



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

Elaborato **Architettura dei sistemi digitali**

Tesina ASDi

Anno Accademico 2020/21

Gruppo 3:

Michele Maresca M63/1151

Vincenzo Riccardi M63/1146

Marco Feliciano M63/1136

Indice

INDICE	III
ESERCIZIO 1	5
1.1 TRACCIA	5
1.2 SOLUZIONE	5
1.3 CODICE	7
1.3.1 <i>Struttura ad albero</i>	7
1.3.2 <i>Struttura a semi selezione</i>	9
1.4 SIMULAZIONE	10
ESERCIZIO 2	16
2.1 TRACCIA	16
2.2 SOLUZIONE	16
2.2.1 <i>Riconoscitore 1-1</i>	16
2.2.2 <i>Riconoscitore 1-10</i>	18
2.3 CODICE	19
2.3.1 <i>Riconoscitore 1-1</i>	19
2.3.2 <i>Riconoscitore 1-10</i>	25
2.4 SIMULAZIONE	29
2.4.1 <i>Riconoscitore 1-1</i>	29
2.4.2 <i>Riconoscitore 1-10</i>	32
ESERCIZIO 3	36
3.1 TRACCIA	36
3.2 SOLUZIONE	36
3.3 CODICE	37
3.4 SIMULAZIONE	42
3.5 SINTESI SU FPGA	44
ESERCIZIO 4	51
4.1 TRACCIA	51
4.2 SOLUZIONE	51
4.3 CODICE	52
4.3.1 <i>Approccio Strutturale</i>	52
4.3.2 <i>Approccio comportamentale</i>	56
4.4 SIMULAZIONE	57
ESERCIZIO 5	60
5.1 TRACCIA	60
5.2 SOLUZIONE	60
5.3 CODICE	63
5.3.1 <i>Logica Cablata</i>	63
5.3.2 <i>Logica Micropogrammata</i>	71
5.4 SIMULAZIONE	85
5.4.1 <i>Simulazione logica cablata</i>	85
5.4.2 <i>Simulazione logica micropogrammata</i>	88
ESERCIZIO 6	91
6.1 TRACCIA	91
6.2 SOLUZIONE	91
6.3 CODICE	92
6.4 SIMULAZIONE	97
ESERCIZIO 7	100

7.1 TRACCIA	100
7.2 SOLUZIONE	100
7.2.1 <i>Implementazione con vettori in rom</i>	100
7.2.2 <i>Implementazione con handshake e contatore</i>	102
7.3 CODICE	103
7.3.1 <i>Implementazione con vettori in rom</i>	103
7.3.2 <i>Implementazione con handshake e contatore</i>	111
7.4 SIMULAZIONE	117
7.4.1 <i>Implementazione con vettori in rom</i>	117
7.4.2 <i>Implementazione con handshake e contatore</i>	120
ESERCIZIO 8	124
8.1 TRACCIA	124
8.2 RICHIAMO SULL'ARCHITETTURA	124
8.3 ANALISI IN SIMULAZIONE	128
8.3.1 <i>Analisi Bipush</i>	130
8.3.2 <i>Analisi IADD</i>	135
8.3.3 <i>Analisi ISTORE</i>	139
8.4 MODIFICA DI UN'ISTRUZIONE E RELATIVA ANALISI IN SIMULAZIONE	146
ESERCIZIO 9	150
9.1 TRACCIA	150
9.2 RICHIAMO SUL FUNZIONAMENTO DELLA PERIFERICA SERIALE	150
9.3 PROGETTAZIONE, IMPLEMENTAZIONE, SIMULAZIONE E SINTESI SU FPGA DEI COMPONENTI	155
9.3.1 <i>Progettazione UART tappo e 2_UART</i>	155
9.3.2 <i>Implementazione in VHDL</i>	157
9.3.3 <i>Simulazione</i>	160
9.3.4 <i>Sintesi su FPGA</i>	166
ESERCIZIO 10	173
10.1 TRACCIA	173
10.2 RICHIAMO GENERALE SUGLI SWITCH	173
10.3 SOLUZIONE CON SCHEMA A PRIORITÀ FISSA	177
10.4 SOLUZIONE GENERIC CON GESTIONE DELLE COLLISIONI	181
10.5 CODICE	188
10.5.1 <i>Codice rete a priorità</i>	188
10.5.2 <i>Codice rete generic</i>	196
ESERCIZIO 11	207
11.1 TRACCIA	207
11.2 ACCENNI ALL'ALGORITMO DEL MOLTIPLICATORE DI ROBERTSON	207
11.3 SOLUZIONE	208
11.4 CODICE	212
11.5 SIMULAZIONE	222
11.6 SINTESI SU FPGA	224

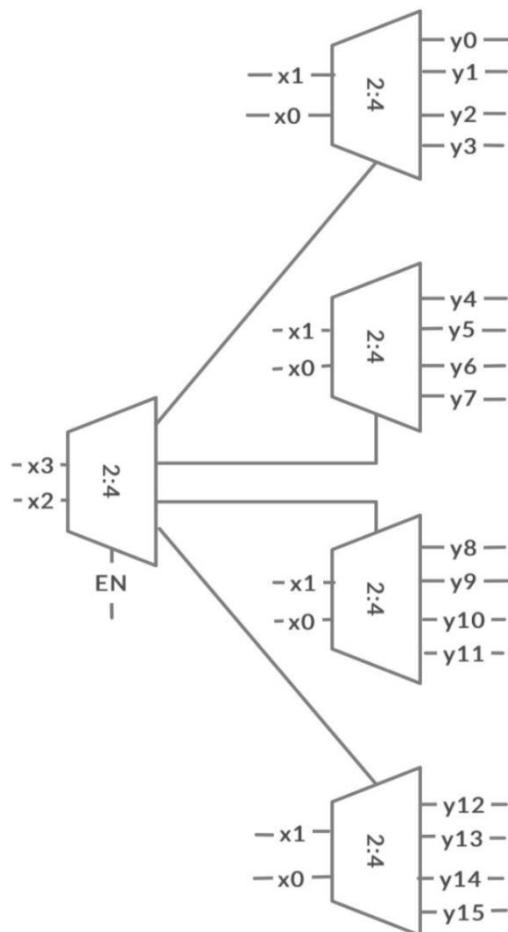
Esercizio 1

1.1 Traccia

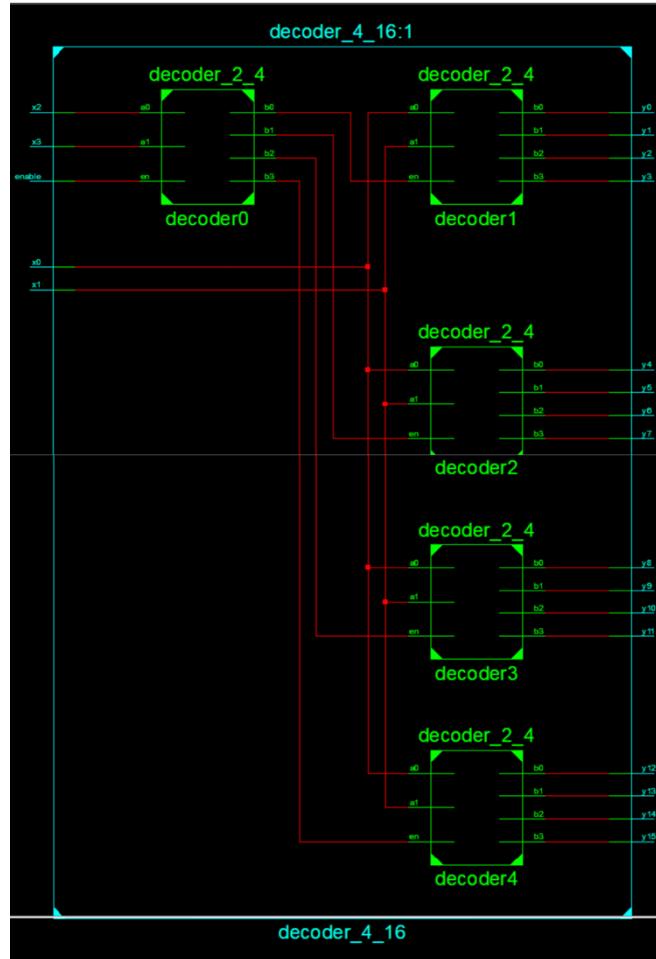
Si progetti un decoder 4:16 utilizzando componenti decoder 2:4 opportunamente interconnessi a) in una struttura ad albero e b) in una struttura a semiselezione.

1.2 Soluzione

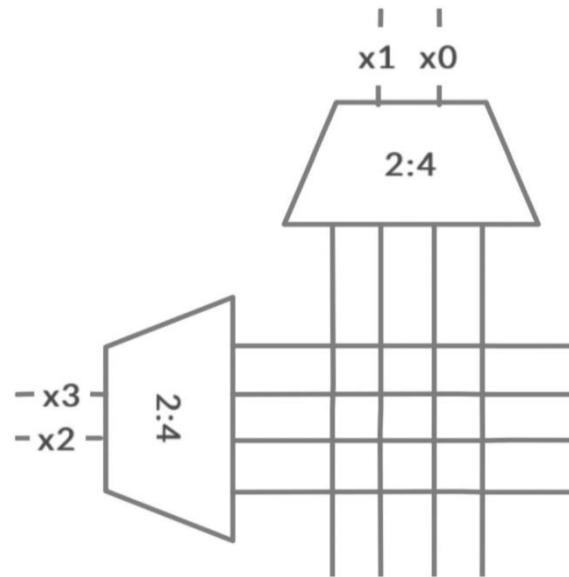
Per realizzare un decoder 4:16 in una struttura ad albero con decoder 2:4, si utilizza un approccio strutturale, in cui c'è un decoder 2:4 "d'interfaccia" che prende in ingresso i due bit più significativi ed un segnale di abilitazione. Il fatto che i decoder abbiano un segnale di abilitazione è indispensabile per la realizzazione di tale struttura. Infatti, ciascuna uscita del decoder d'interfaccia si comporta come segnale di abilitazione per i decoder successivi, che chiaramente sono 4, perché ci sono 4 uscite. Ciascuno di tali decoder che si trova al secondo stadio, dunque, prende come ingresso i 2 bit meno significativi ed un'uscita del decoder che si trova al primo stadio come segnale di abilitazione. Le 4 uscite di ciascun decoder del secondo stadio sono infine concatenate, per ottenere un vettore di 16 bit in uscita. Si ottiene il seguente schema:



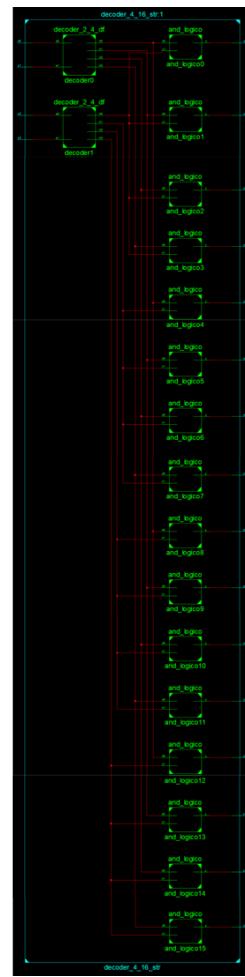
Di seguito è rappresentato lo schematico generato da Xilinx ISE, relativo a tale architettura.



Anche per realizzare la struttura a semiselezione si utilizza un approccio strutturale. L'idea è di avere due decoder 2:4 d'interfaccia, entrambi abilitati, uno dei quali prende in ingresso i due bit più significativi, e l'altro che prende in ingresso gli altri due bit. Se si mette in AND l'uscita meno significativa di uno con l'uscita meno significativa dell'altro, si ottiene l'uscita meno significativa del decoder complessivo 4:16. Iterando questo procedimento, è facile vedere che incrociando tutte le uscite e mettendole in AND a due a due si ottengono $4 \times 4 = 16$ uscite; se si incrocia il filo d'uscita **i** relativo al decoder che ha in ingresso i bit più significativi, ed il filo **j** relativo all'altro decoder, si ottiene l'uscita $k=4*i + j$ del decoder 4:16 complessivo. In figura è rappresentato lo schema risultato dell'architettura a semiselezione.



Mentre di seguito è rappresentato lo schematico generato da Xilinx ISE.



1.3 Codice

1.3.1 Struttura ad albero

Esplicitiamo inizialmente l'interfaccia del decoder 4:16 comune per entrambe le implementazioni, ad albero e a semiselezione:

```
entity decoder_4_16 is
  Port (
    x : IN  std_logic_vector(3 downto 0);
    s : IN  std_logic;
    y : OUT std_logic_vector(15 downto 0)
  );
end decoder_4_16;
```

Partiamo dal decoder 2:4, componente fondamentale per entrambi i casi:

```
entity decoder_2_4 is
  Port ( x1 : in STD_LOGIC;
         x0 : in STD_LOGIC;
         s : in STD_LOGIC;
         yv : out STD_LOGIC_VECTOR(3 downto 0));
end decoder_2_4;

architecture Dataflow of decoder_2_4 is

begin
  yv(0) <= (NOT(x1) AND NOT(x0)) AND s;
  yv(1) <= (NOT(x1) AND x0) AND s;
  yv(2) <= (x1 AND NOT(x0)) AND s;
  yv(3) <= (x1 AND x0) AND s;
end Dataflow;
```

Dunque, l'architettura ad albero:

```
architecture Tree of decoder_4_16 is
-- segnali interni
  signal u : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
-- dispositivi interni
  component decoder_2_4
    port (
      x1 : in STD_LOGIC;
      x0 : in STD_LOGIC;
      s : in STD_LOGIC;
      yv : out STD_LOGIC_VECTOR(3 downto 0));
  end component;

begin
  dec_first: decoder_2_4
    Port map(
      x1 => x(3),
      x0 => x(2),
      s => s,
```

```

        yv => u
    );

dec0: decoder_2_4
Port map(
    x1 => x(1),
    x0 => x(0),
    s => u(0),
    yv => y(3 downto 0)
);

dec1: decoder_2_4
Port map(
    x1 => x(1),
    x0 => x(0),
    s => u(1),
    yv => y(7 downto 4)
);

dec2: decoder_2_4
Port map(
    x1 => x(1),
    x0 => x(0),
    s => u(2),
    yv => y(11 downto 8)
);

dec3: decoder_2_4
Port map(
    x1 => x(1),
    x0 => x(0),
    s => u(3),
    yv => y(15 downto 12)
);

end Tree;

```

1.3.2 Struttura a semi selezione

Per l'architettura a semiselezione, è stato necessario definire un componente logic_and:

```

entity logic_and is
    Port ( a : in STD_LOGIC;
           b : in STD_LOGIC;
           y : out STD_LOGIC);
end logic_and;

architecture Dataflow of logic_and is
begin

```

```

y <= a and b;

end Dataflow;

```

Dunque, l'implementazione della semiselezione è stata fatta con il costrutto for ... generate:

```

Architecture SemiSel of decoder_4_16 is
-- segnali interni
signal u1 : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
signal u2 : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
-- dispositivi interni
component decoder_2_4
    port (
        x1 : in STD_LOGIC;
        x0 : in STD_LOGIC;
        s : in STD_LOGIC;
        yv : out STD_LOGIC_VECTOR(3 downto 0));
    end component;

component logic_and
    port (a : in STD_LOGIC; b : in STD_LOGIC; y: out STD_LOGIC);
end component;

begin
    dec1 : decoder_2_4 port map(
        x1 => x(3), x0 => x(2), s => s, yv => u1
    );
    dec2 : decoder_2_4 port map(
        x1 => x(1), x0 => x(0), s => s, yv => u2
    );
    -- u2 deve stare nel for più annidato del generate
    and_u1 : for i in 0 to 3 generate
        and_u2 : for j in 0 to 3 generate
            bit_out : logic_and port map(
                a => u2(j), b => u1(i), y => y(j + i*4)
            );
        end generate;
    end generate;
end SemiSel;

```

1.4 Simulazione

Per la simulazione e la verifica di funzionamento del sistema viene creato il *testbench*, il cui codice è mostrato di seguito:

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

```

```

use work.all;

ENTITY decoder_4_16_test_bench IS
END decoder_4_16_test_bench;

ARCHITECTURE behavior OF decoder_4_16_test_bench IS

COMPONENT decoder_4_16
PORT(
    x : IN  std_logic_vector(3 downto 0);
    s : IN  std_logic;
    y : OUT  std_logic_vector(15 downto 0)
);
END COMPONENT;

--Inputs
signal x : std_logic_vector(3 downto 0) := (others => '0');
signal s : std_logic := '1';

--Outputs
signal y : std_logic_vector(15 downto 0) := (others => '0');

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: entity work.decoder_4_16(SemiSel) PORT MAP (
    x => x,
    s => s,
    y => y
);

-- Stimulus process
stim_proc: process
begin

    x <= "0000";
    wait for 10 ns;
    assert y = "0000000000000001";
    report "errore0"
    severity failure;

    x <= "0001";
    wait for 10 ns;
    assert y = "00000000000000010";
    report "errore1"
    severity failure;

    x <= "0010";
    wait for 10 ns;

```

```

assert y = "000000000000100";
report "errore2"
severity failure;

x <= "0011";
wait for 10 ns;
assert y = "0000000000001000";
report "errore3"
severity failure;

x <= "0100";
wait for 10 ns;
assert y = "000000000010000";
report "errore4"
severity failure;

x <= "0101";
wait for 10 ns;
assert y = "0000000000100000";
report "errore5"
severity failure;

x <= "0110";
wait for 10 ns;
assert y = "0000000001000000";
report "errore6"
severity failure;

x <= "0111";
wait for 10 ns;
assert y = "0000000010000000";
report "errore7"
severity failure;

x <= "1000";
wait for 10 ns;
assert y = "0000000100000000";
report "errore8"
severity failure;

x <= "1001";
wait for 10 ns;
assert y = "0000001000000000";
report "errore9"
severity failure;

```

```

x <= "1010";
wait for 10 ns;
assert y = "0000010000000000";
report "errore10"
severity failure;

x <= "1011";
wait for 10 ns;
assert y = "0000100000000000";
report "errore11"
severity failure;

x <= "1100";
wait for 10 ns;
assert y = "0001000000000000";
report "errore12"
severity failure;

x <= "1101";
wait for 10 ns;
assert y = "0010000000000000";
report "errore13"
severity failure;

x <= "1110";
wait for 10 ns;
assert y = "0100000000000000";
report "errore14"
severity failure;

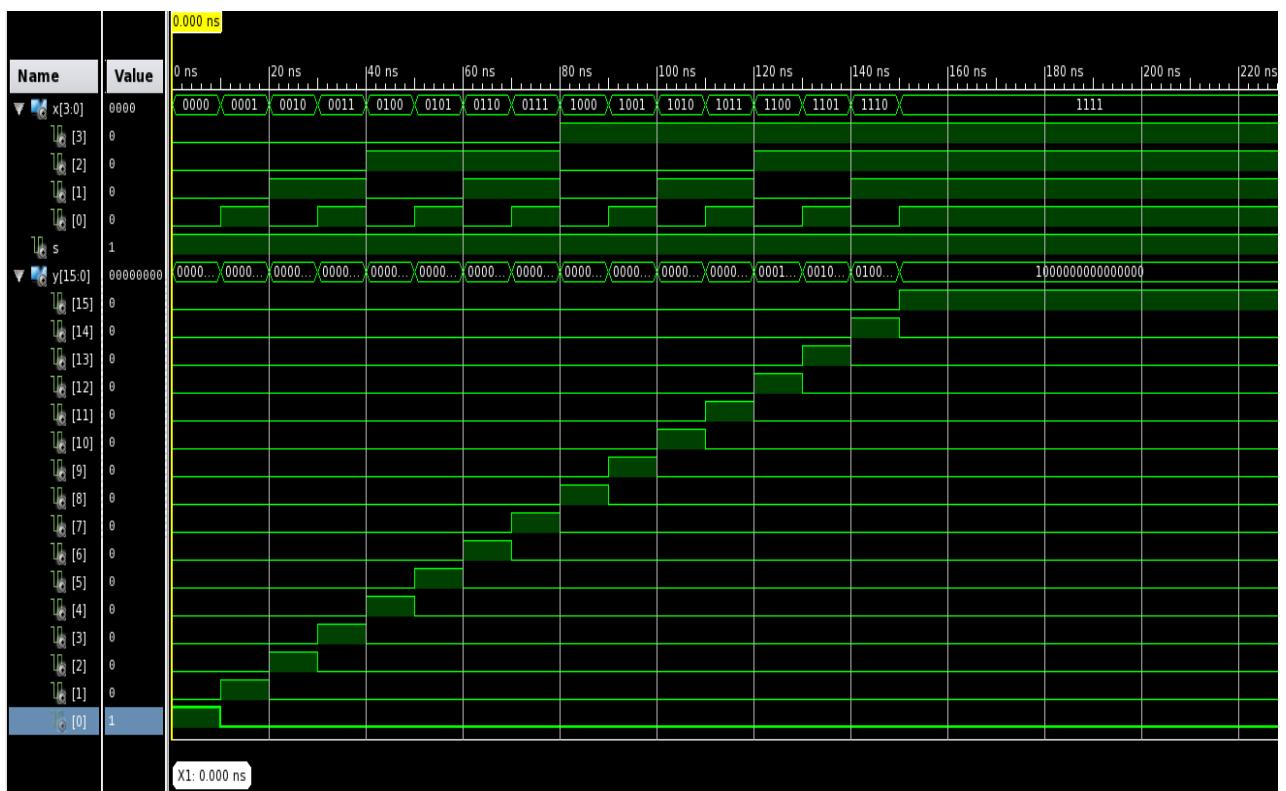
x <= "1111";
wait for 10 ns;
assert y = "1000000000000000";
report "errore15"
severity failure;

wait;
wait;
end process;

