_LOGIC;

begin
  compl: comp port map(
    X =>X,
    u_c_c =>ucc
  );

```

```

cont1: cont Port map (
    clk => clk,
    rst => rst,
    ab_c => abc,
    rst1 => reset1,
    address => conteg
);

reg1: reg Port map(
    clk => clk,
    rst => rst,
    conteggio => conteg,
    ab_reg => ab_registro,
    conteggio_out => conteggio
);

rete_controllo: RC_C port map(
    clk => clk,
    rst => rst,
    req1 => req1,
    ack3 => ack3,
    ok3 => ok3,
    u_c_c => ucc,
    ok1 => ok1,
    ack1 => ack1,
    req3 => req3,
    ab_reg => ab_registro,
    ab_cont => abc,
    rst1 => reset1
);

end Structural;

```

12.2.4: TOTALE

Riportiamo di seguito il sistema globale che racchiude le tre entità descritte nei paragrafi precedenti:

```
entity TOT is
  generic (
    K: integer := 7
  );
  Port (
    clk : in STD_LOGIC;
    rst : in STD_LOGIC;
    start : in STD_LOGIC;

    U_A: out std_logic_vector((K-2) downto 0);
    req1 : out STD_LOGIC;
    req2 : out STD_LOGIC;
    req3: out std_logic;
    conteggio_C: out std_logic_vector(3 downto 0);
    conteggio_B: out STD_LOGIC_VECTOR (3 downto 0)
  );
end TOT;

architecture Structural of TOT is

component A is
  generic (
    K: integer := 7
  );
  Port (
    clk : in STD_LOGIC;
    rst : in STD_LOGIC;
    start : in STD_LOGIC;
    ack1 : in STD_LOGIC;
    ok1 : in STD_LOGIC;
    ack2 : in STD_LOGIC;
    ok2 : in STD_LOGIC;

    req1 : out STD_LOGIC;
    req2 : out STD_LOGIC;
    U: out std_logic_vector((K-2) downto 0)
  );
end component;
```

```

component B is
  generic (
    K: integer := 7
  );
  Port (
    clk: in std_logic;
    rst: in std_logic;
    req2: in std_logic;
    req3: in std_logic;
    conteggio: in std_logic_vector(3 downto 0);
    X: in std_logic_vector((K-2) downto 0);

    ack2: out std_logic;
    ok2: out std_logic;
    ack3: out std_logic;
    ok3: out std_logic;
    conteg_o: out STD_LOGIC_VECTOR (3 downto 0)
  );
end component;

component C is
  generic (
    K: integer := 7
  );
  Port (
    clk: in std_logic;
    rst: in std_logic;
    X: in std_logic_vector((K-2) downto 0);
    req1: in std_logic;
    ack3: in std_logic;
    ok3: in std_logic;

    req3: out std_logic;
    ack1: out std_logic;
    ok1: out std_logic;
    conteggio: out std_logic_vector(3 downto 0)
  );
end component;

signal ackk1: std_logic;
signal okk1: std_logic;
signal ackk2: std_logic;
signal okk2: std_logic;
signal ackk3: std_logic;
signal okk3: std_logic;
signal reqq1: std_logic;
signal reqq2: std_logic;
signal reqq3: std_logic;
signal usc_a: std_logic_vector((K-2) downto 0);
signal co: std_logic_vector(3 downto 0);

begin

```

```

A1: A Port map(
  clk => clk,
  rst => rst,
  start => start,
  ack1 => ackk1,
  ok1 => okk1,
  ack2 => ackk2,
  ok2 => okk2,
  req1 => reqq1,
  req2 => reqq2,
  U => usc_a
);

B1: B Port map (
  clk => clk,
  rst => rst,
  req2 => reqq2,
  req3 => reqq3,
  conteggio => co,
  X => usc_a,
  ack2 => ackk2,
  ok2 => okk2,
  ack3 => ackk3,
  ok3 => okk3,
  conteg_o => conteggio_B
);

C1: C Port map(
  clk => clk,
  rst => rst,
  X => usc_a,
  req1 => reqq1,
  ack3 => ackk3,
  ok3 => okk3,
  req3 => reqq3,
  ack1 => ackk1,
  ok1 => okk1,
  conteggio => co
);

req1 <=reqq1;
req2 <=reqq2;
req3 <= reqq3;
U_A <= usc_a;
conteggio_C <= co;

end Structural;

```

12.3: SIMULAZIONE

Dato il segnale di start, il sistema “TOT” inizia la sua elaborazione.

```

process
begin

    rst<='1';
    wait for 105ns;
    rst<='0';

    st<='1';
    wait for 30 ns;
    st<='0';

    wait;
end process;

end Behavioral;

```

Il risultato della simulazione è riportato in figura 12.8:

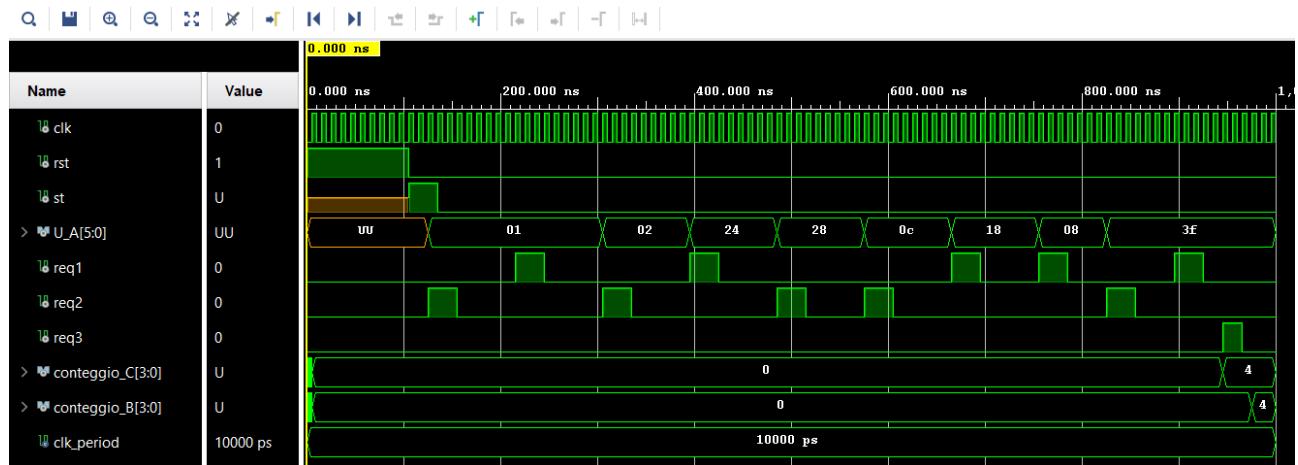


Figura 12.8: Testbench

In simulazione possiamo osservare: le uscite del nodo A (che corrispondono con quanto descritto nel paragrafo 12.2.1.2 sulla Rom di A), le richieste (req1 e req2 si alzano secondo quanto aspettato), il conteggio di C e l’uscita Conteggio_B. Quest’ultima rappresenta il conteggio inviato da C.

La trasmissione va a buon fine, infatti, il conteggio inviato da C (nel nostro caso 4) coincide con quello ricevuto da B.