END;

```

Sono state quindi simulate le due architetture usando lo stesso testbench, mediante l'operazione di binding, per la quale è stato necessario aggiungere “use work.all” e specificare l'architettura nell'istanziazione dell'UUT. I risultati della simulazione, uguali per entrambe le architetture, sono riportati nella seguente figura:



Esercizio 2

2.1 Traccia

Esercizio 2-1

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

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

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

Esercizio 2-2

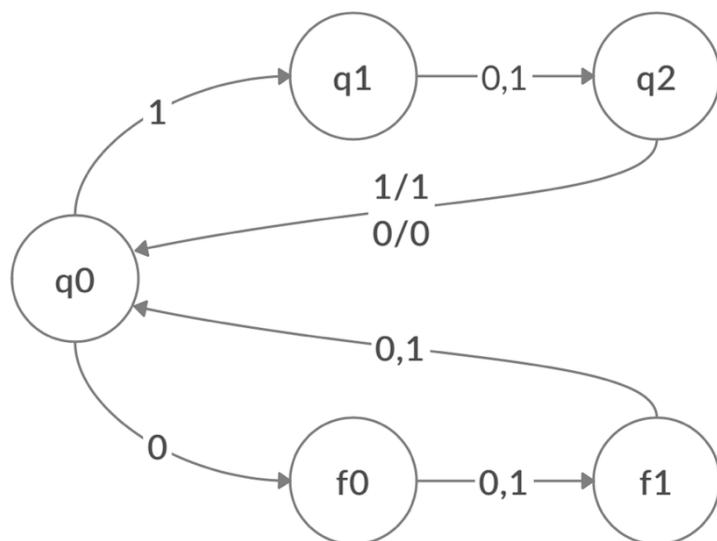
Si vuole progettare un riconoscitore di sequenza come macchina sincrona a sincronizzazione esterna: la macchina fornisce uscita alta quando viene riconosciuta la sequenza 1-10, e le sequenze possono sovrapporsi (esempio: la sequenza 11101010 produrrebbe un'uscita alta in corrispondenza del quarto, sesto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

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

2.2 Soluzione

2.2.1 Riconoscitore 1-10

Per implementare la soluzione, è stato conveniente partire dalla descrizione dell'automa in termini di grafo degli stati.



Dato che in questo caso era assente il concetto di sovrapposizione delle sequenze, un bit fuori sequenza porta all'interno di un percorso composto da stati di fallimento, che poi riporta allo stato iniziale q_0 in corrispondenza della ricezione di un numero di bit pari alla lunghezza della sequenza che si vuole riconoscere. Dal punto di vista della temporizzazione, sono stati usati dei flip-flop D sincronizzati sul fronte di salita: lo stato prossimo e l'uscita sono elaborati in funzione dell'ingresso e dello stato corrente, ed i loro valori commutano solo in corrispondenza del fronte di salita del segnale di abilitazione. Poiché gli stati sono 5, sono necessari 3 bit per codificarli, dunque si ha un registro formato da 3 flip-flop. Inoltre, per rispettare il requisito secondo il quale anche la commutazione dell'uscita è sincronizzata sul fronte di salita, è stato utilizzato un ulteriore flip-flop che ha la funzione di "buffer" dell'uscita.

A questo punto, visto che è richiesta una descrizione strutturale con componenti AND e OR, è stato necessario effettuare la sintesi mediante mappe di Karnaugh.

Nelle seguenti mappe di Karnaugh è stato indicato con DC l'eventuale presenza di punti di non specificazione "don't care".

Codifica degli stati:

$q_0 = 000, q_1 = 001, q_2 = 010, f_0 = 011, f_1 = 100$, con variabili di stato rispettivamente x_2, x_1 e x_0 .

Uscita:

y	x0 i	00	01	11	10
x2 x1					
00					
01			1		
11	DC	DC		DC	DC
10				DC	DC

$$y = x_1 \text{ and } (\text{not } x_0) \text{ and } i$$

Variabile di stato x_0 :

x0	x0 i	00	01	11	10
x2 x1					
00	1	1			
01					
11	DC	DC	DC	DC	DC
10				DC	DC

$$x_0 = (\text{not } x_2) \text{ and } (\text{not } x_1) \text{ and } (\text{not } x_0)$$

Variabile di stato x_1 :

x1	x0 i	00	01	11	10
x2 x1					
00	1		1	1	
01					
11	DC	DC	DC	DC	DC
10				DC	DC

$$x_1 = ((\text{not } x_1) \text{ and } x_0) \text{ or } ((\text{not } x_2) \text{ and } (\text{not } x_1) \text{ and } (\text{not } i))$$

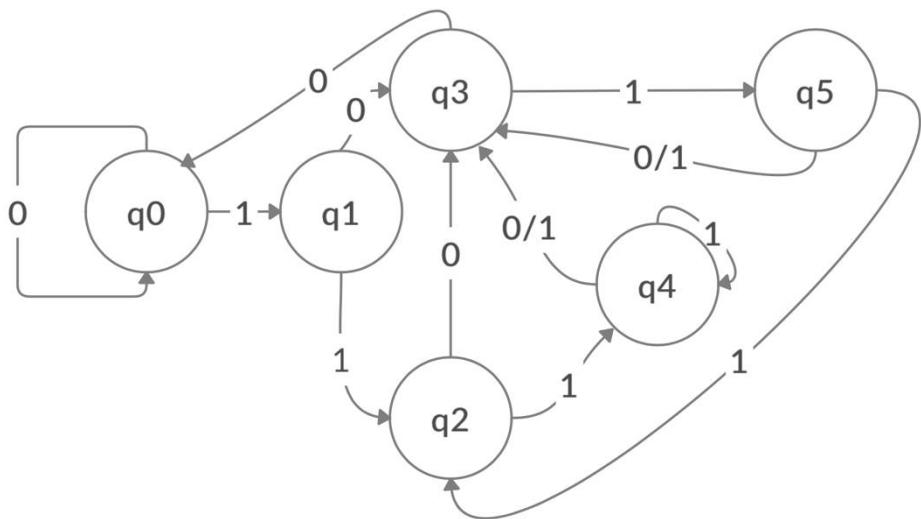
Variabile di stato x_2 :

x_2					
	$x_0 i$	00	01	11	10
$x_2 x_1$					
00					
01				1	1
11	DC	DC	DC	DC	DC
10			DC	DC	DC

$x_2 = x_1 \text{ and } x_0$

2.2.2 Riconoscitore 1-10

In questo caso bisognava riconoscere i casi in cui una sequenza composta dagli ultimi k bit ricevuti costituiva una sequenza parziale dei primi k bit della sequenza da riconoscere, mediante un approccio “greedy”, cioè cerco di riutilizzare il maggior numero di bit che ho già ricevuto: questo perché altrimenti non è rispettato il requisito di sequenze sovrapposte, che equivale ad una finestra scorrevole su 4 bit. Considerando i vari percorsi scaturiti dalla presenza del punto di non specificazione nella sequenza e riconoscendo le possibili sequenze parziali, l'automa risultante è il seguente:



Dal punto di vista della tempificazione, sono stati adottati gli stessi accorgimenti adottati per l'automa precedente.

Poiché è richiesta in seguito una descrizione ibrida in cui le funzioni di uscita/transizione sono realizzate mediante un modello dataflow, è stato necessario effettuare la sintesi mediante mappe di Karnaugh.

Codifica degli stati:

$q0 = 000, q1 = 001, q2 = 010, q3 = 011, q4 = 100, q5 = 101$ con variabili di stato rispettivamente x_2, x_1 e x_0 .

Uscita:

y	x0 i	00	01	11	10
x2 x1					
00					
01					
11	DC	DC	DC	DC	
10	1				1

$$y = x2 \text{ and } (\text{not } i)$$

Variabile di stato x0:

x0	x0 i	00	01	11	10
x2 x1					
00	1	1		1	
01					
11	DC	DC	DC	DC	
10	1				1

$$x0 = (x1 \text{ and } (\text{not } x0) \text{ and } (\text{not } i)) \text{ or } (x2 \text{ and } (\text{not } i)) \text{ or } (\text{not}(x1) \text{ and } x0 \text{ and } \text{not}(i)) \text{ or } (x1 \text{ and } x0 \text{ and } i) \text{ or } (\text{not}(x2) \text{ and } \text{not}(x1) \text{ and } \text{not}(x0) \text{ and } i)$$

Variabile di stato x1:

x1	x0 i	00	01	11	10
x2 x1					
00	1			1	1
01					
11	DC	DC	DC	DC	
10	1			1	1

$$x1 = (x1 \text{ and } \text{not}(x0) \text{ and } \text{not}(i)) \text{ or } (x2 \text{ and } \text{not}(i)) \text{ or } ((\text{not } x1) \text{ and } x0)$$

Variabile di stato x2:

x2	x0 i	00	01	11	10
x2 x1					
00					
01			1	1	
11	DC	DC	DC	DC	DC
10	1				

$$x2 = (x1 \text{ and } i) \text{ or } (x2 \text{ and } \text{not}(x0) \text{ and } i)$$

2.3 Codice

2.3.1 Riconoscitore 1-1

Riportiamo l'entity, che è ovviamente uguale per i punti (a) e (b), essendo l'interfaccia:

```
entity sequenza_1_dc_1 is
  Port ( i : in STD_LOGIC;
         clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         y : out STD_LOGIC);
end sequenza_1_dc_1;
```

A questo punto, per il punto (a), l'implementazione comportamentale:

```
architecture Behavioral of sequenza_1_dc_1 is

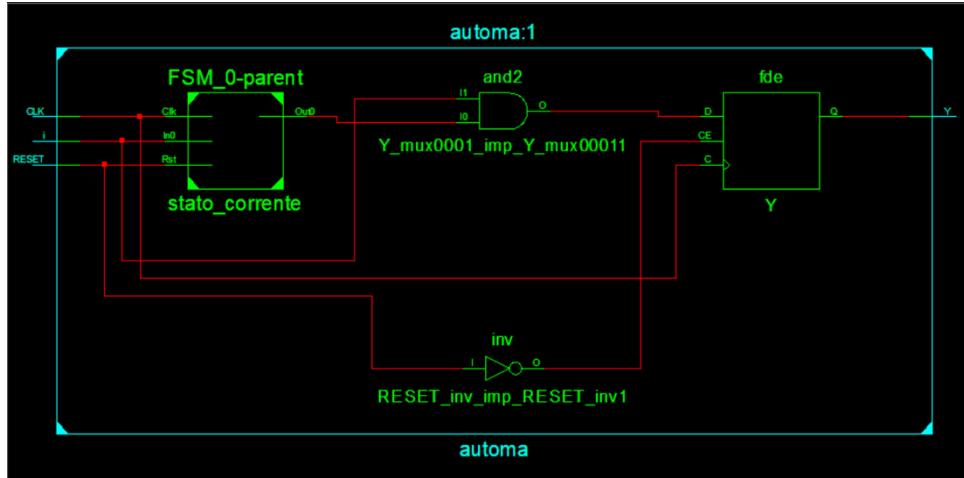
type state is (q0, q1, q2, f0, f1);
signal current : state := q0;

begin

seq_rec : process(clk)
begin
    if (clk = '1' and clk'event) then
        if (rst = '1') then
            current <= q0;
            y <= '0';
        else
            case current is
                when q0 =>
                    if (i = '1') then
                        current <= q1;
                    elsif (i = '0') then
                        current <= f0;
                    end if;
                    y <= '0';
                when q1 =>
                    current <= q2;
                    y <= '0';
                when q2 =>
                    current <= q0;
                    if (i = '1') then
                        y <= '1';
                    else
                        y <= '0';
                    end if;
                when f0 =>
                    current <= f1;
                    y <= '0';
                when f1 =>
                    current <= q0;
                    y <= '0';
                when others =>
                    current <= q0;
                    y <= '0';
            end case;
        end if;
    end if;
end process;
```

```
end Behavioral;
```

Si ottiene un'architettura RTL simile alla seguente:



Invece, per il punto (b), che richiedeva un'implementazione strutturale mediante componenti base realizzati ad-hoc; riporteremo prima i sotto-componenti e poi l'architettura che li mette insieme per comporre l'automa.

Flip-flop D sincronizzato sul fronte di salita, con reset sincrono:

```
entity flip_flop_D is
  Port ( d : in STD_LOGIC;
         clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         q : out STD_LOGIC);
end flip_flop_D;

architecture Behavioral of flip_flop_D is

begin
  ff : process(clk)
  begin
    if (clk = '1' and clk'event) then
      if (rst = '1') then
        q <= '0';
      else
        q <= d;
      end if;
    end if;
  end Behavioral;
```

Logic NOT (necessaria, AND e OR da sole non bastano):

```
entity logic_not is
```

```

Port ( a : in STD_LOGIC;
       y : out STD_LOGIC);
end logic_not;

architecture dataflow of logic_not is

begin
    y <= not a;

end dataflow;

```

Logic OR a due ingressi:

```

entity logic_or is
    Port ( a : in STD_LOGIC;
           b : in STD_LOGIC;
           y : out STD_LOGIC);
end logic_or;

architecture dataflow of logic_or is

begin
    y <= a or b;

end dataflow;

```

Logic AND con un numero generico di ingressi:

```

entity logic_and is
    Generic ( n : positive := 2 );
    Port ( a : in STD_LOGIC_VECTOR(n-1 downto 0);
           y : out STD_LOGIC);
end logic_and;

architecture Behavioral of logic_and is

begin
    and_func : process(a)
        variable result : std_logic;
        begin
            result := '1';
            for i in 0 to n-1 loop
                result := result and a(i);
            end loop;
            y <= result;
        end process;
    end Behavioral;

```

Architettura che mette insieme i componenti realizzando le funzioni di transizione di stato, quella di uscita e le memorie:

```

architecture Structural of sequenza_1_dc_1 is
-- segnali interni
signal neg_i : std_logic := '0';
signal x0 : std_logic := '0';
signal x1 : std_logic := '0';
signal x2 : std_logic := '0';
signal neg_x0 : std_logic := '0';
signal neg_x1 : std_logic := '0';
signal neg_x2 : std_logic := '0';
signal x0_next : std_logic := '0';
signal x1_next : std_logic := '0';
signal x2_next : std_logic := '0';
signal u0_x1 : std_logic := '0';
signal u1_x1 : std_logic := '0';
signal u_y : std_logic := '0';
-- dispositivi interni
component flip_flop_D is
    Port ( d : in STD_LOGIC;
           clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           q : out STD_LOGIC);
end component;

component logic_and is
    Generic ( n : positive := 2 );
    Port ( a : in STD_LOGIC_VECTOR(n-1 downto 0);
           y : out STD_LOGIC);
end component;

component logic_or is
    Port ( a : in STD_LOGIC;
           b : in STD_LOGIC;
           y : out STD_LOGIC);
end component;

component logic_not is
    Port ( a : in STD_LOGIC;
           y : out STD_LOGIC);
end component;
begin
    not_x0 : logic_not
        Port map(
            a => x0,
            y => neg_x0
        );
    not_x1 : logic_not
        Port map(
            a => x1,
            y => neg_x1

```

```

);
not_x2 : logic_not
Port map(
    a => x2,
    y => neg_x2
);
not_i : logic_not
Port map(
    a => i,
    y => neg_i
);
x0_func : logic_and generic map(3)
Port map(
a(0) => neg_x0,
    a(1) => neg_x1,
    a(2) => neg_x2,
    y => x0_next
);
u0_x1_func : logic_and generic map(3)
Port map(
    a(0) => neg_i,
    a(1) => neg_x1,
    a(2) => neg_x2,
    y => u0_x1
);
u1_x1_func : logic_and generic map(2)
Port map(
    a(0) => x0,
    a(1) => neg_x1,
    y => u1_x1
);
x1_func : logic_or
Port map(
    a => u0_x1,
    b => u1_x1,
    y => x1_next
);
x2_func : logic_and generic map(2)
Port map(
    a(0) => x0,
    a(1) => x1,
    y => x2_next
);
mem0 : flip_flop_D
Port map(
    d => x0_next,
    clk => clk,
    rst => rst,
    q => x0
);

```

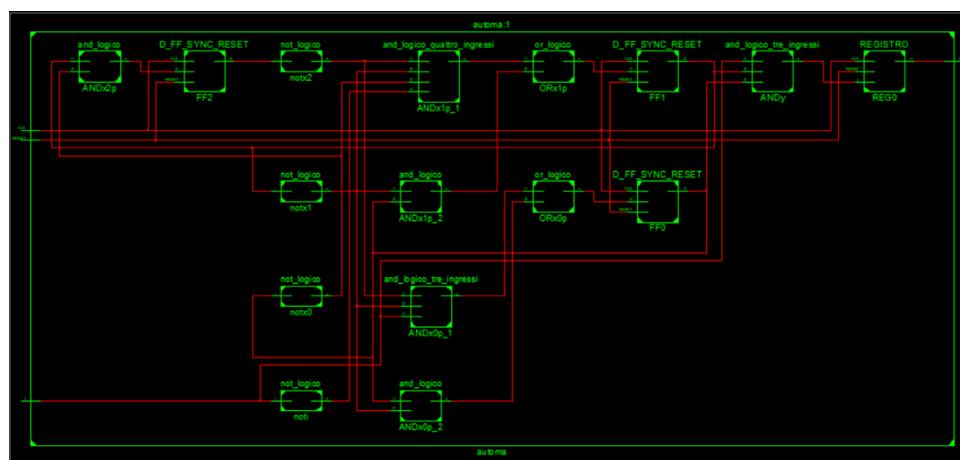
```

mem1 : flip_flop_D
Port map(
    d => x1_next,
    clk => clk,
    rst => rst,
    q => x1
);
mem2 : flip_flop_D
Port map(
    d => x2_next,
    clk => clk,
    rst => rst,
    q => x2
);
y_func : logic_and generic map(3)
Port map(
    a(0) => neg_x0,
    a(1) => x1,
    a(2) => i,
    y => u_y
);
y_buffer : flip_flop_D
Port map(
    d => u_y,
    clk => clk,
    rst => rst,
    q => y
);
);

end Structural;

```

L'architettura RTL che si ottiene come risultato è simile alla seguente:



Anche in questo caso, riportiamo prima l'entity:

```
entity seq_sovr_1_dc_1_0 is
```

```

Port ( i : in STD_LOGIC;
       clk : in STD_LOGIC;
       rst : in STD_LOGIC;
       y : out STD_LOGIC);
end seq_sovr_1_dc_1_0;

```

Per il punto (a), cioè l'implementazione mediante due processi:

```

architecture Behavioral of seq_sovr_1_dc_1_0 is
type state is (q0, q1, q2, q3, q4, q5);
signal current : state := q0;
signal next_state : state;
signal next_out : std_logic;

begin
transition : process(i, current, clk)
begin
  case current is
    when q0 =>
      if (i = '1') then
        next_state <= q1;
      else
        next_state <= q0;
      end if;
      next_out <= '0';
    when q1 =>
      if (i = '1') then
        next_state <= q2;
      else
        next_state <= q3;
      end if;
      next_out <= '0';
    when q2 =>
      if (i = '1') then
        next_state <= q4;
      else
        next_state <= q3;
      end if;
      next_out <= '0';
    when q3 =>
      if (i = '1') then
        next_state <= q5;
      else
        next_state <= q0;
      end if;
      next_out <= '0';
    when q4 =>
      if (i = '1') then
        next_state <= q4;
      next_out <= '0';

```

```

        else
            next_state <= q3;
            next_out <= '1';
        end if;
    when q5 =>
        if (i = '1') then
            next_out <= '0';
            next_state <= q2;
        else
            next_out <= '1';
            next_state <= q3;
        end if;

    when others =>
        next_state <= q0;
        next_out <= '0';
    end case;
    if (clk = '1' and clk'event) then
        y <= next_out;
    end if;
end process;

mem : process(clk, rst)
begin
    if (rst = '1') then
        current <= q0;
    elsif (clk = '1' and clk'event) then
        current <= next_state;
    end if;
end process;

end Behavioral;

```

Per il punto (b), riportiamo prima i sotto-componenti.
Funzioni di transizione di stato e di uscita:

```

entity funzioni_tau_omega is
    Port ( q : in STD_LOGIC_VECTOR(2 downto 0);
           i : in STD_LOGIC;
           q_next : out STD_LOGIC_VECTOR(2 downto 0);
           y : out STD_LOGIC );
end funzioni_tau_omega;

architecture Dataflow of funzioni_tau_omega is

begin
    y <= q(2) and (not i);
    q_next(0) <= ((not q(1)) and q(0) and i) or (((not q(2)) and (not q(0))) and ((not q(1)) and i) or ((not i) and q(1))) or (q(2) and (not i));

```

```

q_next(1) <= (q(2) and ((not i) or q(0))) or (q(1) and (not q(0)) and
(not i)) or ((not q(1)) and q(0));
q_next(2) <= (q(1) or (q(2) and (not q(0)))) and i;

end Dataflow;

```

Flip-flop D sincronizzato sul fronte di salita, con reset asincrono:

```

entity ff_D_async_rst is
  Port ( i : in STD_LOGIC;
         clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         q : out STD_LOGIC);
end ff_D_async_rst;

architecture Behavioral of ff_D_async_rst is
signal q_tmp : std_logic := '0';

begin
ff_D : process(clk, rst)
begin
  if (rst = '1') then
    q_tmp <= '0';
  elsif (clk = '1' and clk'event) then
    q_tmp <= i;
  end if;
end process;

q <= q_tmp;

end Behavioral;

```

Architettura “ibrida” che mette insieme il componente dataflow e quello behavioral:

```

architecture Hybrid of seq_sovr_1_dc_1_0 is
-- signals
signal q_next : std_logic_vector(2 downto 0) := (others => '0');
signal q : std_logic_vector(2 downto 0) := (others => '0');
signal y_tmp : std_logic := '0';
-- components
component funzioni_tau_omega is
  Port ( q : in STD_LOGIC_VECTOR(2 downto 0);
         i : in STD_LOGIC;
         q_next : out STD_LOGIC_VECTOR(2 downto 0);
         y : out STD_LOGIC );
end component;

component ff_D_async_rst is
  Port ( i : in STD_LOGIC;
         clk : in STD_LOGIC;

```

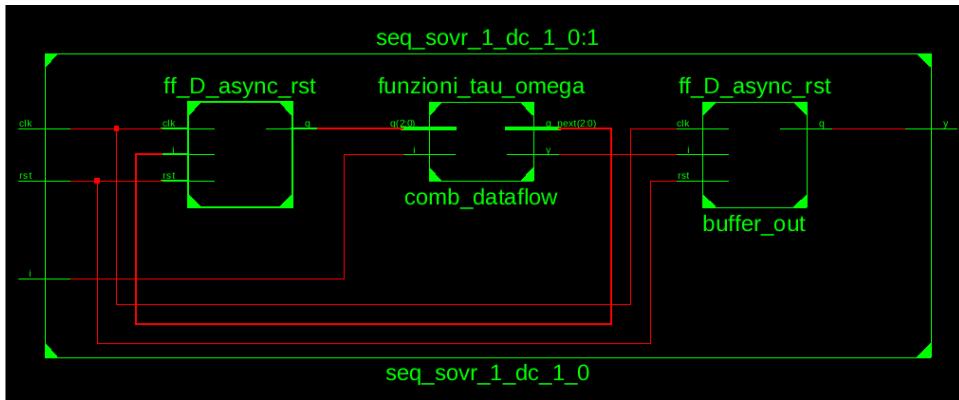
```

        rst : in  STD_LOGIC;
        q : out  STD_LOGIC);
end component;

begin
    comb_dataflow : funzioni_tau_omega port map(
        q => q, i => i, q_next => q_next, y => y_tmp
    );
    mem_state : for i in 2 downto 0 generate
        mem : ff_D_async_rst port map(
            i => q_next(i), clk => clk, rst => rst, q => q(i)
        );
    end generate;
    buffer_out : ff_D_async_rst port map(
        i => y_tmp, clk => clk, rst => rst, q => y
    );
end Hybrid;

```

L'architettura RTL ottenuta è la seguente:



2.4 Simulazione

2.4.1 Riconoscitore 1-1

Per la simulazione e la verifica di funzionamento del sistema 2-1, il quale deve riconoscere la sequenza 1-1, ogni tre bit posti in ingresso, viene creato il *testbench*, il cui codice è mostrato di seguito:

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY automa_tb IS
END automa_tb;

ARCHITECTURE behavior OF automa_tb IS
    -- Component Declaration for the Unit Under Test (UUT)

```

```

COMPONENT sequenza_1_dc_1
PORT(
    i : IN std_logic;
    CLK : IN std_logic;
    RESET : IN std_logic;
    Y : OUT std_logic
);
END COMPONENT;

--Inputs
signal i : std_logic := '0';
signal CLK : std_logic := '0';
signal RESET : std_logic := '1';

--Outputs
signal Y : std_logic;

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

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: sequenza_1_dc_1 PORT MAP (
        i => i,
        CLK => CLK,
        RESET => RESET,
        Y => Y
    );

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

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;
        RESET <= '0';

        -- insert stimulus here

        i      <= '0';

```

```

    wait for CLK_period;
    i    <= '1';
    wait for CLK_period;
    i    <= '1';
    wait for CLK_period;
    assert Y='0'
    report "errore0"
    severity failure;

    i    <= '1';
    wait for CLK_period;
    assert Y='0'
    report "errore1"
    severity failure;

    i    <= '0';
    wait for CLK_period;
    i    <= '1';
    wait for CLK_period;
    assert Y='1'
    report "errore2"
    severity failure;

    i    <= '0';
    wait for CLK_period;
    i    <= '1';
    wait for CLK_period;
    i    <= '1';
    wait for CLK_period;
    assert Y='0'
    report "errore3"
    severity failure;

    i    <= '1';
    wait for CLK_period;
    i    <= '1';
    wait for CLK_period;
    i    <= '1';
    wait for CLK_period;
    assert Y='1'
    report "errore4"
    severity failure;

    i    <= '1';
    wait for CLK_period;
    i    <= '1';
    wait for CLK_period;
    i    <= '0';
    wait for CLK_period;
    assert Y='0'

```

```

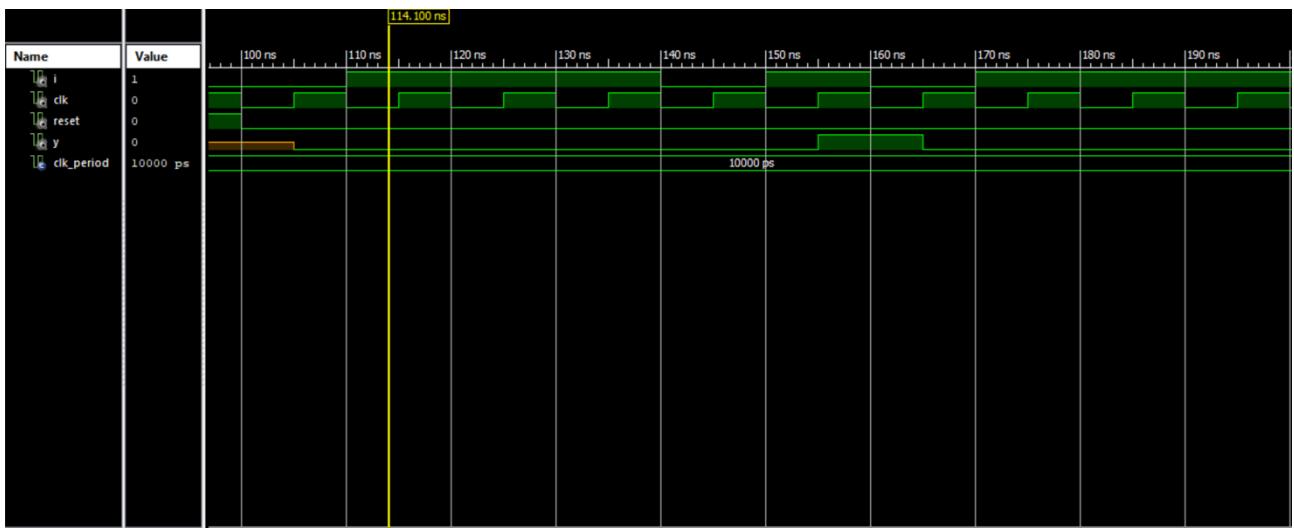
        report "errore5"
        severity failure;

      wait;
    end process;

END;

```

I risultati della simulazione sono riportati nella seguente figura:



2.4.2 Riconoscitore 1-10

Per la simulazione e la verifica di funzionamento del sistema 2-2, il quale deve riconoscere la sequenza 1-10, in maniera continua, ovvero permettendo di riconoscere anche sequenze che si sovrappongono tra loro, viene creato il *testbench*, il cui codice è mostrato di seguito:

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY automa_tb IS
END automa_tb;

ARCHITECTURE behavior OF automa_tb IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT seq_sovr_1_dc_1_0
PORT(
  i : IN std_logic;
  CLK : IN std_logic;

```

```

        RESET : IN  std_logic;
        Y : OUT  std_logic
    );
END COMPONENT;

--Inputs
signal i : std_logic := '0';
signal CLK : std_logic := '0';
signal RESET : std_logic := '1';

--Outputs
signal Y : std_logic;

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

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: seq_sovr_1_dc_1_0 PORT MAP (
        i => i,
        CLK => CLK,
        RESET => RESET,
        Y => Y
    );

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

    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;
        RESET <= '0';

        -- insert stimulus here

        i     <= '1';
        wait for CLK_period;
        i     <= '1';
        wait for CLK_period;

```

```

        i      <= '1';
        wait for CLK_period;
        i      <= '0';
        wait for CLK_period/2;
        assert Y='1'
        report "errore0"
        severity failure;

        wait for CLK_period/2;
        i      <= '1';
        wait for CLK_period;
        i      <= '0';
        wait for CLK_period/2;
        assert Y='1'
        report "errore1"
        severity failure;

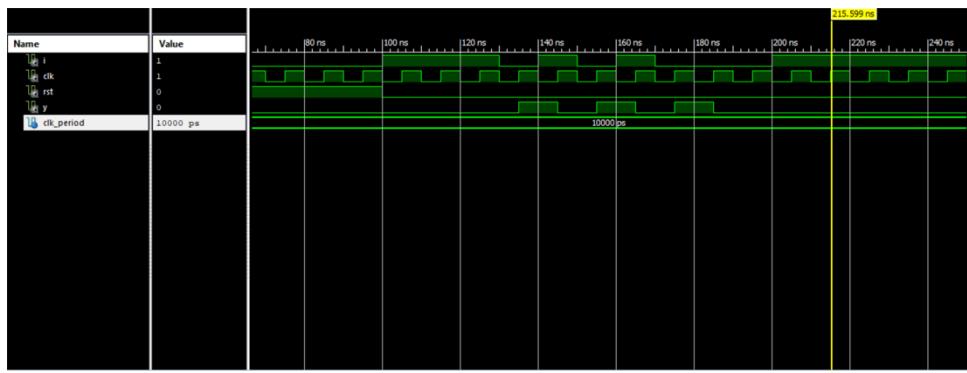
        wait for CLK_period/2;
        i      <= '1';
        wait for CLK_period;
        i      <= '0';
        wait for CLK_period/2;
        assert Y='1'
        report "errore2"
        severity failure;

        wait for CLK_period/2;
        i      <= '0';
        wait for CLK_period;
        i      <= '0';
        wait for CLK_period;
        i      <= '1';
        wait for CLK_period/2;
        assert Y='0'
        report "errore3"
        severity failure;

    wait;
end process;
END;

```

I risultati della simulazione sono riportati nella seguente figura:



Esercizio 3

3.1 Traccia

Esercizio 3

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

Il progetto deve prevedere la possibilità di inizializzare l'orologio con un valore iniziale, sempre espresso in termini di ore, minuti e secondi, mediante un opportuno ingresso di set, e deve prevedere un ingresso di reset per azzerare il tempo.

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

3.2 Soluzione

Per l'implementazione strutturale di un sistema di questo tipo, conviene utilizzare un approccio bottom-up. Per ottenere l'orologio, sono necessari i contatori mod M, con M generic. Per ottenere i contatori mod M, è necessaria una combinazione di un contatore a N bit, che quindi conta 2^N valori, e di un comparatore, il quale effettua il confronto tra l'uscita del contatore a N bit ed il valore M: l'uscita del comparatore si utilizza come reset del contatore. Per implementare il contatore a N bit, l'approccio strutturale consiste nell'utilizzare dei flip flop di tipo T, edge-triggered sul fronte di discesa. L'idea è di utilizzare l'uscita del flip flop T in posizione i come segnale di abilitazione del flip flop T in posizione $i+1$. Notare che se fossero sincronizzati sul fronte di salita, i flip flop commuterebbero praticamente tutti insieme, in cascata, e non si otterrebbe il risultato desiderato, che è una divisione di frequenza per 2 con ciascun flip flop: per questo è necessario che siano sincronizzati sul fronte di discesa. A questo punto, si hanno due possibilità:

- Utilizzare direttamente l'uscita di un flip flop come segnale di "clock" del flip flop successivo, utilizzando un ingresso costante pari ad 1 in ciascun flip flop: questo è lo schema seriale.
- Utilizzare lo stesso clock per tutti i flip flop, implementando la condizione di discesa allo stadio i usando come ingresso l'uscita di una AND tra tutte le precedenti uscite dei flip flop: questo è lo schema parallelo.

Lo schema parallelo ha il vantaggio di essere un sistema isocrono, ma non è una soluzione scalabile al crescere del numero di flip flop nella cascata, ovvero al crescere del numero di bit del contatore; inoltre, non è del tutto esente da problemi di tempificazione, perché nell'implementazione hardware potrebbe non essere disponibile una AND a N bit ma ad esempio solo AND a 2 bit: quindi una AND a N bit diventerebbe una cascata di AND a 2 bit e si avrebbero comunque degli sfasamenti temporali tra i bit, dato che ciascuna porta AND ha un tempo di risposta non nullo. Nello schema seriale si possono avere delle commutazioni spurie, dovute al fatto che prima commuta il flip flop in posizione i e al colpo di clock successivo commuta il flip flop in posizione $i+1$; questa cosa vale anche nel caso di interconnessione tra contatori, infatti un flip flop T non è altro che un contatore mod 2. Per risolvere questo problema con i contatori mod M, è possibile adottare la seguente strategia: si usano due comparatori, uno che effettua il confronto con M e che con la sua uscita effettua il reset del contatore, l'altro che effettua il confronto con $M-1$ e che con la sua uscita segnala al contatore successivo che deve commutare. Funziona perché i contatori commutano sul

fronte di discesa, quindi è giusto che il segnale di “overflow” deve alzarsi un colpo di clock prima di quello di reset in modo tale che il suo fronte di discesa sia sincronizzato con il reset. Questa strategia sarà applicata nella versione della soluzione adattata per la sintesi su FPGA; infatti per quanto detto, è abbastanza chiaro che la scelta è ricaduta sullo schema seriale. Visto che uno dei requisiti era il caricamento parallelo di ore, minuti e secondi ovvero la scrittura parallela nei flip flop, per ciascun flip flop è stato aggiunto un altro segnale di dato ed un segnale di set asincrono. Quando si mettono insieme i vari flip flop in un contatore, esso collega il segnale di set esterno con tutti i segnali di set dei flip flop e fa una cosa simile per il dato che riceve in parallelo. Non abbiamo implementato il salvataggio degli intertempi, ma una possibile soluzione era la seguente: utilizzare un buffer circolare di N registri, acceduto mediante un contatore mod N, la cui uscita abilita la scrittura di uno dei registri andando in AND con il segnale di stop; il segnale di stop, peraltro, funge anche da abilitazione del contatore mod N, facendogli effettuare un conteggio per ogni stop; l’uscita parallela dell’orologio è posta come ingresso di ciascuno degli N registri, dei quali solo uno alla volta è abilitato, come detto.

Concludendo, per implementare l’orologio bisogna quindi collegare tra loro dei contatori mod M:

- Un primo contatore funge da prescaler, serve per portare la frequenza della board da X Hz a 1 Hz, quindi è mod X.
- Il secondo è mod 60, conta i secondi: vi è collegato l’ingresso parallelo relativo ai secondi, e la sua uscita parallela è l’uscita parallela dell’interfaccia, relativa ai secondi, quindi è su 6 bit.
- Il terzo è mod 60, conta i minuti.
- L’ultimo è mod 24, conta le ore, e quindi è su 5 bit.

Anche per questo ci sarà una differenza in fase di sintesi su FPGA: un contatore realizzato in maniera strutturale mediante dei flip flop T, che sono degli elementi non presenti sulla scheda ma a loro volta composti da più elementi, non è in grado di operare da prescaler alla frequenza di funzionamento della scheda, perché è troppo lento. Per questo motivo, il primo contatore mod M è stato poi sostituito dal clock filter, che è un componente in grado di commutare alla velocità richiesta per operare da prescaler.

3.3 Codice

Seguendo l’approccio bottom-up, partiamo dalla base, cioè dal flip-flop T:

```
entity flip_flop_T is
    Port ( t : in STD_LOGIC;
           clk : in STD_LOGIC;
           set : in STD_LOGIC;
           d : in STD_LOGIC;
           rst : in STD_LOGIC;
           q : out STD_LOGIC);
end flip_flop_T;

architecture Behavioral of flip_flop_T is
signal q_tmp : STD_LOGIC := '0';

begin
ff_T: process(clk, rst)
begin
    if (rst = '1') then
        q_tmp <= '0';
    elsif (set = '1') then
        q_tmp <= d;
```

```

        elsif (clk'event and clk = '0') then
            if (t = '1') then
                q_tmp <= not q_tmp;
            end if;
        end if;
    end process;

q <= q_tmp;

end Behavioral;

```

Ora mettiamo insieme più flip flop T per realizzare un contatore generico a N bit:

```

entity contatore_N_bit is
    Generic (n : positive := 3);
    Port ( clk : in STD_LOGIC;
            rst : in STD_LOGIC;
            par : in STD_LOGIC_VECTOR(n-1 downto 0);
            set : in STD_LOGIC;
            y : out STD_LOGIC_VECTOR(n-1 downto 0));
end contatore_N_bit;

architecture Structural of contatore_N_bit is

signal y_tmp : STD_LOGIC_VECTOR(n-1 downto 0);

component flip_flop_T is
    Port ( t : in STD_LOGIC;
            clk : in STD_LOGIC;
            set : in STD_LOGIC;
            d : in STD_LOGIC;
            rst : in STD_LOGIC;
            q : out STD_LOGIC);
end component;

begin
frontend : flip_flop_T Port map(
    t => '1', clk => clk, set => set, d => par(0), rst => rst, q =>
y_tmp(0)
);
inner : for i in 1 to n-1 generate
    ff_t : flip_flop_T Port map(
        t => '1', clk => y_tmp(i-1), set => set, d => par(i), rst => rst,
q => y_tmp(i)
    );
end generate;

y <= y_tmp;

end Structural;

```

A questo punto, per implementare il contatore generico mod M, è necessario il componente comparator, implementato in modo behavioral in questo caso:

```

entity comparator is
    Generic (n : positive := 2);
    Port ( a : in STD_LOGIC_VECTOR(n-1 downto 0);
            b : in STD_LOGIC_VECTOR(n-1 downto 0);
            clk : in STD_LOGIC;
            y : out STD_LOGIC);
end comparator;

architecture Behavioral of comparator is

begin
comp : process(clk)
begin
    if (clk'event and clk = '0') then
        if a = b then
            y <= '1';
        else
            y <= '0';
        end if;
    end if;
end process;

end Behavioral;

```

Siamo pronti per il contatore mod M:

```

entity contatore_mod_M is
    -- M dovrebbe essere n_bit downto 0, ma deve rimanere unconstrained
    -- altrimenti
    -- dà errore a tempo di compilazione; NOTA: M è inclusivo, ovvero in
    -- realtà
    -- conta mod(M + 1)
    Generic (n_bit : positive := 3; M : std_logic_vector := "111");
    Port ( clk : in STD_LOGIC;
            rst : in STD_LOGIC;
            par : in STD_LOGIC_VECTOR(n_bit-1 downto 0);
            set : in STD_LOGIC;
            y : out STD_LOGIC_VECTOR(n_bit-1 downto 0);
            overflow : out STD_LOGIC);
end contatore_mod_M;

architecture Hybrid of contatore_mod_M is
signal y_tmp : std_logic_vector(n_bit-1 downto 0) := (others => '0');
signal rst_overflow : std_logic := '0';
signal u_RST : std_logic := '0';

```

```

component contatore_N_bit is
    Generic (n : positive := 3);
    Port ( clk : in STD_LOGIC;
            rst : in STD_LOGIC;
            par : in STD_LOGIC_VECTOR(n-1 downto 0);
            set : in STD_LOGIC;
            y : out STD_LOGIC_VECTOR(n-1 downto 0));
end component;

component comparator is
    Generic (n : positive := 2);
    Port ( a : in STD_LOGIC_VECTOR(n-1 downto 0);
            b : in STD_LOGIC_VECTOR(n-1 downto 0);
            clk : in STD_LOGIC;
            y : out STD_LOGIC);
end component;

begin
contatore : contatore_N_bit generic map(n_bit) port map(
    clk => clk, rst => u_RST, par => par, set => set, y => y_TMP
);

feedback : comparator generic map(n_bit) port map(
    a => M, b => y_TMP, clk => CLK, y => RST_OVERFLOW
);

MUX_RST : process(RST, RST_OVERFLOW)
begin
    if (RST = '1') then
        u_RST <= '1';
    else
        u_RST <= RST_OVERFLOW;
    end if;
end process;

OVERFLOW <= RST_OVERFLOW;
Y <= Y_TMP;

end Hybrid;

```

Infine, mettiamo tutto insieme per implementare l'orologio:

```

entity orologio is
    Port ( clk : in STD_LOGIC;
            rst : in STD_LOGIC;
            set : in STD_LOGIC;
            set_s : in STD_LOGIC_VECTOR(5 downto 0);
            set_m : in STD_LOGIC_VECTOR(5 downto 0);
            set_h : in STD_LOGIC_VECTOR(4 downto 0);
            s : out STD_LOGIC_VECTOR(5 downto 0));

```

```

        m : out STD_LOGIC_VECTOR(5 downto 0);
        h : out STD_LOGIC_VECTOR(4 downto 0));
end orologio;

architecture Structural of orologio is
-- costanti: frequenza del clock 50 MHz
constant n : positive := 26;
constant m_psc : std_logic_vector(n-1 downto 0) :=
"10111110101111000010000000";
-- per simulare:
--constant n : positive := 3;
--constant m_psc : std_logic_vector(n-1 downto 0) := "101";
-- segnali interni
signal clk_sec : std_logic := '0';
signal clk_min : std_logic := '0';
signal clk_h : std_logic := '0';
-- mi serve un segnale zero perché non posso mettere (others => '0') nel
port map
signal zero : std_logic_vector(n-1 downto 0) := (others => '0');
-- dispositivi interni
component contatore_mod_M is
    Generic (n_bit : positive := 3; M : std_logic_vector := "111");
    Port ( clk : in STD_LOGIC;
            rst : in STD_LOGIC;
            par : in STD_LOGIC_VECTOR(n_bit-1 downto 0);
            set : in STD_LOGIC;
            y : out STD_LOGIC_VECTOR(n_bit-1 downto 0);
            overflow : out STD_LOGIC);
end component;

begin
prescaler : contatore_mod_M generic map(n_bit => n, M => m_psc) port map(
    clk => clk, rst => rst, par => zero, set => set, overflow => clk_sec
);
-- 111011 = 59 in decimale
cont_sec : contatore_mod_M generic map(n_bit => 6, M => "111011") port
map(
    clk => clk_sec, rst => rst, par => set_s, set => set, overflow =>
clk_min,
    y => s
);
cont_min : contatore_mod_M generic map(n_bit => 6, M => "111011") port
map(
    clk => clk_min, rst => rst, par => set_m, set => set, overflow =>
clk_h,
    y => m
);
-- 10111 = 23 in decimale
cont_h : contatore_mod_M generic map(n_bit => 5, M => "10111") port map(
    clk => clk_h, rst => rst, par => set_h, set => set, y => h

```

```

);
end Structural;

```

Notare che questa non è la soluzione finale: questa è la soluzione che è stata oggetto di simulazione; alcuni problemi di questa soluzione non sono stati riscontrati oppure notati durante la simulazione, ma sono emersi in fase d'implementazione su FPGA. Nella sezione relativa alla sintesi su FPGA riporteremo gli ultimi raffinamenti.

3.4 Simulazione

La simulazione è stata un po' problematica a causa della natura del sistema; ad esempio, è stato necessario abbassare il numero di campioni del prescaler, non realizzando un effettivo conteggio dei secondi, perché altrimenti, a causa del tempo di step della simulazione, il tempo necessario per effettuare il test sarebbe stato enorme. Di seguito il codice del testbench:

```

ENTITY orologio_testbench IS
END orologio_testbench;

ARCHITECTURE behavior OF orologio_testbench IS

COMPONENT orologio
PORT(
    clk : IN std_logic;
    rst : IN std_logic;
    set : IN std_logic;
    set_s : IN std_logic_vector(5 downto 0);
    set_m : IN std_logic_vector(5 downto 0);
    set_h : IN std_logic_vector(4 downto 0);
    s : OUT std_logic_vector(5 downto 0);
    m : OUT std_logic_vector(5 downto 0);
    h : OUT std_logic_vector(4 downto 0)
);
END COMPONENT;

signal clk : std_logic := '0';
signal rst : std_logic := '0';
signal set : std_logic := '0';
signal set_s : std_logic_vector(5 downto 0) := (others => '0');
signal set_m : std_logic_vector(5 downto 0) := (others => '0');
signal set_h : std_logic_vector(4 downto 0) := (others => '0');

signal s : std_logic_vector(5 downto 0);
signal m : std_logic_vector(5 downto 0);
signal h : std_logic_vector(4 downto 0);

-- 50 MHz di frequenza -> 20 ns di periodo
constant clk_period : time := 20 ns;

BEGIN

```

```

uut: orologio PORT MAP (
    clk => clk,
    rst => rst,
    set => set,
    set_s => set_s,
    set_m => set_m,
    set_h => set_h,
    s => s,
    m => m,
    h => h
);

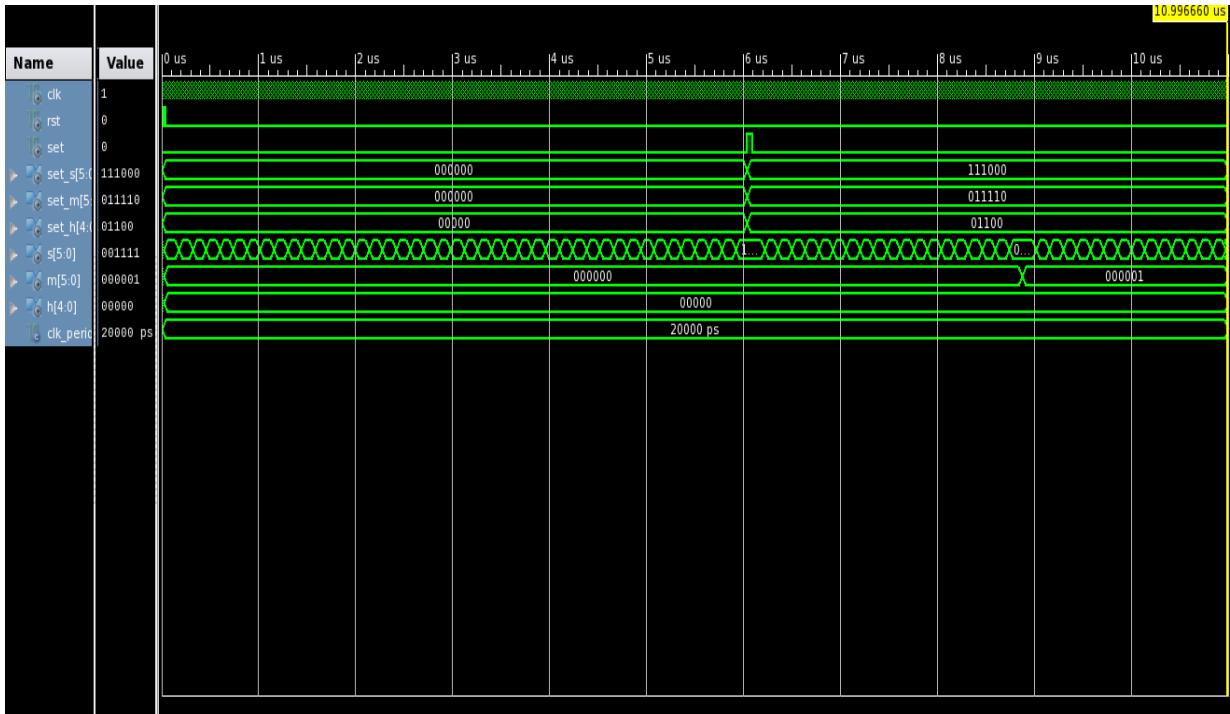
clk_process :process
begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
end process;

stim_proc: process
begin
    -- NOTA: simulare usando il frontend modulo 5 al posto di 50
    -- milioni;
    -- simulare per un tempo di 10 us
    wait for clk_period;
    rst <= '1';
    wait for clk_period;
    rst <= '0';
    wait for clk_period*300;
    set_s <= "111000";
    set_m <= "011110";
    set_h <= "01100";
    set <= '1';
    wait for clk_period*3;
    set <= '0';
    wait;
end process;

END;

```

Risultati della simulazione:



3.5 Sintesi su FPGA

L'FPGA su cui è stata fatta la sintesi è la Nexys A7 50-T. Essa ha un clock di 100 MHz. Come accennato nella soluzione, il prescaler presente nell'orologio è stato sostituito dal componente clock filter, che ci è stato fornito dai Docenti:

```
entity clock_filter is
  generic(
    clock_frequency_in : integer := 50000000;
    clock_frequency_out : integer := 5000000
  );
  Port ( clock_in : in STD_LOGIC;
         reset_n : in STD_LOGIC;
         clock_out : out STD_LOGIC);
end clock_filter;

architecture Behavioral of clock_filter is

signal clockfx, reset : std_logic := '0';

constant count_max_value : integer := (clock_frequency_in/(clock_frequency_out))-1;

begin

clock_out <= clockfx;
reset <= not reset_n;

count_for_division: process(clock_in, reset)

```

```

variable counter : integer range 0 to count_max_value := 0;
begin

    if reset = '1' then
        counter := 0;
        clockfx <= '0';
    elsif clock_in'event and clock_in = '1' then
        if counter = count_max_value then
            clockfx <= '1';
            counter := 0;
        else
            clockfx <= '0';
            counter := counter + 1;
        end if;
    end if;

end process;

```

Nell'orologio è stato quindi inserito il nuovo prescaler:

```

prescaler : clock_filter generic map(clock_frequency_in =>
100000000, clock_frequency_out => 1) port map(
    clock_in => clock, reset_n => reset_n, clock_out => clock_sec
);

```

La frequenza d'uscita è 1Hz in quanto il contatore che riceve il segnale di clock in ingresso è il counter dei secondi.

Un'altra modifica che è stata fatta riguarda i valori di M dei contatori: per quello della simulazione sono stati inizialmente passati come valori generic le rappresentazioni binarie di 60, 60 e 24, ma in realtà per come è stato implementato il contatore mod M, era necessario passare 59, 59 e 23, e così è stato fatto nella versione della sintesi su FPGA.

Per effettuare la sintesi su FPGA in modo tale da visualizzare in qualche modo l'output e controllare alcuni segnali di input, abbiamo usato i bottoni in input e il display a 7 segmenti per visualizzare l'output. Per la gestione di Input/Output abbiamo apportato delle lievi modifiche a dei moduli forniti dai Docenti, i quali erano stati implementati per la Nexys A2:

- Per l'input dai bottoni, l'unica cosa da fare è modificare opportunamente il configuration file, per “saldare” i fili fisici relativi ai bottoni con i fili simbolici che si trovano in ingresso all'interfaccia del top module.
- Per l'output al display a 7 segmenti, nella Nexys A7 si hanno 8 cifre, mentre ce n'erano 4 per la Nexys A2, quindi le modifiche fatte ai moduli riguardavano principalmente questo aspetto; a tal proposito, osserviamo la modalità di funzionamento del display a 7 segmenti, che prevede che una cifra alla volta sia accesa mediante la tensione applicata all'anodo (e quindi si deve avere uno scorrimento abbastanza veloce tra le cifre per dare l'impressione che siano tutte accese contemporaneamente), e c'è un valore di tensione alto o basso per ogni segmento, più uno per il punto: questa “stringa” va in ingresso a tutte le cifre, ma come detto una alla volta è accesa, quindi con un contatore si può scorrere sul vettore di abilitazione degli anodi e sul vettore in cui è presente la codifica esadecimale delle cifre.

Per mostrare secondi, minuti e ore sul display a 7 segmenti, è necessario estrarre le cifre in base decimale relative a decine ed unità: esse sono pari rispettivamente alla divisione intera per 10 e al resto modulo 10. Abbiamo quindi implementato in maniera Behavioral un modulo che realizza tali operazioni:

```

use IEEE.NUMERIC_STD.ALL;

entity translate_count is
port ( x : in std_logic_vector(5 downto 0);
       y : out std_logic_vector(7 downto 0));
end translate_count;

architecture Behavioral of translate_count is

begin
  process(x)
    variable var : integer range 0 to 63;
    variable count,i : integer range 0 to 64;
    variable b : integer;

    begin
      b := 10;
      i:=0; var:=to_integer(unsigned(x)); count:=0;
      for i in 63 downto 0 loop
        if (var>=b) then
          var := var-b;
          count:= count+1;
        else
          y      <=      std_logic_vector(to_unsigned(count,      4))      &
to_unsigned(var, 4));
          exit;
        end if;
      end loop;
      end process;
  end Behavioral;

```

Il top module è “display_on_board”, che è una classe control che contiene sia il componente per gestire l’output sul display, il quale a sua volta contiene il manager degli anodi ed il manager dei catodi, sia la “control_unit” che è il componente in cui è presente l’effettiva logica che si va a sintetizzare. Nel top module l’unica cosa modificata è il parallelismo dei dati in ingresso al componente per gestire l’output sul display, oltre ovviamente all’interfaccia della control unit. Riportiamo solo la nostra control unit, dato che gli altri moduli ci sono stati forniti dai Docenti. Abbiamo supposto di avere solo un bottone per il reset ed uno per il preset, cioè il caricamento parallelo di una word: per semplicità abbiamo messo la word hardcoded, cioè con delle stringhe di bit costanti in ingresso all’orologio. Ecco quindi la control unit:

```

entity control_unit is
  Port (
    clock : in STD_LOGIC;
    reset_n : in STD_LOGIC;
    preset : in STD_LOGIC;
    value : out STD_LOGIC_VECTOR(31 downto 0);
    enable : out STD_LOGIC_VECTOR(7 downto 0)
      );
end control_unit;

```

```

architecture Structural of control_unit is

signal reset : std_logic;
signal u_s : std_logic_vector(5 downto 0) := (others => '0');
signal u_m : std_logic_vector(5 downto 0) := (others => '0');
signal u_h : std_logic_vector(5 downto 0) := (others => '0');

component orologio is
    Port ( clock : in STD_LOGIC;
            reset : in STD_LOGIC;
            preset : in STD_LOGIC;
            set_s : in STD_LOGIC_VECTOR(5 downto 0);
            set_m : in STD_LOGIC_VECTOR(5 downto 0);
            set_h : in STD_LOGIC_VECTOR(4 downto 0);
            s : out STD_LOGIC_VECTOR(5 downto 0);
            m : out STD_LOGIC_VECTOR(5 downto 0);
            h : out STD_LOGIC_VECTOR(4 downto 0));
end component;

component translate_count is
port ( x : in std_logic_vector(5 downto 0);
       y : out std_logic_vector(7 downto 0));
end component;

begin

reset <= not reset_n;
enable <= "11111111";

cronometro : orologio port map(
    -- il preset hardcoded e' 11h, 17m e 50s
    clock => clock, reset => reset, preset => preset, set_s => "110010",
    set_m => "010001", set_h => "01011", s => u_s,
    m => u_m, h => u_h(4 downto 0)
);

translate_s : translate_count port map(
    x => u_s, y => value(7 downto 0)
);

translate_m : translate_count port map(
    x => u_m, y => value(15 downto 8)
);

translate_h : translate_count port map(
    x => u_h, y => value(23 downto 16)
);

value(31 downto 24) <= (others => '0');

```

```
end Structural;
```

L'ultimo accorgimento da prendere era l'overflow anticipato nel contatore mod M, di cui ora riportiamo solo l'architecture body:

```
contatore : contatore_N_bit generic map(n_bit) port map(
    clock => clock, reset => u_reset, par => par, preset => preset, y =>
y_tmp
);

feedback : comparator generic map(n_bit) port map(
    a => M, b => y_tmp, clock => clock, y => reset_overflow
);

propagation : comparator generic map(n_bit) port map(
    a => std_logic_vector(to_unsigned(to_integer(unsigned(M))-1, n_bit)),
b => y_tmp, clock => clock, y => overflow
);

mux_reset : process(reset, reset_overflow)
begin
    if (reset = '1') then
        u_reset <= '1';
    else
        u_reset <= reset_overflow;
    end if;
end process;

y <= y_tmp;
```

Abbiamo quindi verificato sull'FPGA che non ci sono più commutazioni spurie, cioè si ha:

1. 00:00:00:59
2. 00:00:01:00
3. 00:00:01:01

Se non si fa così, a seconda dell'implementazione succede o che si passa da 00:59 a 00:00 e poi 01:01, oppure da 00:59 a 01:00, poi di nuovo 01:00 ed infine 01:01, per un problema di scheduling dei segnali: lo scheduling della commutazione del contatore successivo dev'essere fatto un quanto di tempo prima del reset. Anche il caricamento parallelo funziona come previsto, ed il tempo è scandito in modo preciso (ovviamente non l'abbiamo osservato per un tempo tale da notare un clock drift, ma se non altro abbiamo verificato per confronto con un altro orologio che il sistema scandisce effettivamente i secondi).

Come ultima cosa aggiungiamo le **constraints** utilizzate, solo la parte non commentata.

```
## Clock signal
set_property -dict { PACKAGE_PIN E3      IO_STANDARD LVCMOS33 } [get_ports {
clock }];
create_clock -add -name sys_clock_pin -period 10.00 -waveform {0 5}
```

```

[get_ports {clock}];

##7 segment display
set_property -dict { PACKAGE_PIN T10    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[1] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[2] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[3] }]; #IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[6] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17    IO_STANDARD LVCMOS33 } [get_ports {
anodes[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18    IO_STANDARD LVCMOS33 } [get_ports {
anodes[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9     IO_STANDARD LVCMOS33 } [get_ports {
anodes[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14    IO_STANDARD LVCMOS33 } [get_ports {
anodes[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14    IO_STANDARD LVCMOS33 } [get_ports {
anodes[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14    IO_STANDARD LVCMOS33 } [get_ports {
anodes[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2     IO_STANDARD LVCMOS33 } [get_ports {
anodes[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13    IO_STANDARD LVCMOS33 } [get_ports {
anodes[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons
set_property -dict { PACKAGE_PIN N17    IO_STANDARD LVCMOS33 } [get_ports {
reset }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18    IO_STANDARD LVCMOS33 } [get_ports {
preset }]; #IO_L4N_T0_D05_14 Sch=btnu

```


Esercizio 4

4.1 Traccia

Esercizio 4

Progettare un registro a scorrimento di 4 bit in grado di operare, in base ad una selezione, in 4 diverse modalità: (1) scorrimento a sinistra con caricamento seriale di un bit pari a 0, (2) scorrimento a destra con caricamento seriale di un bit pari a 0, (3) scorrimento circolare verso sinistra, (4) scorrimento a sinistra con caricamento seriale di un bit x.

Il valore iniziale del registro può essere configurato mediante un segnale di reset oppure tramite il caricamento parallelo di un valore A3A2A1A0 fornito dall'esterno, inserito grazie ad un segnale di load. Un segnale di shift regola lo scorrimento del registro.

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

4.2 Soluzione

Il registro a scorrimento (o Shift Register) è costituito da una catena di celle di memoria ad 1 bit interconnesse tra loro, implementate mediante flip-flop di tipo D rise edge-triggered. Ad ogni impulso di clock (**CLK**) essi consentono lo scorrimento dei bit da una cella a quella immediatamente adiacente. In particolare, in base alla selezione di una delle quattro diverse modalità, fatta mediante il segnale **SHIFT** è consentito lo scorrimento a sinistra con caricamento seriale di un bit pari a 0, lo scorrimento a destra con caricamento seriale di un bit pari a 0, lo scorrimento circolare verso sinistra, nel quale l'uscita dell'ultimo registro, corrispondente all'uscita (**Y**) del registro complessivo, ritorna in ingresso al primo registro e infine, lo scorrimento a sinistra con caricamento seriale di un bit (**X**). Inoltre, è possibile configurare il valore iniziale del registro mediante un segnale **RESET**, che pone a 0 tutti i valori memorizzati dai flip-flop che compongono la catena, o mediante il caricamento parallelo di un valore **A** di 4 bit fornito dall'esterno, inserito tramite il segnale **LOAD**.

Nell'approccio strutturale sono stati implementati i flip-flop D rise edge-triggered. Essi definiscono i registri interni di 1 bit che compongono la catena. Sono stati utilizzati quattro flip-flop poiché l'obiettivo è costruire un registro a quattro bit. Tutti i flip-flop ricevono lo stesso clock in ingresso al registro.

Inoltre, sono stati progettati dei multiplexer 4:1 anch'essi con un approccio Structural, essi sono composti da multiplexer 2:1 e permettono di scegliere, tramite un ingresso di selezione, la modalità di funzionamento, ponendo in base a quest'ultimo l'ingresso dei flip-flop opportuno. Sono presenti 4 multiplexer, uno per ogni flip-flop, poiché devono gestirne l'ingresso. Il segnale di selezione è comune a tutti i multiplexer ed è rappresentato dal segnale **SHIFT** sopra citato, questo è costituito da 2 bit.

Nell'approccio behavioral è stato utilizzato un solo costrutto process, nel quale viene descritto il comportamento del registro con un algoritmo. Sul fronte di salita del clock in ingresso al sistema è possibile in base al segnale campionato, resettare il registro (reset), precaricare il valore dei flip-flop (load), oppure anche in questo caso, operare in una delle quattro modalità definite mediante un costrutto case.

4.3 Codice

4.3.1 Approccio Strutturale

Il seguente codice descrive l'implementazione dello Shift Register risolto con il primo approccio, strutturale:

```
entity REG_SS_4_BIT is
    generic(N: integer :=4);
    Port ( CLK : in STD_LOGIC;
            SHIFT : in STD_LOGIC_VECTOR(1 downto 0);
            LOAD : in STD_LOGIC;
            A : in STD_LOGIC_VECTOR(0 to N-1);
            RESET : in STD_LOGIC;
            X : in STD_LOGIC;
            Y : out STD_LOGIC);
end REG_SS_4_BIT;

architecture Structural of REG_SS_4_BIT is

    signal Tp : STD_LOGIC_VECTOR(0 to N-1) := (others => '0');--uscite flip-flop
    signal T : STD_LOGIC_VECTOR(0 to N-1) := (others => '0');--uscite mux

    component MUX4_1
        port( I : in STD_LOGIC_VECTOR(0 to 3);
              S : in STD_LOGIC_VECTOR(1 downto 0);
              U : out STD_LOGIC
            );
    end component;

    component FF_D
        port( CLK : in STD_LOGIC;
              RESET : in STD_LOGIC;
              A : in STD_LOGIC;
              LOAD : in STD_LOGIC;
              D : in STD_LOGIC;
              Q : out STD_LOGIC
            );
    end component;

    component REGISTRO
        port( CLK : in STD_LOGIC;
              RESET : in STD_LOGIC;
              y : in STD_LOGIC;
              U : out STD_LOGIC
            );
    end component;

begin
```

```

mux0: MUX4_1
Port map( I(0) => '0',
            I(1) => Tp(1),
            I(2) => Tp(N-1),
            I(3) => X,
            S => SHIFT,
            U =>T(0)
            );
mux1to2: FOR j IN 1 TO N-2 GENERATE
m: MUX4_1
Port map( I(0) => Tp(j-1),
            I(1) => Tp(j+1),
            I(2) => Tp(j-1),
            I(3) => Tp(j-1),
            S => SHIFT,
            U =>T(j)
            );
END GENERATE;

muxN_1: MUX4_1
Port map( I(0) => Tp(N-2),
            I(1) => '0',
            I(2) => Tp(N-2),
            I(3) => Tp(N-2),
            S => SHIFT,
            U =>T(N-1)
            );
ff_t0toN_2: FOR k IN 0 TO N-2 GENERATE
ffd: FF_D
Port map( CLK =>CLK,
            RESET => RESET,
            A => A(N-1-k),
            LOAD => LOAD,
            D =>T(k),
            Q =>Tp(k)
            );
END GENERATE;

```

```

ff_dN_1: FF_D
Port map( CLK =>CLK,
            RESET => RESET,
            A => A(0),
            LOAD => LOAD,
            D =>T(N-1),
            Q =>Tp(N-1)
            );

```

```

REG: REGISTRO
Port map( CLK => CLK,
           RESET => RESET,
           y => Tp(N-1),
           U =>y
         );
end Structural;

```

Di seguito è presente il codice delle componenti utilizzate per implementare la strutturale del registro, Flip flop D:

```

entity FF_D is
  Port ( CLK : in STD_LOGIC;
         RESET : in STD_LOGIC;
         A : in STD_LOGIC;
         LOAD : in STD_LOGIC;
         D : in STD_LOGIC;
         Q : out STD_LOGIC);
end FF_D;

architecture Behavioral of FF_D is

signal QINTERNAL: std_logic;

begin
ff: process(CLK)
begin
  if(CLK'event and CLK = '1') then
    if(RESET = '1') then
      QINTERNAL <= '0';
    elsif(LOAD = '1') then
      QINTERNAL <= A;
    else
      QINTERNAL <= D;
    end if;
  end if;
end process;

Q <= QINTERNAL;

end Behavioral;

```

Di seguito il codice relativo al multiplexer 4:1 e al multiplexer 2:1 che lo compone:

```

entity MUX4_1 is
  Port ( I : in STD_LOGIC_VECTOR(0 to 3);
         S : in STD_LOGIC_VECTOR(1 downto 0);

```

```

        U : out STD_LOGIC);
end MUX4_1;

architecture Structural of MUX4_1 is
signal Y : STD_LOGIC_VECTOR(0 to 1) := (others => '0');
component MUX2_1
port( I : in STD_LOGIC_VECTOR(1 downto 0);
S : in STD_LOGIC;
U : out STD_LOGIC
);
end component;

begin
mux0to1: FOR j IN 0 TO 1 GENERATE
m: MUX2_1
Port map( I => I(j*2 to j*2+1),
S => S(0),
U => Y(j)
);
END GENERATE;

mux2: MUX2_1
Port map( I => Y(0 to 1),
S => S(1),
U => U
);

end Structural;

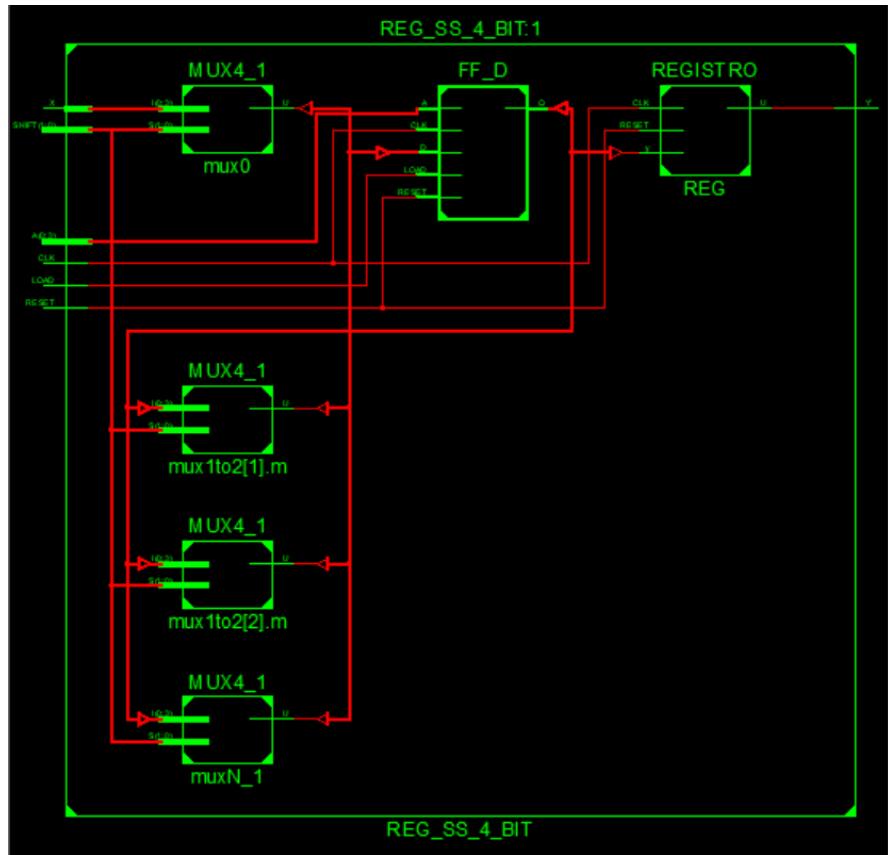
entity MUX2_1 is
  Port ( I : in STD_LOGIC_VECTOR(0 to 1);
         S : in STD_LOGIC;
         U : out STD_LOGIC);
end MUX2_1;

architecture Dataflow of MUX2_1 is

begin
  U <= ((I(0) AND (NOT S)) OR (I(1) AND S));
end Dataflow;

```

Di seguito è rappresentato lo schematico relativo allo shift register con approccio strutturale.



4.3.2 Approccio comportamentale

Di seguito è presente il codice per l'implementazione con approccio behavioral dello Shift Register:

```

entity REG_SS_4_BIT is
generic( N: integer := 4);
  Port ( CLK : in STD_LOGIC;
         SHIFT : in STD_LOGIC_VECTOR(1 DOWNTO 0);
         LOAD: in STD_LOGIC;
         A: in STD_LOGIC_VECTOR(0 TO N-1);
         RESET : in STD_LOGIC;
         X : in STD_LOGIC;
         Y : out STD_LOGIC);
end REG_SS_4_BIT;

architecture Behavioral of REG_SS_4_BIT is

signal T: std_logic_vector(0 to N-1);
signal Yinternal: std_logic;

begin

```

```

reg: PROCESS (CLK)
BEGIN
if(CLK'event and CLK = '1') then
    if(RESET = '1') then
        T <= (others => '0');
        Yinternal <= T(N-1);
    elsif(LOAD = '1') then
        T <= A;
        Yinternal <= T(N-1);
    else
        CASE SHIFT IS
            WHEN "00" =>
                T(0) <= '0';
                T(1 to N-1) <= T(0 to N-2);
                Yinternal <= T(N-1);
            WHEN "01" =>
                T(0 to N-2) <= T(1 to N-1);
                T(N-1) <= '0';
            WHEN "10" =>
                T(0) <= T(3);
                T(1 to N-1) <= T(0 to N-2);
                Yinternal <= T(N-1);
            WHEN "11" =>
                T(0) <= X;
                T(1 to N-1) <= T(0 to N-2);
                Yinternal <= T(N-1);
            WHEN OTHERS =>
                T <=(others => '0');
        END CASE;
    end if;
end if;
END PROCESS;

Y <= Yinternal;

end Behavioral;

```

4.4 Simulazione

Per la simulazione e la verifica di funzionamento del registro viene creato il *testbench*, il cui codice è mostrato di seguito:

```

ENTITY REG_SS_4_BIT_tb IS
END REG_SS_4_BIT_tb;

ARCHITECTURE behavior OF REG_SS_4_BIT_tb IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT REG_SS_4_BIT
PORT(
    CLK : IN std_logic;
    SHIFT : IN std_logic_vector(1 downto 0);
    LOAD : IN std_logic;
    A : IN std_logic_vector(0 to 3);
    RESET : IN std_logic;
    X : IN std_logic;
    Y : OUT std_logic
);
END COMPONENT;

--Inputs
signal CLK : std_logic := '0';
signal SHIFT : std_logic_vector(1 downto 0) := (others => '0');
signal LOAD : std_logic := '0';
signal A : std_logic_vector(0 to 3) := (others => '0');
signal RESET : std_logic := '1';
signal X : std_logic := '0';

--Outputs
signal Y : std_logic;

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

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: REG_SS_4_BIT PORT MAP (
    CLK => CLK,
    SHIFT => SHIFT,
    LOAD => LOAD,
    A => A,
    RESET => RESET,
    X => X,
    Y => Y
);

-- Clock process definitions
CLK_process :process
begin

```

```

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

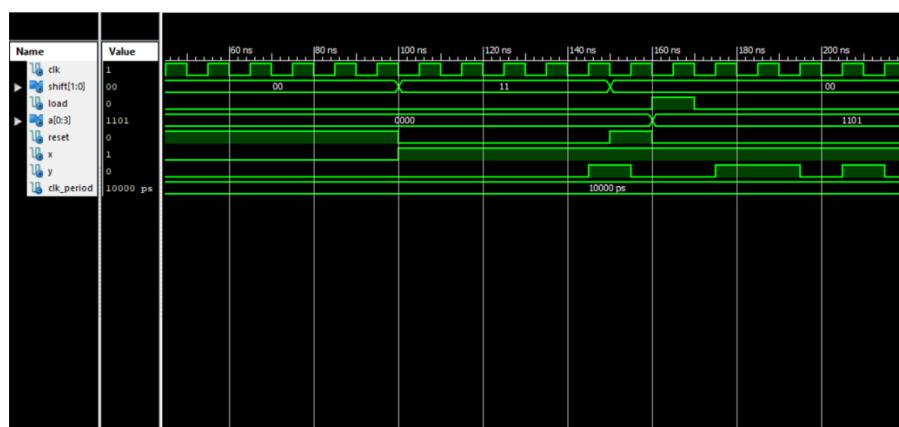
-- Stimulus process
stim_proc: process
begin
-- hold reset state for 100 ns.
wait for 100 ns;
-- insert stimulus here

RESET <= '0';
SHIFT <= "11";
X <= '1';
wait for CLK_period*5;
SHIFT <="00";
RESET <='1';
wait for CLK_period;
RESET <='0';
A <= "1101";
LOAD <='1';
wait for CLK_period;
LOAD <='0';
wait for CLK_period*4;
assert Y='1'
report "Errore"
severity failure;

wait;
end process;
END;

```

I risultati della simulazione sono riportati nella seguente figura:



Esercizio 5

5.1 Traccia

Progettare e implementare in VHDL un sistema che, date due stringhe binarie A e B di 8 bit ciascuna acquisite mediante handshaking, calcoli il valore **A mod B**.

Il sistema deve essere progettato utilizzando un approccio modulare basato sull'individuazione della parte operativa e della parte di controllo, e la parte di controllo deve essere realizzata mediante (a) **logica cablata** e (b) **logica micropogrammata**.

Con riferimento alle modalità di acquisizione delle stringhe in input mediante handshaking, si discutano due diverse soluzioni possibili.

5.2 Soluzione

La soluzione proposta, per questo sistema, consiste nell'affidare all'unità operativa il compito di fare delle operazioni di sottrazione che, con un opportuno feedback, portano a calcolare il valore di A mod B. È chiaro che l'unità operativa non sa quante iterazioni deve fare, e deve avere dal punto di vista logico tre diversi "stati": uno in cui effettua la sottrazione tra due valori che entrano dall'esterno, che corrisponde alla prima sottrazione $A - B$; uno che si itera più volte, in cui la sottrazione è effettuata tra il valore in feedback e B, quindi dal punto di vista logico si ha:

$$A_1 = A_0 - B$$

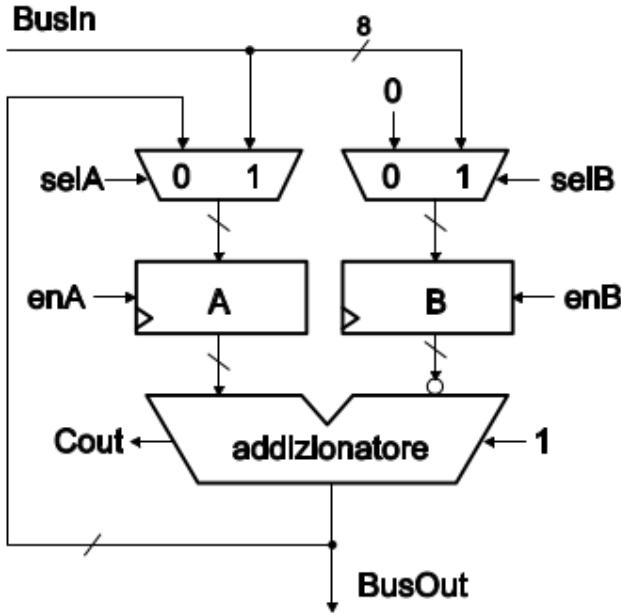
$$A_2 = A_1 - B$$

...

$$A_n = A_{n-1} - B$$

Quand'è che si deve fermare? Risposta: quando risulta $A_n < 0$, e quindi risulterà che:

$A_{n-1} = A_0 \text{ mod } B$, dove A_0 è il valore di A inserito dall'esterno. Visto che alla fine si vuole fornire in uscita il risultato di A mod B, il valore in "underflow" non deve sovrascrivere il valore precedente presente nel registro che memorizza il risultato del **feedback**, altrimenti si otterrebbe un risultato transitorio, mentre invece il risultato in uscita dev'essere reso disponibile fino al suo consumo. Per garantire questa proprietà, è necessario che l'unità operativa segnali uno "stato di terminazione" all'unità di controllo, generato proprio dalla sottrazione che ha dato un risultato negativo. In risposta, l'unità di controllo deve fare due cose: impedire che il registro in cui ora è presente A mod B sia sovrascritto, e fare in modo che in uscita sia effettivamente presente A mod B, quindi in pratica sovrascrivere con il valore 0 il registro in cui si trova B, e questo è il terzo "stato". Notare che sia il registro in cui inizialmente va A e poi va l'uscita in feedback, sia il registro in cui si trova per n iterazioni B e poi 0, hanno due possibili ingressi, la cui selezione è gestita dall'unità di controllo. Questo vuol dire che per ciascuno di questi registri c'è un MUX 2:1. Inoltre, per evitare interferenze esterne con B e per evitare che il feedback sia troppo veloce da gestire per l'unità di controllo, è opportuno che i registri siano triggerati da dei segnali di abilitazione. Per cui, questa è l'architettura strutturale dell'unità operativa:



A questo punto, parte dell'unità di controllo si progetta direttamente a partire dall'unità operativa: nella fase in cui A e B sono già stati caricati, si controlla se Cout (che rappresenta la condizione di underflow) è pari ad 1, e si ha costantemente selA=0 perché ci si predisponde per caricare l'uscita in feedback nel registro A, e selB conviene cominciare a metterlo a 0, tra poco sarà noto il perché. Se Cout è 0 vuol dire che bisogna caricare l'uscita in feedback nel registro A: visto che quando l'unità di controllo riceve il valore di Cout l'uscita in feedback sarà in ingresso al mux, l'unità di controllo può schedulare l'abilitazione enA per il quanto di tempo successivo, perché quando enA arriverà ad A, il mux avrà già fornito in uscita il nuovo valore e quindi non si hanno problemi di tempi di risposta relativi tra i componenti. Notare che per implementare correttamente quanto detto, il registro dovrà essere edge-triggered, e quindi l'unità di controllo dovrà opportunamente commutare enA da 0 ad 1 e viceversa. Quando Cout diventa 1, come detto non bisogna sovrascrivere il contenuto del registro A (che in tale istante vale A mod B), quindi enA non varia, ma bisogna fornire A mod B in uscita quindi scrivere 0 nel registro B: in teoria bisogna selezionare l'ingresso pari a 0 ponendo selB a 0 ed abilitare la scrittura nel registro facendo variare enB; tuttavia, facendo così, si scriverebbe il valore B nel registro B, perché il mux e il registro operano in parallelo e l'uscita del mux è ancora il valore B. Per risolvere questo problema, prima abbiamo detto che "selB conviene cominciare a metterlo a 0": questo problema di tempistiche era il motivo; così facendo, si ha che l'uscita del mux è già il valore 0, e ci si può limitare a far variare il segnale di abilitazione enB.

La restante parte dell'unità di controllo implementa **l'handshaking**. Il requisito è che le stringhe A e B siano acquisite mediante handshaking; in realtà, anche l'uscita dovrebbe essere gestita mediante handshaking, dato che altrimenti il sistema "client" non ha modo di sapere quando il calcolo è terminato. Questa modalità di gestione dell'handshaking è analoga al modello logico di handshaking completo, solo che in realtà l'handshaking non è un unico, grande handshaking "monolitico", ma è gestito in maniera modulare mediante più handshaking. A tal proposito, anche un singolo handshaking può essere implementato in due diversi modi, che hanno in comune la prima parte.

Client:

- Invio del dato
- Invio dello start
- Attesa dell'ack

Server:

- Attesa dello start
- Consumo del dato ed invio dell'ack

Il secondo modo prevede che il client invii una conferma della conferma, cioè un altro ack, dopo aver ricevuto l'ack del server, e quindi che il server attenda l'ack del client prima di procedere. Entrambi i modi sono validi, ma nel primo caso bisogna stare più attenti ai tempi relativi delle operazioni. Quindi, uno di questi due possibili handshake viene eseguito una volta per inviare A ed una volta per inviare B (“in A” e “in B”), dopodiché client e server si scambiano di posto e la macchina che ha calcolato A mod B avvia un terzo handshake per segnalare al client che il dato è pronto (un “end of operation”).

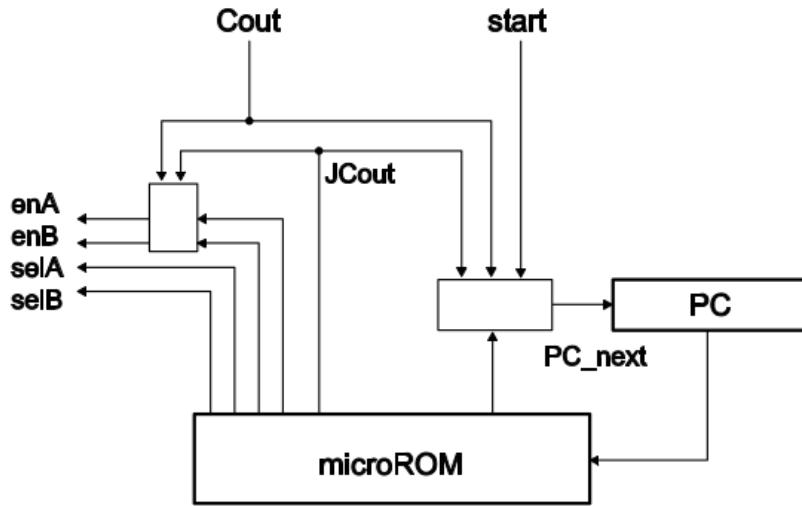
Ora siamo pronti per progettare l’unità di controllo: bisogna capire come farla in logica cablata e come farla in logica microprogrammata. In ogni caso, notare che, pur supponendo asincrono il protocollo di comunicazione tra i dispositivi, è necessario che l’unità di controllo sia sincronizzata mediante un clock, il quale deve essere caratterizzato da un periodo sufficientemente lungo da permettere all’unità operativa di fornire un nuovo risultato.

Dunque, in **logica cablata** si ha praticamente un automa a stati finiti. Dal punto di vista logico, gli stati sono i seguenti:

- Uno stato **idle**, in cui la macchina è pronta ad entrare in azione, in attesa che il client avvii l’handshake.
- Uno stato **in1** nel quale si consuma l’ingresso A, concludendo l’handshake ed inviando all’unità operativa gli opportuni comandi, come descritto in precedenza; ripetendo quanto detto prima, in questo stato l’unità di controllo deve prima impostare il segnale di selezione, e poi in un quanto di tempo successivo abilitare il registro in scrittura: per implementare questo aspetto più agevolmente, può essere conveniente che il server aspetti la conferma della conferma da parte del client.
- Dopo la fine del primo handshake, l’unità di controllo attende l’inizio di un secondo handshake per andare nello stato **in2**, nel quale consuma l’ingresso B, effettuando delle operazioni simili a quelle che ha fatto nello stato in1.
- Avendo ricevuto entrambe le stringhe, è ora di far funzionare il meccanismo di feedback dell’unità operativa, entrando nello stato **op**; in questo stato, in corrispondenza di ogni colpo di clock bisogna controllare il valore di underflow, ovvero il segnale di stato restituito dall’unità operativa: se il segnale è basso, allora si procede con l’abilitazione del registro, altrimenti ci si predisponde per concludere l’operazione. Visto che in precedenza abbiamo detto che il registro dev’essere edge triggered, è necessario fare in modo che il segnale di abilitazione si alzi e si abbassi, e questa cosa può ad esempio essere gestita suddividendo lo stato op in due stati, op1 e op2: soltanto in uno dei due si alza l’abilitazione, nell’altro si abbassa.
- L’ultimo stato prima di tornare in idle è **end_op**, in cui l’unità di controllo avvia l’handshake verso il client per segnalargli che l’operazione è conclusa.

In **logica microprogrammata** l’idea è di salvare, per ogni stato, le uscite e lo stato prossimo in una ROM, e di implementare lo stato corrente mediante un Program Counter (PC). Il PC funge da indirizzo per la ROM, selezionando una specifica control word. Nella control word ci sono tre tipi di segnali: il PC_Next, i segnali di uscita e i segnali di controllo interno. Il PC_Next è il prossimo valore del PC nel caso di “evoluzione libera” del microprogramma, e va in ingresso ad una logica combinatoria che seleziona il valore da scrivere nel PC in funzione del PC_Next, di segnali di ingresso esterni e dei segnali di controllo interno. I segnali di uscita, come si può immaginare, sono i comandi che vengono inviati all’unità operativa. I segnali di controllo interno specificano se la logica combinatoria che seleziona il valore da scrivere nel PC debba prendere in considerazione dei segnali di ingresso esterni. In generale, in una ROM si potrebbe implementare anche il protocollo di comunicazione; tuttavia, fare ciò porterebbe ad un numero di control word piuttosto elevato, e a molta complessità nella gestione dei segnali di ingresso esterni a causa della “variabilità” della sequenza di stati. Quindi conviene separare la ROM dall’automa che implementa gli handshake, riducendo significativamente sia la complessità del microprogramma sia la complessità dell’automa, il quale non deve più comandare l’unità operativa, ma stimolare la parte di logica microprogrammata. Come segnale di controllo interno, la ROM usa sicuramente un segnale che indica che bisogna controllare

la condizione di underflow, e lo fa in corrispondenza di quello che è lo stato op, dal punto di vista logico. L'architettura che si ottiene è simile alla seguente:



5.3 Codice

5.3.1 Logica Cablata

Riportiamo dunque il codice relativo alla logica cablata, seguendo un approccio top-down. Il top module:

```

entity a_mod_b is
  Port ( clk : in STD_LOGIC;
         start_a : in STD_LOGIC;
         start_b : in STD_LOGIC;
         ack : in STD_LOGIC;
         rst : in STD_LOGIC;
         x : in STD_LOGIC_VECTOR(7 downto 0);
         received_a : out STD_LOGIC;
         received_b : out STD_LOGIC;
         end_operation : out STD_LOGIC;
         y : out STD_LOGIC_VECTOR(7 downto 0));
end a_mod_b;

architecture Structural of a_mod_b is
signal cout : std_logic := '0'; signal sel_a : std_logic := '0';
signal sel_b : std_logic := '0'; signal en_a : std_logic := '0';
signal en_b : std_logic := '0';

component cntrl_unit is
  Port ( clk : in STD_LOGIC;
         start_a : in STD_LOGIC;
         start_b : in STD_LOGIC;
         cout : in STD_LOGIC;
         ack : in STD_LOGIC;
         rst : in STD_LOGIC;
         sel_a : out STD_LOGIC);
end component;

```

```

        sel_b : out STD_LOGIC;
        en_a : out STD_LOGIC;
        en_b : out STD_LOGIC;
        received_a : out STD_LOGIC;
        received_b : out STD_LOGIC;
        end_operation : out STD_LOGIC);
end component;

component op_unit is
    Port ( x : in STD_LOGIC_VECTOR(7 downto 0);
           sel_a : in STD_LOGIC;
           sel_b : in STD_LOGIC;
           en_a : in STD_LOGIC;
           en_b : in STD_LOGIC;
           rst : in STD_LOGIC;
           cout : out STD_LOGIC;
           y : out STD_LOGIC_VECTOR(7 downto 0));
end component;

begin
cntrl: cntrl_unit port map(
    clk => clk, start_a => start_a, start_b => start_b, cout => cout,
    sel_a => sel_a, sel_b => sel_b, en_a => en_a, en_b => en_b,
    received_a => received_a, received_b => received_b,
    ack => ack, end_operation => end_operation, rst => rst
);
op : op_unit port map(
    x => x, sel_a => sel_a, sel_b => sel_b, en_a => en_a, en_b => en_b,
    cout => cout, y => y, rst => rst
);
end Structural;

```

L'unità operativa:

```

entity op_unit is
    Port ( x : in STD_LOGIC_VECTOR(7 downto 0);
           sel_a : in STD_LOGIC;
           sel_b : in STD_LOGIC;
           en_a : in STD_LOGIC;
           en_b : in STD_LOGIC;
           rst : in STD_LOGIC;
           cout : out STD_LOGIC;
           y : out STD_LOGIC_VECTOR(7 downto 0));
end op_unit;

architecture Structural of op_unit is
-- op1 ed op2 in ingresso all'adder; res in uscita
signal op1 : std_logic_vector(7 downto 0) := (others => '0');

```

```

signal op2 : std_logic_vector(7 downto 0) := (others => '0');
signal res : std_logic_vector(7 downto 0) := (others => '0');
-- u_op2 viene negato per ottenerne u2
signal u_op2 : std_logic_vector(7 downto 0) := (others => '0');

component not_N_bit is
    Generic(n : positive := 8);
    Port ( x : in STD_LOGIC_VECTOR(n-1 downto 0);
            y : out STD_LOGIC_VECTOR(n-1 downto 0));
end component;

component registro_N_bit_multiplexed is
    Generic(n : positive := 8);
    Port ( sel : in STD_LOGIC;
            en : in STD_LOGIC;
            a : in STD_LOGIC_VECTOR(n-1 downto 0);
            b : in STD_LOGIC_VECTOR(n-1 downto 0);
            rst : in STD_LOGIC;
            y : out STD_LOGIC_VECTOR(n-1 downto 0));
end component;

component adder is
    Generic(n : positive := 8);
    Port ( a : in STD_LOGIC_VECTOR(n-1 downto 0);
            b : in STD_LOGIC_VECTOR(n-1 downto 0);
            y : out STD_LOGIC_VECTOR(n-1 downto 0);
            cout : out STD_LOGIC);
end component;

begin
frontend_A : registro_N_bit_multiplexed generic map(n => 8) port map(
    sel => sel_a, en => en_a, a => res, b => x, y => op1, rst => rst
);

frontend_B : registro_N_bit_multiplexed generic map(n => 8) port map(
    sel => sel_b, en => en_b, a => "00000000", b => x, y => u_op2, rst => rst
);

negate : not_N_bit generic map(n => 8) port map(
    x => u_op2, y => op2
);

compare : adder generic map(n => 8) port map(
    a => op1, b => op2, y => res, cout => cout
);

y <= res;

end Structural;

```

L'unità di controllo:

```
entity cntrl_unit is
  Port ( clk : in STD_LOGIC;
         start_a : in STD_LOGIC;
         start_b : in STD_LOGIC;
         cout : in STD_LOGIC;
         ack : in STD_LOGIC;
         rst : in STD_LOGIC;
         sel_a : out STD_LOGIC;
         sel_b : out STD_LOGIC;
         en_a : out STD_LOGIC;
         en_b : out STD_LOGIC;
         received_a : out STD_LOGIC;
         received_b : out STD_LOGIC;
         end_operation : out STD_LOGIC);
end cntrl_unit;

architecture Cablata of cntrl_unit is
type stato_enum is (in1, in2, op1, op2, idle, end_op);
signal current_state : stato_enum;

begin

cntrl_func : process(clk, rst)
begin
  if (rst = '1') then
    current_state <= idle; sel_a <= '0'; sel_b <= '0'; en_a <= '0';
    en_b <= '0'; received_a <= '0'; received_b <= '0'; end_operation
    <= '0';
  elsif (clk'event and clk = '1') then
    case current_state is
      when idle =>
        if (start_a = '1') then
          current_state <= in1; sel_a <= '0'; sel_b <= '0';
          en_a <= '0';
          en_b <= '0'; received_a <= '0'; received_b <= '0';
          end_operation <= '0';
        else
          current_state <= idle; sel_a <= '0'; sel_b <= '0';
          en_a <= '0';
          en_b <= '0'; received_a <= '0'; received_b <= '0';
          end_operation <= '0';
        end if;
      when in1 =>
        if (start_a = '1') then
          current_state <= in1; sel_a <= '1'; sel_b <= '0';
          en_a <= '0';
        end if;
      when op1 =>
        if (start_a = '1') then
          current_state <= op1; sel_a <= '0'; sel_b <= '0';
          en_a <= '0';
          en_b <= '0'; received_a <= '0'; received_b <= '0';
          end_operation <= '0';
        end if;
      when op2 =>
        if (start_b = '1') then
          current_state <= op2; sel_a <= '0'; sel_b <= '0';
          en_a <= '0';
          en_b <= '0'; received_a <= '0'; received_b <= '0';
          end_operation <= '0';
        end if;
      when idle =>
        if (rst = '1') then
          current_state <= idle; sel_a <= '0'; sel_b <= '0';
          en_a <= '0';
          en_b <= '0'; received_a <= '0'; received_b <= '0';
          end_operation <= '0';
        end if;
      when end_op =>
        if (end_operation = '1') then
          current_state <= end_op; sel_a <= '0'; sel_b <= '0';
          en_a <= '0';
          en_b <= '0'; received_a <= '0'; received_b <= '0';
          end_operation <= '0';
        end if;
    end case;
  end if;
end process;
end Cablata;
```

```

            en_b <= '0'; received_a <= '1'; received_b <= '0';
end_operation <= '0';
            elsif (start_b = '1') then
                current_state <= in2; sel_a <= '1'; sel_b <= '0';
en_a <= '0';
            en_b <= '0'; received_a <= '0'; received_b <= '0';
end_operation <= '0';
            else
                current_state <= in1; sel_a <= '1'; sel_b <= '0';
en_a <= '1';
                en_b <= '0'; received_a <= '0'; received_b <= '0';
end_operation <= '0';
            end if;
when in2 =>
            if (start_b = '1') then
                current_state <= in2; sel_a <= '0'; sel_b <= '1';
en_a <= '0';
                en_b <= '0'; received_a <= '0'; received_b <= '1';
end_operation <= '0';
            else
                current_state <= op1; sel_a <= '0'; sel_b <= '0';
en_a <= '0';
                en_b <= '1'; received_a <= '0'; received_b <= '0';
end_operation <= '0';
            end if;
when op1 =>
            if (cout = '0') then
                current_state <= op2; sel_a <= '0'; sel_b <= '0';
en_a <= '1';
                en_b <= '0'; received_a <= '0'; received_b <= '0';
end_operation <= '0';
            else
                current_state <= end_op; sel_a <= '0'; sel_b <= '0';
en_a <= '0';
                en_b <= '1'; received_a <= '0'; received_b <= '0';
end_operation <= '0';
            end if;
when op2 =>
            if (cout = '0') then
                current_state <= op1; sel_a <= '0'; sel_b <= '0';
en_a <= '0';
                en_b <= '0'; received_a <= '0'; received_b <= '0';
end_operation <= '0';
            else
                current_state <= end_op; sel_a <= '0'; sel_b <= '0';
en_a <= '0';
                en_b <= '1'; received_a <= '0'; received_b <= '0';
end_operation <= '0';
            end if;
when end_op =>

```

```

        if (ack = '0') then
            current_state <= end_op; sel_a <= '0'; sel_b <= '0';
en_a <= '0';
            en_b <= '0'; received_a <= '0'; received_b <= '0';
end_operation <= '1';
        else
            current_state <= idle; sel_a <= '0'; sel_b <= '0';
en_a <= '0';
            en_b <= '0'; received_a <= '0'; received_b <= '0';
end_operation <= '0';
        end if;
    when others =>
        current_state <= idle; sel_a <= '0'; sel_b <= '0'; en_a
<= '0';
        en_b <= '0'; received_a <= '0'; received_b <= '0';
end_operation <= '0';
    end case;
end if;
end process;

end Cablata;

```

I componenti dell'unità operativa:

```

entity not_N_bit is
    Generic(n : positive := 8);
    Port ( x : in STD_LOGIC_VECTOR(n-1 downto 0);
            y : out STD_LOGIC_VECTOR(n-1 downto 0));
end not_N_bit;

architecture Behavioral of not_N_bit is

begin
y <= std_logic_vector(to_signed(- to_integer(signed(x)), n));

end Behavioral;

entity registro_N_bit_multiplexed is
    Generic(n : positive := 8);
    Port ( sel : in STD_LOGIC;
            en : in STD_LOGIC;
            a : in STD_LOGIC_VECTOR(n-1 downto 0);
            b : in STD_LOGIC_VECTOR(n-1 downto 0);
            rst : in STD_LOGIC;
            y : out STD_LOGIC_VECTOR(n-1 downto 0));
end registro_N_bit_multiplexed;

architecture Behavioral of registro_N_bit_multiplexed is
signal u : std_logic_vector(n-1 downto 0) := (others => '0');

```

```

component mux_2_1_parallel_N is
    Generic(n : positive := 8);
    Port ( a : in STD_LOGIC_VECTOR(n-1 downto 0);
            b : in STD_LOGIC_VECTOR(n-1 downto 0);
            sel : in STD_LOGIC;
            y : out STD_LOGIC_VECTOR(n-1 downto 0));
end component;

component registro_N_bit is
    Generic(n : positive := 8);
    Port ( par : in STD_LOGIC_VECTOR(n-1 downto 0);
            en : in STD_LOGIC;
            rst : in STD_LOGIC;
            y : out STD_LOGIC_VECTOR(n-1 downto 0));
end component;

begin
mux : mux_2_1_parallel_N generic map(n => n) port map(
    a => a, b => b, sel => sel, y => u
);

registro : registro_N_bit generic map(n => n) port map(
    par => u, en => en, y => y, rst => rst
);

end Behavioral;

entity adder is
    Generic(n : positive := 8);
    Port ( a : in STD_LOGIC_VECTOR(n-1 downto 0);
            b : in STD_LOGIC_VECTOR(n-1 downto 0);
            y : out STD_LOGIC_VECTOR(n-1 downto 0);
            cout : out STD_LOGIC);
end adder;

architecture Behavioral of adder is

begin
y <= std_logic_vector(to_signed(to_integer(signed(a)) +
to_integer(signed(b)), n));

carry : process(a, b)
begin
    if ((to_integer(signed(a)) + to_integer(signed(b))) < 0) then
        cout <= '1';
    else
        cout <= '0';
    end if;
end process;

```

```
end Behavioral;
```

I sotto-componenti del registro a N bit multiplexed:

```
entity mux_2_1_parallel_N is
  Generic(n : positive := 8);
  Port ( a : in STD_LOGIC_VECTOR(n-1 downto 0);
          b : in STD_LOGIC_VECTOR(n-1 downto 0);
          sel : in STD_LOGIC;
          y : out STD_LOGIC_VECTOR(n-1 downto 0));
end mux_2_1_parallel_N;

architecture Structural of mux_2_1_parallel_N is

component mux_2_1 is
  Port ( a : in STD_LOGIC;
          b : in STD_LOGIC;
          sel : in STD_LOGIC;
          y : out STD_LOGIC);
end component;

begin
banco_mux : for i in 0 to n-1 generate
  mux_i : mux_2_1 port map(
    a => a(i), b => b(i), sel => sel, y => y(i)
  );
end generate;

end Structural;

entity registro_N_bit is
  Generic(n : positive := 8);
  Port ( par : in STD_LOGIC_VECTOR(n-1 downto 0);
          en : in STD_LOGIC;
          rst : in STD_LOGIC;
          y : out STD_LOGIC_VECTOR(n-1 downto 0));
end registro_N_bit;

architecture Structural of registro_N_bit is

component ff_D is
  Port ( d : in STD_LOGIC;
          en : in STD_LOGIC;
          rst : in STD_LOGIC;
          q : out STD_LOGIC);
end component;

begin
banco_ff : for i in 0 to n-1 generate
  ff_i : ff_D port map(

```

```

        d => par(i), en => en, q => y(i), rst => rst
    );
end generate;

end Structural;

```

Essi a loro volta sono strutturali (se si legge Behavioral dove dovrebbe essere Structural non è un errore, è che l'IDE di default mette Behavioral, ma è irrilevante ai fini del risultato):

```

entity mux_2_1 is
  Port ( a : in  STD_LOGIC;
         b : in  STD_LOGIC;
         sel : in  STD_LOGIC;
         y : out  STD_LOGIC);
end mux_2_1;

architecture Dataflow of mux_2_1 is

begin
y <= (a and (not sel)) or (b and sel);

end Dataflow;

entity ff_D is
  Port ( d : in  STD_LOGIC;
         en : in  STD_LOGIC;
         rst : in  STD_LOGIC;
         q : out  STD_LOGIC);
end ff_D;

architecture Behavioral of ff_D is

begin
ff : process(en)
begin
  if (rst = '1') then
    q <= '0';
  elsif (en = '1' and en'event) then
    q <= d;
  end if;
end process;

end Behavioral;

```

5.3.2 Logica Microprogrammata

Di seguito è riportato il codice impiegato per l'implementazione della logica microprogrammata. Il primo codice che vediamo è l'interfaccia dell'intero sistema.

```

entity system is
  port (
    buff_in : in std_logic_vector ( 7 downto 0 );
    buff_out : out std_logic_vector ( 7 downto 0 );
    clk : in std_logic;
    start : in std_logic;
    syn_out : out std_logic;
    ack_out : out std_logic;
    syn_in : in std_logic;
    ack_in : in std_logic;
    EOOP : out std_logic
  );
end system;

architecture structural of system is
  component FF_D is
    port (
      clk, en, reset : in std_logic;
      D : in std_logic_vector ( 7 downto 0 );
      Q : out std_logic_vector ( 7 downto 0 )
    );
  end component;

  component U_0 is
    port (
      busin : in std_logic_vector ( 7 downto 0 );
      clk, enA, enB, selB, selA, start : in std_logic;
      busout : out std_logic_vector ( 7 downto 0 );
      cout : out std_logic
    );
  end component;

  component A_B is
    port (
      clk : in std_logic;
      buff_out_in : in std_logic_vector ( 7 downto 0 );
      buff_in_out : in std_logic_vector ( 7 downto 0 );
      to_in : out std_logic_vector ( 7 downto 0 );
      to_out : out std_logic_vector ( 7 downto 0 );
      start : in std_logic;
      syn_out : out std_logic;
      ack_in : in std_logic;
      syn_in : in std_logic;
      ack_out : out std_logic;
      EOOP_in : in std_logic;
      EOOP_out : out std_logic;
      ready : out std_logic);
  end component;

```

```

end component;

component U_C is
  Port (
    ready : in std_logic;
    clk : in std_logic;
    start : in std_logic;
    cout : in std_logic;
    selA : out std_logic;
    enA : out std_logic;
    selB : out std_logic;
    enB : out std_logic;
    EOOP : out std_logic
  );
end component;

signal buff : std_logic_vector ( 7 downto 0 ) := (others => '0');
signal buffout : std_logic_vector ( 7 downto 0 ) := (others => '0');
signal ready : std_logic := '0';
signal cout : std_logic := '0';
signal selA : std_logic := '0';
signal enA : std_logic := '0';
signal selB : std_logic := '0';
signal enB : std_logic := '0';
signal enRC : std_logic := '0';
signal EOOP_internal : std_logic := '0';
signal result : std_logic_vector ( 7 downto 0 ) := (others => '0');

begin

U_O_S_2 : U_O
  port map (
    busin => buff,
    clk => clk,
    enA => enA,
    enB => enB,
    selB => selB,
    selA => selA,
    start => start,
    busout => result,
    cout => cout
  );

U_C_S_2 : U_C
  port map (

```

```

    ready => ready,
    clk => clk,
    start => start,
    cout => cout,
    selA => selA,
    enA => enA,
    selB => selB,
    enB => enB,
    EOOP => EOOP_internal

);

A_B_S_2 : A_B
port map (
    clk => clk,
    buff_out_in => buff_in,
    buff_in_out => result,
    to_in => buff,
    to_out => buff_out,
    start => start,
    ack_in => ack_in,
    syn_in => syn_in,
    ack_out => ack_out,
    syn_out => syn_out,
    EOOP_in => EOOP_internal,
    EOOP_out => EOOP,
    ready => ready
);

end structural;

```

Di seguito è riportato il codice del proxy implementato con approccio behavioral.

```

entity A_B is
    port (
        clk : in std_logic;
        buff_out_in : in std_logic_vector (7 downto 0);
        buff_in_out : in std_logic_vector (7 downto 0);
        to_in : out std_logic_vector (7 downto 0);
        to_out : out std_logic_vector (7 downto 0);
        start : in std_logic;
        syn_out : out std_logic;
        ack_in : in std_logic;
        syn_in : in std_logic;
        ack_out : out std_logic;
        EOOP_in : in std_logic;
        EOOP_out : out std_logic;
        ready : out std_logic);

```

```

end A_B;

architecture Behavioral of A_B is

signal a, b : std_logic_vector ( 7 downto 0 ) := (others => '0');
type state is ( idle,waitEvent, pollingA, waitAB, pollingB, Ap, Bp,
end_op_in, end_op_out);
signal stato, prossimo : state := idle;

begin
process (clk, start)
begin
    if ( start = '1' ) then
        to_in <= "00000000";
        to_out <= "00000000";
        ready <= '0';
        prossimo <= waitEvent;
    else
        case stato is
            when idle =>
                if (clk'event and clk = '1') then
                    prossimo <= idle;
                end if;
            when waitEvent =>
                if (clk'event and clk = '1') then
                    if syn_in = '1' then
                        prossimo <= pollingA;
                    else
                        prossimo <= waitEvent;
                    end if;
                end if;
            when pollingA =>
                if (clk'event and clk = '1') then
                    if ack_in = '1' then
                        a <= buff_out_in;
                        syn_out <= '0';
                        ack_out <= '0';
                    end if;
                end if;
        end case;
    end if;
end process;
end Behavioral;

```

```

        prossimo <= waitAB;

    else

        syn_out <= '1';
        ack_out <= '1';
        prossimo <= pollingA;

    end if;
end if;

when waitAB =>

    if (clk'event and clk = '1') then

        if syn_in = '1' then

            prossimo <= pollingB;
        else
            prossimo <= waitAB;

        end if;
    end if;

when pollingB =>

    if (clk'event and clk = '1') then

        if ack_in = '1' then

            b <= buff_out_in;
            ready <= '1';
            syn_out <= '0';
            ack_out <= '0';
            prossimo <= Ap;

        else

            syn_out <= '1';
            ack_out <= '1';
            prossimo <= pollingB;

        end if;
    end if;

when Ap =>

    if (clk'event and clk = '1') then

        prossimo <= Bp;

```

```

        to_in <= a;

    end if;

when Bp =>

    if (clk'event and clk = '1') then
        prossimo <= end_op_in;
        to_in <= b;
    end if;

when end_op_in =>

    if (clk'event and clk = '1') then

        if EOOP_in = '1' then

            ready <= '0';
            prossimo <= end_op_out;

        else

            syn_out <= '0';
            prossimo <= end_op_in;
        end if;
    end if;

when end_op_out =>

    if (clk'event and clk = '1') then

        if ack_in = '1' and syn_in = '1' then

            syn_out <= '0';
            ack_out <= '1';
            EOOP_out <= '1';
            prossimo <= idle;

        else

            ack_out <= '0';
            syn_out <= '1';
            EOOP_out <= '0';
            prossimo <= end_op_out;
            to_out <= buff_in_out;

        end if;
    end if;
end case;
end if;

```

```

    end process;

    stato <= prossimo;
end Behavioral;
```

Di seguito è riportato il codice dell'unità operativa, realizzata con un approccio di tipo strutturale.

```

entity U_0 is
    port (
        busin : in std_logic_vector ( 7 downto 0 );
        clk, enA, enB, selB, selA, start : in std_logic;
        busout : out std_logic_vector ( 7 downto 0 );
        cout : out std_logic
    );
end U_0;

architecture structural of U_0 is

component Ripple_Carry is
    port ( x : in std_logic_vector (7 downto 0);
           y : in std_logic_vector (7 downto 0);
           Cin : in std_logic;
           Sum : out std_logic_vector (7 downto 0);
           Cout : out std_logic);
end component;

begin
    component FF_D is
        port (
            clk, en, reset : in std_logic;
            D : in std_logic_vector ( 7 downto 0 );
            Q : out std_logic_vector ( 7 downto 0 )
        );
    end component;

    component mux2_1 is
        port ( I0 : in std_logic_vector ( 7 downto 0 );
               I1 : in std_logic_vector ( 7 downto 0 );
               S : in std_logic;
               U : out std_logic_vector ( 7 downto 0 )
        );
    end component;

    signal a_mux : std_logic_vector ( 7 downto 0 );
```

```

signal b_mux : std_logic_vector ( 7 downto 0 );
signal a_ff : std_logic_vector ( 7 downto 0 );
signal b_ff : std_logic_vector (7 downto 0);
signal a_feedback : std_logic_vector (7 downto 0);
signal Cout_interno : std_logic := '0';

begin

muxa : mux2_1
port map (
    I0 => a_feedback,
    I1 => busin,
    S => selA,
    U => a_mux
);

muxb : mux2_1
port map (
    I0 => "00000000",
    I1 => busin,
    S => selB,
    U => b_mux
);

ff_a : FF_D
port map (
    clk => clk,
    en => (enA and Cout_interno),
    reset => start,
    D => a_mux,
    Q => a_ff
);

ff_b : FF_D
port map (
    clk => clk,
    en => enb,
    reset => start,
    D => b_mux,
    Q => b_ff
);

sottrattore : Ripple_Carry
port map (
    x => a_ff,
    y => not(b_ff),
    cin => '1',
    Sum => a_feedback,
    cout => Cout_interno
);

```

```

cout <= Cout_interno;

busout <= a_feedback;

end structural;

```

Adesso troviamo il codice delle componenti dell'unità operativa. Troviamo il multiplexer implementato con approccio di tipo dataflow, mentre il registro ed il sottrattore sono ottenuti con approccio di tipo comportamentale.

```

entity mux2_1 is

    port ( I0 : in std_logic_vector ( 7 downto 0 );
           I1 : in std_logic_vector ( 7 downto 0 );
           S : in std_logic;
           U : out std_logic_vector ( 7 downto 0 )
         );
end mux2_1;

architecture df of mux2_1 is

begin
    with S select
        U <= I0 when '0',
                  I1 when '1',
                  (others => '-') when others;
end df;

entity FF_D is

port (
    clk, en, reset : in std_logic;
    D : in std_logic_vector ( 7 downto 0 );
    Q : out std_logic_vector ( 7 downto 0 )
  );
end FF_D;

architecture Behavioral of FF_D is

begin

process (clk)
begin
    if (clk'event and clk = '1') then

```

```

        if reset = '1' then
            Q <= "00000000";
        elsif en = '1' then
            Q <= D;
        end if;
    end if;
end process;

end Behavioral;

entity Ripple_Carry is

port ( x : in std_logic_vector (7 downto 0);
       y : in std_logic_vector (7 downto 0);
       Cin : in std_logic;
       Sum : out std_logic_vector (7 downto 0);
       Cout : out std_logic);

end Ripple_Carry;

architecture Behavioral of Ripple_Carry is

signal extended_x : std_logic_vector (8 downto 0) := (others => '0');
signal extended_y : std_logic_vector (8 downto 0) := (others => '0');
signal extended_cin : std_logic_vector (8 downto 0) := (others => '0');
signal extended_sum : std_logic_vector (8 downto 0) := (others => '0');

begin

extended_x <='0'&x;
extended_y <='0'&y;
extended_cin <= "00000000"&Cin;
extended_sum <= std_logic_vector (unsigned(extended_x) +
unsigned(extended_y) + unsigned(extended_cin));
Cout <= extended_sum(8);
Sum <= extended_sum(7 others 0);

end Behavioral;

```

Di seguito troviamo il codice dell'unità di controllo realizzato con un approccio ibrido sfruttando il componente ROM in modo strutturale ed il processo di controllo ottenuto con approccio di tipo comportamentale.

```

entity U_C is
    Port (
        ready : in std_logic;
        clk : in std_logic;

```

```

    start : in std_logic;
    cout : in std_logic;
    selA : out std_logic;
    enA : out std_logic;
    selB : out std_logic;
    enB : out std_logic;
    EOOP : out std_logic
);
end U_C;

architecture hybrid of U_C is
component ROM is
  Port (
    PC : in unsigned (2 downto 0);
    PC_next : out unsigned (2 downto 0);
    jcout : out std_logic;
    selA : out std_logic;
    enA : out std_logic;
    selB : out std_logic;
    enB : out std_logic;
    EOOP : out std_logic
  );
end component;

signal jcout : std_logic;
signal PC, PC_next : unsigned (2 downto 0):= "000";
begin

MicroRom : ROM
  port map (
    PC => PC,
    PC_next => PC_next,
    jcout => jcout,
    selA => selA,
    enA => enA,
    selB => selB,
    enB => enB,
    EOOP => EOOP
  );

control : process (clk)
  begin

    if (clk'event and clk = '1') then

      if (ready = '0') then
        PC <= (others => '0');
      elsif (jcout = '1' and cout = '0') then
        PC <= "101";
      end if;
    end if;
  end process;
end;

```

```

        else
            PC <= PC_next;
        end if;
    end if;

end process;

end hybrid;

entity ROM is
    Port (
        PC : in unsigned (2 downto 0);
        PC_next : out unsigned (2 downto 0);
        jcout : out std_logic;
        selA : out std_logic;
        enA : out std_logic;
        selB : out std_logic;
        enB : out std_logic;
        EOOP : out std_logic
    );
end ROM;

architecture Behavioral of ROM is

type control_record is record
    PC_next : unsigned (2 downto 0);
    jcout : std_logic;
    selA : std_logic;
    enA : std_logic;
    selB : std_logic;
    enB : std_logic;
    EOOP : std_logic;
end record;

constant idle : control_record := (
    PC_next => "001",
    jcout => '0',
    selA => '0',
    enA => '0',
    selB => '0',
    enB => '1',
    EOOP => '0'
);

constant getA : control_record := (
    PC_next => "010",
    jcout => '0',

```

```

        selA => '1',
        enA => '1',
        selB => '0',
        enB => '0',
        EOOP => '0'
    );
constant prepareB : control_record := (
    PC_next => "011",
    jcout => '0',
    selA => '0',
    enA => '0',
    selB => '1',
    enB => '0',
    EOOP => '0'
);
constant getB : control_record := (
    PC_next => "100",
    jcout => '0',
    selA => '0',
    enA => '0',
    selB => '1',
    enB => '1',
    EOOP => '0'
);
constant op : control_record := (
    PC_next => "100",
    jcout => '1',
    selA => '0',
    enA => '1',
    selB => '0',
    enB => '0',
    EOOP => '0'
);
constant end_op : control_record := (
    PC_next => "000",
    jcout => '0',
    selA => '0',
    enA => '0',
    selB => '0',
    enB => '1',
    EOOP => '1'
);
type ROM_TYPE is array ( 0 to 5 ) of control_record;
constant control_store : ROM_TYPE := (
    0 => idle,
    1 => getA,

```

```

        2 => prepareB,
        3 => getB,
        4 => op,
        5 => end_op
    );
signal controllo : control_record;

begin

controllo <= control_store(to_integer(PC));

PC_next <= controllo.PC_next;
jcout <= controllo.jcout;
selA <= controllo.selA;
enA <= controllo.enA;
selB <= controllo.selB;
enB <= controllo.enB;
EOOP <= controllo.EOOP;

End Behavioral;

```

5.4 Simulazione

5.4.1 Simulazione logica cablata

Poiché il sistema è composto da parecchi moduli, sono stati creati vari test d'unità per verificare il corretto funzionamento dei singoli moduli, partendo da quelli base a salire nella gerarchia dei moduli. E' stato utile per risolvere dei problemi di temporizzazione riscontrati nel protocollo, e per decidere di rendere comunque sincrono un automa che esegue un protocollo asincrono: può sembrare ovvio che l'automa nelle sue transizioni deve comunque lavorare in maniera sincrona ad un clock, ma non lo è. Riportiamo quindi il testbench impiegato per la simulazione della logica cablata.

```

ENTITY a_mod_b_testbench IS
END a_mod_b_testbench;

ARCHITECTURE behavior OF a_mod_b_testbench IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT a_mod_b
PORT(
    clk : in STD_LOGIC;
    start_a : in STD_LOGIC;
    start_b : in STD_LOGIC;

```

```

        ack : in STD_LOGIC;
        rst : in STD_LOGIC;
        x : in STD_LOGIC_VECTOR(7 downto 0);
        received_a : out STD_LOGIC;
        received_b : out STD_LOGIC;
        end_operation : out STD_LOGIC;
        y : out STD_LOGIC_VECTOR(7 downto 0)
    );
END COMPONENT;

--Inputs
signal clk : std_logic := '0';
signal start_a : std_logic := '0';
signal start_b : std_logic := '0';
signal x : std_logic_vector(7 downto 0) := (others => '0');
signal ack : std_logic := '0';
signal rst : std_logic := '0';

--Outputs
signal received_a : std_logic := '0';
signal received_b : std_logic := '0';
signal end_operation : std_logic := '0';
signal y : std_logic_vector(7 downto 0) := (others => '0');

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

```

BEGIN

```

    -- Instantiate the Unit Under Test (UUT)
    uut: a_mod_b PORT MAP (
        clk => clk,
        start_a => start_a,
        start_b => start_b,
        x => x,
        received_a => received_a,
        received_b => received_b,
        y => y,
        ack => ack,
        end_operation => end_operation,
        rst => rst
    );

    -- Clock process definitions
    clk_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';

```

```

        wait for clk_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
    wait for clk_period*10;
    rst <= '1';
    wait for clk_period;
    rst <= '0';
    wait for clk_period;
    -- protocollo asincrono: non uso il clock ma gli eventi
    -- è stato comunque necessario fare la control unit sincrona perché
next_state
    -- poteva essere schedulato più volte ed i segnali di controllo
erano troppo
        -- brevi in durata
        x <= "01111111"; -- A=127
        wait for clk_period;
        start_a <= '1';
        wait until (received_a = '1');
        start_a <= '0';
        wait until (received_a = '0');

        x <= "00000100"; -- B=4 -> mi aspetto poi y=3 come resto modulo
        wait for clk_period;
        start_b <= '1';
        wait until (received_b = '1');
        start_b <= '0';
        wait until (received_b = '0');

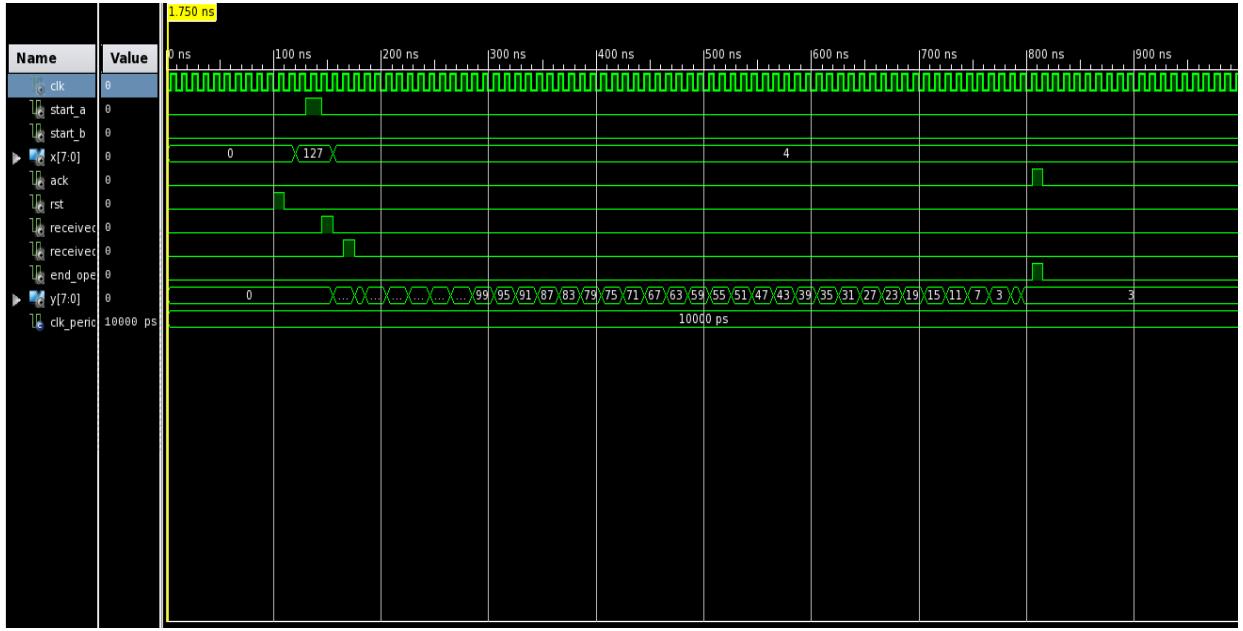
        wait until (end_operation = '1');
        ack <= '1';
        wait until (end_operation = '0');
        ack <= '0';
        assert y = "00000011";

        wait;
end process;

END;

```

I risultati della simulazione sono quelli previsti, e sono stati resi più semplici da visualizzare utilizzando un radix unsigned decimal per il vettore di bit in ingresso e per quello in uscita:



5.4.2 Simulazione logica microprogrammata

Di seguito troviamo il codice impiegato per la simulazione della logica micropogrammata.

```
ENTITY system_tb IS
END system_tb;

architecture behavior OFsystem_tb IS

COMPONENT system
port (
    buff_in : in std_logic_vector ( 7 downto 0 );
    buff_out : out std_logic_vector ( 7 downto 0 );
    clk : in std_logic;
    start : in std_logic;
    syn_out : out std_logic;
    ack_out : out std_logic;
    syn_in : in std_logic;
    ack_in : in std_logic;
    EOOP : out std_logic
);
end COMPONENT;

signal buff_in : std_logic_vector ( 7 downto 0 ) := (others => '0');
signal clk : std_logic := '0';
signal start : std logic := '0';
```

```

signal ack_in : std_logic := '0';
signal syn_in : std_logic := '0';

signal ack_out : std_logic;
signal buff_out : std_logic_vector ( 7 downto 0 );
signal syn_out : std_logic;
signal EOOP : std_logic;

constant clk_period : time := 10 ns;

begin

uut: system port map(
    buff_in => buff_in,
    buff_out => buff_out,
    clk => clk,
    start => start,
    ack_in => ack_in,
    syn_in => syn_in,
    ack_out => ack_out,
    syn_out => syn_out,
    EOOP => EOOP
);

clk_process : process
begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
end process;

stim_proc: process
begin

    start <= '1';
    wait for 100 ns;

    start <= '0';

    wait for clk_period;

    syn_in <= '1';
    buff_in <= "00000111";
    wait until ack_out = '1';
    syn_in <= '0';
    ack_in <= '1';
    wait until ack_out = '0';

```

```
ack_in <= '0';
syn_in <= '1';
buff_in <= "00000011";
wait until ack_out = '1';
wait for 3* clk_period;
ack_in <= '1';
syn_in <= '0';
wait for clk_period;
ack_in <= '0';
wait until syn_out = '1';
syn_in <= '1';
ack_in <= '1';
wait until EOP = '1' and ack_out = '1';

wait for 2* clk_period;

wait;
end process;

END;
```

Esercizio 6

6.1 Traccia

Progettare un sistema per inviare due parole da 16 bit ciascuna da una unità A ad una unità B.

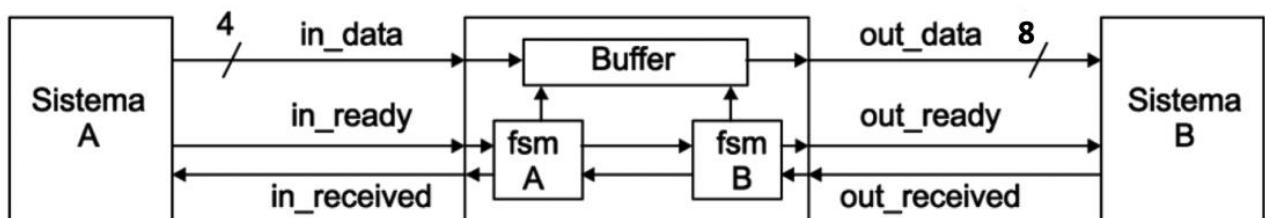
Le due unità non sono dotate di un collegamento diretto costituito da un bus di 16 bit, ma l'unità A possiede un bus di 4 bit in uscita e l'unità B possiede un bus di 8 bit in ingresso. Per questa ragione, il trasferimento deve avvenire in più passi facendo uso di un buffer.

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

6.2 Soluzione

Il sistema deve permettere di inviare due parole da 16 bit ciascuna da una unità A ad una unità B. Le due unità non hanno lo stesso parallelismo nel trasferimento di dati. Il sistema A comunica tramite un bus di 4 bit in uscita, mentre il sistema B legge da un bus di 8 bit in ingresso. A tale scopo è stato progettato un blocco di interfaccia che legge sequenzialmente 2 parole di 4 bit dal sistema A, memorizzandole in un buffer di 8 bit, e che trasferisce, successivamente al sistema B. Questa operazione viene effettuata due volte per ogni parola trasmessa, quindi verrà effettuato in totale 4 volte, dato che le parole da trasferire sono 2. Questo blocco è composto dal buffer e da due automi a stati finiti che compongono l'unità di controllo del sistema. Il primo automa, chiamato fsmA, è collegato al sistema A, mentre il secondo automa, chiamato fsmB è collegato al sistema B.

Il sistema complessivo può essere rappresentato mediante il seguente schema:



L'automa fsmA comunica con il sistema A mediante un protocollo asincrono di handshaking, i due utilizzano due coppie di segnali per gestire il protocollo. La prima coppia di segnali è: “*A_rdy*” e “*gotA_rdy*”. Il primo è inviato da fsmA al sistema A per segnalargli che il buffer è pronto a ricevere le parole che A deve inviare a B; il secondo è inviato dal sistema A in risposta ad “*A_rdy*” e indica che ha ricevuto il segnale inviatogli da fsmA. Dopo questo handshaking di partenza viene utilizzata la seconda coppia di segnali per l'handshaking relativo al trasferimento dei 4 bit della parola. La seconda coppia è: “*in_rdy*” e “*in_rcvd*”. Il primo segnale, “*in_rdy*”, è inviato dal sistema A quando è pronto a trasferire i 4 bit, successivamente si pone in attesa del secondo segnale, “*in_rcvd*”, da parte di fsmA, il quale lo invierà solo in seguito alla ricezione dei 4 bit, per indicare l'avvenuta ricezione. L'automa fsmA, in seguito ad un doppio scambio con il sistema A riempie il buffer di 8 bit che gestisce in maniera condivisa con l'automa fsmB. I due automi, fsmA e fsmB dialogano tra loro mediante i segnali “*buffer_full*” e “*buffer_picked*”. Quando il buffer si riempie, fsmA alza il segnale “*buffer_full*”, per indicare all'automa fsmB che può operare su di esso, e si mette in attesa del segnale “*buffer_picked*” che riceverà da fsmB. L'automa fsmB comunica con il sistema B, anche in tal caso

mediante un protocollo di handshaking, attraverso la coppia di segnali “*B_rdy*” e “*gotB_rdy*”. Appena riceve il segnale “*buffer_full*” dall’altra fsm, la fsmB si occupa di trasferire il contenuto del buffer al sistema B. Quindi pone il contenuto del buffer in uscita e alza il segnale “*B_rdy*” che indica al sistema B che ha inviato 8 bit della parola. Il sistema B risponderà con il segnale “*gotB_rdy*” per indicare l’avvenuta ricezione degli 8 bit.

Vale la pena di fare ulteriori considerazioni riguardo alle possibili soluzioni. In questa soluzione, quando fsmA dice ad A “*A_rdy*”, gli garantisce che A può produrre due vettori da 4 bit da mandare a B, ovvero in totale 8 bit dei 16 che deve mandare in totale; poi il buffer si riempie e quindi fsmA deve effettuare un handshaking con fsmB, che a sua volta deve poi effettuare un handshaking con B. Nel frattempo, il sistema A sarebbe pronto per inviare altri due vettori da 4, ma per farlo deve attendere che B consumi il vettore da 8 presente nel buffer: questo vuol dire che la linea da A ad fsmA in tale tempo è idle, anche se ovviamente A può anche effettuare altre operazioni mentre aspetta, visto che è fsmA a prendere l’iniziativa di comunicare ad A che il buffer è pronto a ricevere. Dal punto di vista del sistema B, fsmB gli invia una word contenenti 8 bit dei 16 totali appena tali 8 bit sono disponibili nel buffer, quindi il buffer non contiene le word per un tempo maggiore del necessario, e **privilegia B rispetto ad A**. Analizziamo altre due possibili soluzioni. Entrambe consistono nell’avere un buffer da 16 bit. In entrambi i casi, A invia le 4 word da 4 bit o mediante 4 handshake asincroni consecutivi o mediante un solo handshake dopo il quale invia le word in maniera sincrona (come nella soluzione precedente). Il vantaggio è che A non deve aspettare che B consumi per continuare a produrre, quindi il sistema A è disaccoppiato dal sistema B. Ora arriva il momento di distinguere i due casi. Nel caso più semplice, fsmA riceve tutte e 4 le word e poi comunica con fsmB che il buffer è pieno, quindi poi fsmB avvia l’handshaking con il sistema B e gli invia due word da 8 bit, in un modo che dipende dal modello di comunicazione (quindi o due handshake asincroni, o un solo handshake dopo il quale si mandano consecutiveamente le word). In tal caso, B deve aspettare che A produca tutte le word anche se nel buffer c’è già informazione utile da inviare, dunque in sostanza B deve attendere A e la linea da fsmB a B è idle per un certo tempo nel quale potrebbe essere utilizzata. Come prima, vale il discorso che è fsmB a prendere l’iniziativa, quindi B nel frattempo può fare altro. Comunque, il buffer mantiene delle informazioni che potrebbe già inviare senza inviarle subito, quindi tale implementazione **privilegia A rispetto a B**. Nel caso più complesso, si prendono i vantaggi di entrambe le implementazioni viste prima, ovvero A invia ad fsmA quattro word da 4 bit senza attendere che B consumi, ed fsmB comincia ad inviare word da 8 bit a B appena ce le ha disponibili quindi senza attendere che vengano riempiti tutti i 16 bit. Questa sarebbe un’implementazione **fair** e molto efficiente, e dal punto di vista dei client A e B il protocollo non cambia, perché i dati possono essere inviati in qualsiasi momento. Tuttavia, il protocollo si complica nella comunicazione tra fsmA ed fsmB, ovvero non bastano “*buffer_full*” e “*buffer_picked*” ma ci vogliono dei segnali aggiuntivi del tipo “*buffer_half_full*” oppure dei contatori mod 2; inoltre, la comunicazione tra fsmA ed fsmB, affinché le prestazioni non degradino, non deve inficiare sul ritmo con il quale A invia i dati, altrimenti di nuovo il sistema tende a privilegiare B e si tende a violare la fairness. Questo vuol dire che fsmA deve fare l’handshake con fsmB in modo parallelo rispetto alla ricezione dei dati da A, e dualmente fsmB dev’essere pronto a fare un nuovo handshake con fsmA mentre sta ancora inviando i dati a B. Dunque, tale soluzione è piuttosto complessa, a meno di non rilassare vincoli sulla fairness nel caso di sistemi lenti (nel senso che se la fairness viene meno perché ad esempio B è lento in ricezione, allora va bene privilegiare A). Per questo motivo, tra le tre soluzioni proposte in genere si sceglie una delle prime due, e noi abbiamo scelto la prima.

6.3 Codice

Il codice riportato è relativo al sistema complessivo, il quale include anche le due entità A e B. È stato utilizzato il costrutto `wait until` per implementare l’attesa dei segnali del protocollo e quindi non è stato utilizzato un segnale di clock in nessuno dei componenti del sistema complessivo. Un’alternativa a questa implementazione che prevede l’utilizzo del clock, anche non sincronizzato per i vari componenti, dato che il protocollo utilizzato è asincrono e non ha bisogno esplicitamente del clock, avrebbe sfruttato il clock solo per campionare i segnali del protocollo.

Di seguito è riportato il codice dei due automi a stati finiti fsmA e fsmB:

```

entity fsmA is
  Port ( in_data : in STD_LOGIC_VECTOR(03 downto 00);
         word_buffer : out STD_LOGIC_VECTOR(07 downto 00);
         A_rdy : out STD_LOGIC;
         gotA_rdy : in STD_LOGIC;
         in_rcvd : out STD_LOGIC;
         in_rdy : in STD_LOGIC;
         buffer_picked : in STD_LOGIC;
         buffer_full : out STD_LOGIC);
end fsmA;

architecture Behavioral of fsmA is
  TYPE STATE IS (FSMA_INVIO_ARDY, FSMA_ATTESA_GOTARDY,
                 FSMA_ATTESA_INRDY1,
                 FSMA_ATTESA_INRDY0,
                 FSMA_ATTESA_BUFFERPICKED);
  signal current_state : STATE := FSMA_INVIO_ARDY;

begin
  U_C: process
    VARIABLE count: integer range 0 to 3 :=0;
  begin
    case current_state is
      when FSMA_INVIO_ARDY =>
        A_rdy <= '1';in_rcvd <= '0';buffer_full <= '0';
        current_state <= FSMA_ATTESA_GOTARDY;
      when FSMA_ATTESA_GOTARDY =>
        wait until gotA_rdy = '1';
        A_rdy <= '0'; in_rcvd <= '0';buffer_full <= '0';
        current_state <= FSMA_ATTESA_INRDY1;
      when FSMA_ATTESA_INRDY1 =>
        A_rdy <= '0';buffer_full <= '0';
        if count < 2 then
          if in_rdy = '0' then
            wait until in_rdy ='1';
          end if;
        end if;
        count := count +1;
        CASE count IS
          When 0 => NULL; in_rcvd <= '0';
          When 1 => word_buffer(03 downto 00) <=
in_data;in_rcvd <= '1'; current_state <= FSMA_ATTESA_INRDY0;
          When 2 => word_buffer(07 downto 04) <=
in_data;in_rcvd <= '1'; cu
rrent_state <= FSMA_ATTESA_INRDY0;
          When 3 => count:=0; buffer_full <= '1';
        current_state <= FSMA_ATTESA_BUFFERPICKED;
        end case;
    end case;
  end process;
end Behavioral;

```

```

        when FSMA_ATTESA_INRDY0 =>
            wait until in_rdy = '0';
            in_rcvd <= '0';A_rdy <= '0';
            current_state <=FSMA_ATTESA_INRDY1;
        when FSMA_ATTESA_BUFFERPICKED =>
            wait until buffer_picked <= '1';
            buffer_full <= '0';A_rdy <= '0'; in_rcvd <= '0';
            current_state <=FSMA_ATTESA_INRDY1;
    end case;
    wait on current_state,gotA_rdy, in_rdy, buffer_picked, in_data;
end process;

end Behavioral;

```

Automa fsmB:

```

entity fsmB is
    Port ( word_buffer : in STD_LOGIC_VECTOR(07 downto 00);
           out_data : out STD_LOGIC_VECTOR(07 downto 00);
           buffer_full : in STD_LOGIC;
           buffer_picked : out STD_LOGIC;
           B_rdy : out STD_LOGIC;
           gotB_rdy : in STD_LOGIC);
end fsmB;

architecture Behavioral of fsmB is
    TYPE STATE IS (FSMB_ATTESA_BUFFERFULL1, FSMB_ATTESA_BUFFERFULL0,
FSMB_ATTESA_GOTBRDY);
    signal current_state : STATE := FSMB_ATTESA_BUFFERFULL1;

begin
    U_C: process
        begin
            case current_state is
                when FSMB_ATTESA_BUFFERFULL1 =>
                    wait until buffer_full = '1';
                    buffer_picked <= '1';B_rdy <= '0';
                    out_data <= word_buffer;
                    current_state <= FSMB_ATTESA_BUFFERFULL0;
                when FSMB_ATTESA_BUFFERFULL0 =>
                    wait until buffer_full = '0';
                    buffer_picked <= '0';
                    B_rdy <= '1';
                    current_state <= FSMB_ATTESA_GOTBRDY;
                when FSMB_ATTESA_GOTBRDY =>
                    wait until gotB_rdy ='1';
                    B_rdy <= '0';buffer_picked <= '0';
                    current_state <= FSMB_ATTESA_BUFFERFULL1;
            end case;
            wait on current_state, buffer_full, gotB_rdy, word_buffer;

```

```
    end process;
```

```
end Behavioral;
```

Di seguito è riportato il codice dei sistemi A e B:

```
entity SistemaA is
  Port ( in_A1 : in STD_LOGIC_VECTOR (15 downto 00);
         in_A2 : in STD_LOGIC_VECTOR (15 downto 00);
         in_data : out STD_LOGIC_VECTOR (03 downto 00);
         A_rdy : in STD_LOGIC;
         gotA_rdy : out STD_LOGIC;
         in_rdy : out STD_LOGIC;
         in_rcvd : in STD_LOGIC
        );
end SistemaA;

architecture Behavioral of SistemaA is
  TYPE STATE IS (A_ATTESA_ARDY1, A_ATTESA_ARDY0, A_INVIO_DATO,
A_ATTESA_INRCVD);
  signal current_state : STATE := A_ATTESA_ARDY1;
begin
  A: process
    VARIABLE count: integer range 0 to 9 :=0;
    begin
      case current_state is
        when A_ATTESA_ARDY1 =>
          wait until A_rdy = '1';
          gotA_rdy <= '1';in_rdy <= '0';
          current_state <= A_ATTESA_ARDY0;
        when A_ATTESA_ARDY0 =>
          wait until A_rdy = '0';
          gotA_rdy <= '0';in_rdy <= '0';
          current_state <= A_INVIO_DATO;
        when A_INVIO_DATO =>
          gotA_rdy <= '0';
          count := count +1;
          CASE count IS
            When 0 => NULL; in_rdy <= '0';
            When 1 => in_data <= in_A1(03 downto
00);in_rdy <= '1'; current_state <= A_ATTESA_INRCVD;
            When 2 => in_data <= in_A1(07 downto
04);in_rdy <= '1'; current_state <= A_ATTESA_INRCVD;
            When 3 => in_data <= in_A1(11 downto
08);in_rdy <= '1'; current_state <= A_ATTESA_INRCVD;
            When 4 => in_data <= in_A1(15 downto
12);in_rdy <= '1'; current_state <= A_ATTESA_INRCVD;
            When 5 => in_data <= in_A2(03 downto
00);in_rdy <= '1'; current_state <= A_ATTESA_INRCVD;
      end CASE;
    end if;
  end process;
end Behavioral;
```

```

                When  6 => in_data <= in_A2(07 downto
04);in_rdy <= '1'; current_state <= A_ATTESA_INRCVD;
                When  7 => in_data <= in_A2(11 downto
08);in_rdy <= '1'; current_state <= A_ATTESA_INRCVD;
                When  8 => in_data <= in_A2(15 downto
12);in_rdy <= '1'; current_state <= A_ATTESA_INRCVD;
                When  9 => count := 0; in_rdy <= '0';
current_state <= A_ATTESA_BRDY1;
            end case;
        when A_ATTESA_INRCVD =>
            wait until in_rcvd <= '1';
            in_rdy <= '0';
            gotA_rdy <= '0';
            current_state <= A_INVIO_DATO;
        end case;
    wait on current_state, A_rdy, in_rcvd;
end process;

end Behavioral;
```

Sistema B:

```

entity SistemaB is
    Port ( out_data : in STD_LOGIC_VECTOR (07 downto 00);
           out_B1 : out STD_LOGIC_VECTOR (15 downto 00);
           out_B2 : out STD_LOGIC_VECTOR (15 downto 00);
           B_rdy : in STD_LOGIC;
           gotB_rdy : out STD_LOGIC
    );
end SistemaB;

architecture Behavioral of SistemaB is
    TYPE STATE IS (B_ATTESA_BRDY1, B_ATTESA_BRDY0);
    signal current_state : STATE := B_ATTESA_BRDY1;
begin
B: process
    VARIABLE count: integer range 0 to 4 :=0;
    begin
        case current_state is
            when B_ATTESA_BRDY1 =>
                wait until B_rdy = '1';
                gotB_rdy <= '1';
                current_state <= B_ATTESA_BRDY0;
            when B_ATTESA_BRDY0 =>
                wait until B_rdy = '0';
                gotB_rdy <= '0';
                count := count +1;
        end case;
    end process;
```

```

        CASE count IS
            When 0 => NULL;
            When 1 => out_B1(07 downto 00)<=out_data;
current_state <= B_ATTESA_BRDY1;
            When 2 => out_B1(15 downto 08)<=out_data;
current_state <= B_ATTESA_BRDY1;
            When 3 => out_B2(07 downto 00)<=out_data;
current_state <= B_ATTESA_BRDY1;
            When 4 => out_B2(15 downto 08)<=out_data;
current_state <= B_ATTESA_BRDY1;
        end case;
    end case;
    wait on current_state, B_rdy, out_data;
end process;

end Behavioral;

```

6.4 Simulazione

Per la simulazione e la verifica di funzionamento del sistema sono state introdotte delle attese fittizie mediante l'utilizzo del costrutto `wait for` per simulare il fatto che il sistema A che invia le parole possa impiegare del tempo per preparare tali parole. Viene creato il *testbench*, il cui codice è mostrato di seguito:

```

ENTITY system_tb IS
END system_tb;

ARCHITECTURE behavior OF system_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT system
    PORT(
        in_A1 : IN  std_logic_vector(15 downto 0);
        in_A2 : IN  std_logic_vector(15 downto 0);
        out_B1 : OUT std_logic_vector(15 downto 0);
        out_B2 : OUT std_logic_vector(15 downto 0)
    );
    END COMPONENT;

    --Inputs
    signal in_A1 : std_logic_vector(15 downto 0) := (others => '0');
    signal in_A2 : std_logic_vector(15 downto 0) := (others => '0');

    --Outputs
    signal out_B1 : std_logic_vector(15 downto 0) := (others => '0');
    signal out_B2 : std_logic_vector(15 downto 0) := (others => '0');

BEGIN

```

```

-- Instantiate the Unit Under Test (UUT)
uut: system PORT MAP (
    in_A1 => in_A1,
    in_A2 => in_A2,
    out_B1 => out_B1,
    out_B2 => out_B2
);

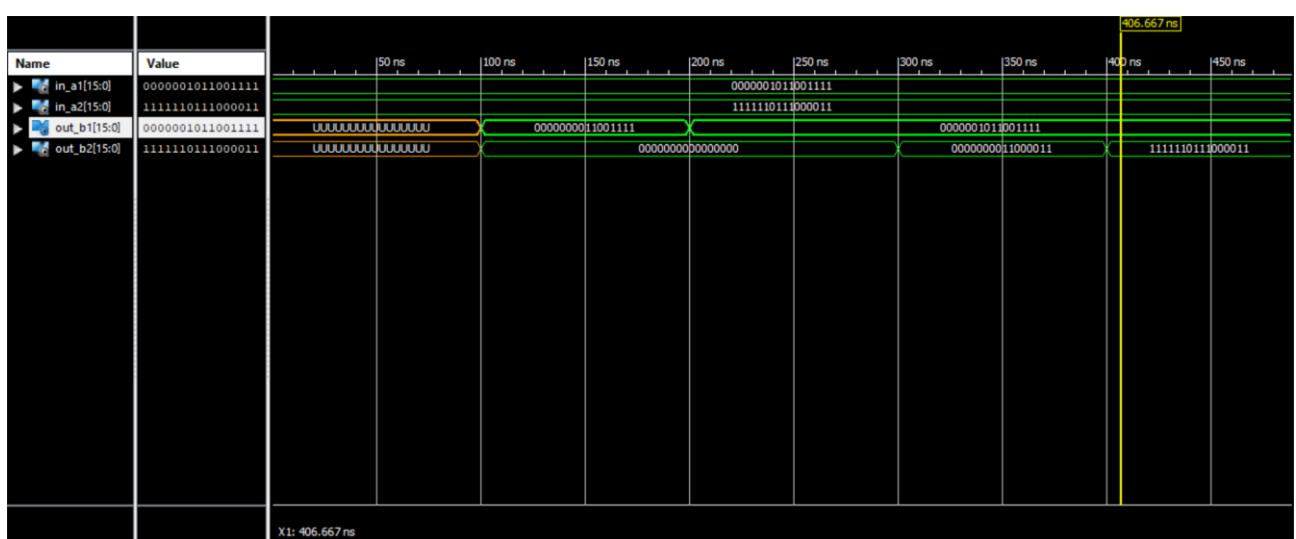
-- Stimulus process
stim_proc: process
begin
    -- insert stimulus here
    in_A1<="0000001011001111";
    in_A2<="1111110111000011";
    wait for 400 ns;
    assert out_B1 = "0000001011001111"
    report "errore0"
    severity failure;
    wait for 400 ns;
    assert out_B2 = "1111110111000011"
    report "errore1"
    severity failure;

    wait;
end process;

END;

```

I risultati della simulazione sono riportati nella seguente figura:



Esercizio 7

7.1 Traccia

Esercizio 7

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

$$\sum_{i=0}^{M-1} A(i) * B(i)$$

Il sistema deve essere alimentato con k coppie di vettori A e B diversi (cioè [A0, B0], [A1,B1],...,,[Ak,Bk]), forniti in uno dei modi seguenti (a scelta dello studente):

1. Tutti i vettori Aj e Bj (j=1, ..., k) sono precaricati in una ROM, e ciascuna coppia è fornita alla macchina in parallelo;
2. Tutti i vettori sono precaricati, e la macchina riceve serialmente gli elementi di ciascuna coppia di vettori tramite l'ausilio di registri a scorrimento (es., nel caso di M=3, vengono forniti in sequenza [A0(0), B0(0)], poi [A0(1), B0(1)], e poi [A0(2), B0(2)]; successivamente, vengono forniti [A1(0), B1(0)], [A1(1), B1(1)] e [A1(2), B1(2)], e così via);
3. Ciascuna coppia di vettori viene ricevuta da un'entità produttore mediante handshaking (e gestita in modalità parallela o seriale a seconda dell'architettura scelta);

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

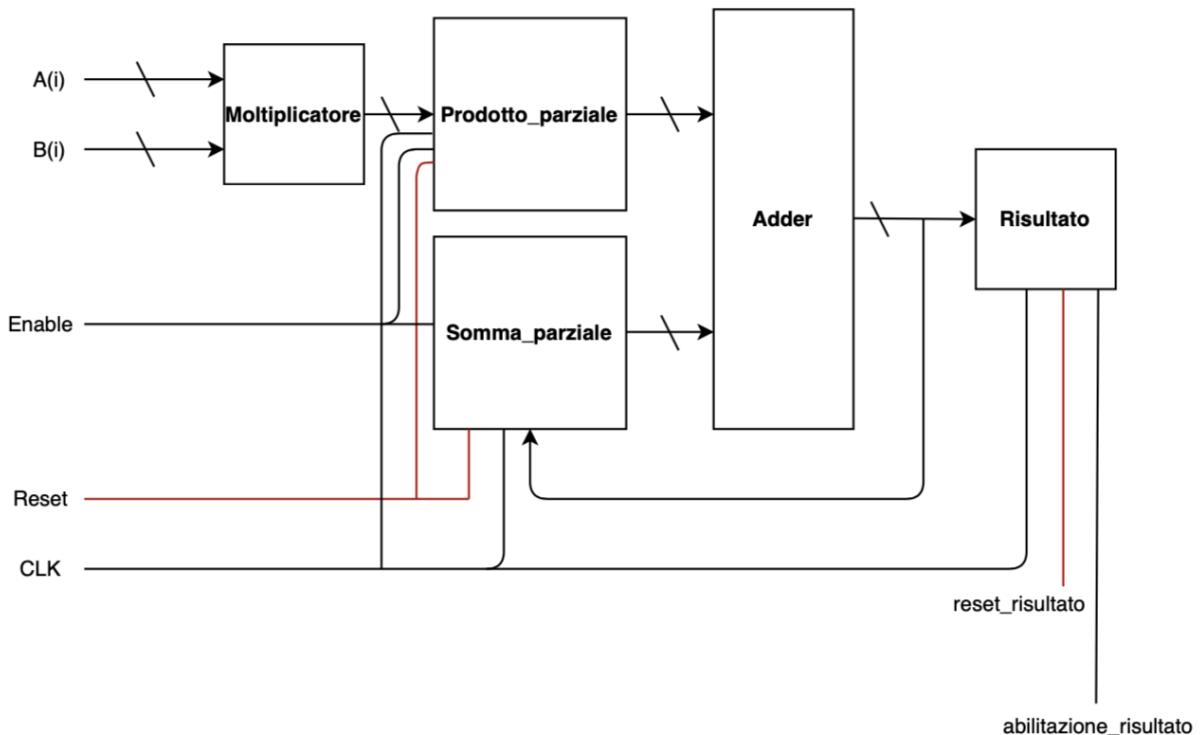
7.2 Soluzione

7.2.1 Implementazione con vettori in rom

Per lo svolgimento dell'esercizio sono stati considerati 3 coppie di vettori ($k = 3$), ognuno costituito da 3 elementi ($M = 3$) di 4 bit ($N = 4$).

Il sistema è costituito da una unità operativa e una unità di controllo. L'unità operativa è realizzata attraverso un datapath pipelined, questa presenta al suo interno un Adder, un moltiplicatore e tre registri. Il moltiplicatore prende in ingresso una coppia di elementi di ciascun vettore, il primo operando è l'elemento i-esimo del vettore A e il secondo operando è l'elemento i-esimo del vettore B e li moltiplica, ponendo successivamente il risultato nel primo registro, chiamato "Prodotto_parziale". L'uscita del registro "Prodotto_parziale" rappresenta il primo operando in ingresso all'addizionatore. Il secondo operando dell'addizionatore è l'uscita di un secondo registro, chiamato "Somma_parziale". Il sommatore calcola la somma dei due operandi e pone il risultato in ingresso al registro "Somma_parziale", perché questo permette di aggiungere, di volta in volta, alla somma appena calcolata il prodotto i+1-esimo dato in uscita al registro "Prodotto_parziale" nel successivo colpo di clock. Il datapath pipelined sfrutta il fatto che mentre il sommatore sta sommando le uscite precedenti del moltiplicatore, quest'ultimo può essere alimentato da altre coppie di elementi dei due vettori da moltiplicare, in questo modo aumenta la produttività del sistema e c'è maggiore disaccoppiamento tra l'unità che si occupa del calcolo della somma e l'unità che si occupa del calcolo del prodotto. Infine, è presente un ulteriore registro chiamato "Risultato", questo ha il compito di memorizzare il risultato finale, al termine di ogni prodotto scalare di due vettori. A tale scopo questo viene abilitato, con l'ausilio di un segnale ricevuto dall'unità di controllo, chiamato "abilitazione_risultato", solo quando l'unità operativa ha terminato il calcolo complessivo del

prodotto scalare e una volta memorizzato il risultato lo mantiene fino a quando non viene sovrascritto dal risultato ottenuto dal prodotto scalare dei vettori successivi.



L'unità di controllo è implementata mediante logica cablata. Essa presenta una MicroROM, nella quale sono già precaricati i vettori da elaborare. I record contenuti all'interno della rom sono costituiti da 5 campi: PC_next, indica il valore del PC successivo, il quale punterà, a seconda dei casi, ai successivi elementi del medesimo vettore in memoria, oppure, nel caso in cui il vettore non abbia più elementi e ci sia una nuova coppia di vettori da elaborare, punterà ai primi elementi della nuova coppia di vettori, altrimenti punterà ad PC = "0000", indicante lo stato di idle; A, che indica il valore di uno degli elementi del vettore A; B, che indica il valore di uno degli elementi del vettore B; EN, che indica l'abilitazione dei registri "Prodotto_parziale" e "Somma_parziale"; RESET, che indica il segnale di reset dei registri "Prodotto parziale" e "Somma parziale".

Sono presenti 10 record all'interno della ROM: il primo è relativo allo stato idle, e presenta come PC_next = "0000", ovvero punta a sé stesso. Questo serve per resettare tutti i registri interni all'unità operativa, in attesa di vettori da elaborare. Gli altri stati sono dovuti alle coppie di elementi dei due vettori da elaborare. L'unità di controllo presenta, inoltre, un componente, implementato con approccio behavioral, che permette di pilotare la ROM. Questo posiziona il PC all'indirizzo dei primi elementi della prima coppia di vettori da calcolare quando campiona il segnale di start sul fronte di salita del clock, e successivamente si limita a "confermare" il PC ottenuto dal record della ROM quando si verifica il fronte di salita del clock, con l'unica eccezione che quando sono finiti gli elementi di un vettore, bisogna memorizzare il risultato nel registro "Risultato" e deve indicare all'unità operativa di predisporsi per elaborare la successiva coppia di vettori. Quest'ultima operazione viene fatta inviando all'unità operativa il segnale "abilitazione_risultato" e ponendo per un ciclo di clock il PC = "0000", mettendo così il sistema in idle per resettare i registri parziali e in seguito al fronte di salita del clock successivo pone il PC pari all'indirizzo in memoria dei primi elementi della successiva coppia da elaborare.

7.2.2 Implementazione con handshake e contatore

Riportiamo un'altra soluzione, relativa al punto 3 della traccia. Facciamo le seguenti ipotesi: il protocollo è sincrono con il via in fase di inserimento dei vettori ed è semi-sincrono in fase di return del risultato, quindi il “client” ed il “server” hanno un clock comune; le coppie di vettori sono fornite in parallelo, il datapath è pipelined e l'automa responsabile sia per l'esecuzione del protocollo sia per il controllo dell'unità operativa è realizzato in logica cablata. Inoltre, il numero di coppie di vettori si assume che non sia noto a priori, e quindi si richiede al client di inserire in parallelo anche un altro valore, relativo al numero di coppie da elaborare. Il numero di “iterazioni” è quindi gestito mediante un contatore. L'architettura dell'unità operativa è simile alla precedente, senza un registro per il risultato: il risultato si trova nel registro somma_parziale, se si usano opportunamente le abilitazioni.

Il server può assumere tre possibili stati: **idle**, **data_in** ed **end_op**. Il funzionamento è il seguente:

- Il client invia il segnale di via al server, avendo già caricato il valore relativo al numero di iterazioni, detto M.
- Il server resetta i suoi registri, invia al client una conferma, si predisponde per cambiare stato da idle a data_in, abilita il caricamento parallelo di una word nel contatore e carica il valore $\text{max_cnt} - M + 1$, in modo tale che il contatore effettui $M - 1$ conteggi prima di segnalare la condizione di overflow (ne deve fare $M - 1$ e non M perché comincia a contare solo in corrispondenza del colpo di clock successivo a quando si abbassa il segnale relativo al caricamento parallelo, mentre la pipeline può già essere abilitata).
- In corrispondenza di ciascun colpo di clock successivo, per un totale di M colpi di clock, il client carica una coppia di vettori di N bit; il server controlla la condizione di overflow del contatore, e se non è verificata allora tiene alto il segnale di abilitazione “step” per i registri relativi al prodotto parziale ed alla somma parziale, i quali sono sincronizzati sul fronte di salita, e quindi si ricava il segnale di abilitazione per i registri facendo la AND tra il segnale step ed il clock.
- Quando il contatore va in overflow, il segnale di overflow viene schedulato per il colpo di clock successivo; quando il server rileva che l'overflow è pari a 1, l'atto di abbassare il segnale di abilitazione dei registri è schedulato per il colpo di clock successivo. Questo vuol dire che in totale l'abilitazione è data $M + 2$ volte: è giusto, perché se una coppia di vettori è in ingresso all'istante T, allora all'istante $T + \Delta$ il loro prodotto sarà scritto nel registro prodotto parziale, e all'istante $T + 2\Delta$ sarà scritto il registro somma parziale. Oltre a schedulare $\text{step} \leq '0'$ per il colpo di clock successivo, il server si predisponde per cambiare stato da data_in a end_op.
- Quando il server si trova nello stato end_op, invia al client informazione riguardo alla terminazione dell'operazione, mediante appunto un segnale “end_op”, e si mette in attesa della conferma da parte del client.
- Quando il client riceve il segnale “end_op” alto, consuma il dato, invia un acknowledgement al server e poi aspetta che il server abbassi il segnale “end_op”.
- Infine, quando il server riceve l'ack dal client relativo al segnale “end_op”, abbassa tale segnale e si porta nello stato idle.

E' stata fatta un'implementazione generic: le coppie di vettori sono rappresentate su N bit, si hanno N bit per specificare quante iterazioni si vogliono fare e quindi l'uscita è su $3*N$ bit. I componenti contatore, moltiplicatore e sommatore sono stati implementati in maniera behavioral; l'unità di controllo, detta “consumatore”, è composta in maniera strutturale dall'automa che implementa sia il protocollo sia appunto la logica di controllo, e dal contatore. Il top module quindi comprende i moduli dell'unità operativa e l'unità di controllo.

7.3 Codice

7.3.1 Implementazione con vettori in rom

Di seguito è riportato il codice del sistema per il calcolo del prodotto scalare:

```
entity ProdScal is
    Port ( CLK : in STD_LOGIC;
            START : in STD_LOGIC;
            AB : out STD_LOGIC_VECTOR(15 downto 0));
end ProdScal;

architecture Structural of ProdScal is

signal EN : std_logic:='0';
signal RESET : std_logic:='0';

signal A : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
signal B : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');

signal abilitazione_risultato : std_logic:='0';
signal reset_risultato : std_logic:='0';

Component unita_operativa is
    Port (      CLK : in STD_LOGIC;
                EN : in STD_LOGIC;
                RESET : in STD_LOGIC;
                A : in STD_LOGIC_VECTOR(7 downto 0);
                B : in STD_LOGIC_VECTOR(7 downto 0);
                AB : out STD_LOGIC_VECTOR(15 downto 0);
                abilitazione_risultato : in std_logic;
                reset_risultato : in std_logic
);
end component;

Component unita_controllo is
    Port ( CLK : in STD_LOGIC;
            START : in STD_LOGIC;
            EN : out STD_LOGIC;
            RESET : out STD_LOGIC;
            A : out STD_LOGIC_VECTOR(7 downto 0);
            B : out STD_LOGIC_VECTOR(7 downto 0);
            abilitazione_risultato : out std_logic;
            reset_risultato : out std_logic
);

end component;

begin
    U_0: unita_operativa
```

```

    Port map(CLK=>CLK,EN=>EN,RESET=>RESET,A=>A,B=>B,AB=>AB,
abilitazione_risultato => abilitazione_risultato,
reset_risultato=>reset_risultato);

    U_C: unita_controllo
    Port map(CLK=>CLK,START=>START,EN=>EN,RESET=>RESET,A=>A,B=>B,
abilitazione_risultato=>abilitazione_risultato,reset_risultato=>reset_risultato);

end Structural;

```

Di seguito è riportato il codice dell'unità operativa:

```

entity unita_operativa is
    Port ( CLK : in STD_LOGIC;
            EN : in STD_LOGIC;
            RESET : in STD_LOGIC;
            A : in STD_LOGIC_VECTOR(7 downto 0);
            B : in STD_LOGIC_VECTOR(7 downto 0);
            AB : out STD_LOGIC_VECTOR(15 downto 0);
            abilitazione_risultato : in STD_LOGIC;
            reset_risultato : in STD_LOGIC
        );
end unita_operativa;

architecture Structural of unita_operativa is

    signal ApiuB : STD_LOGIC_VECTOR(15 downto 0) :=
(others=>'0');
    signal AperB : STD_LOGIC_VECTOR(15 downto 0) :=
(others=>'0');
    signal S_Parziale : STD_LOGIC_VECTOR(15 downto 0) :=
(others=>'0');
    signal P_Parziale : STD_LOGIC_VECTOR(15 downto 0) :=
(others=>'0');
    signal Accumulatore : STD_LOGIC_VECTOR(15 downto 0) :=
(others=>'0');

Component Adder is
    Port ( A : in STD_LOGIC_VECTOR(15 downto 0);
            B : in STD_LOGIC_VECTOR(15 downto 0);
            ApiuB : out STD_LOGIC_VECTOR(15 downto 0)
        );
end component;

Component Moltiplicatore is
    Port ( A : in STD_LOGIC_VECTOR(7 downto 0);
            B : in STD_LOGIC_VECTOR(7 downto 0);
            AperB : out STD_LOGIC_VECTOR(15 downto 0)
        );

```

```

        );
end component;

Component Registro is
    Port ( CLK : in STD_LOGIC;
            EN: in STD_LOGIC;
            RESET : in STD_LOGIC;
            D : in STD_LOGIC_VECTOR(15 downto 0);
            Q : out STD_LOGIC_VECTOR(15 downto 0)
        );
end component;

Begin
    Result: Registro
    Port
map(CLK=>CLK,EN=>abilitazione_risultato,RESET=>reset_risultato,D=>ApiuB,Q=>AB);
    Somma_parziale: Registro
    Port map(CLK=>CLK,EN=>EN,RESET=>RESET,D=>ApiuB,Q=>S_Parziale);
    Prodotto_parziale: Registro
    Port map(CLK=>CLK,EN=>EN,RESET=>RESET,D=>AperB,Q=>P_Parziale);
    Prodotto: Moltiplicatore
    Port map(A=>A,B=>B,AperB=>AperB);
    Somma: Adder
    Port map(A=>P_Parziale,B=>S_Parziale,ApiuB=>ApiuB);
End Structural;

```

Di seguito sono riportati i codici per i componenti che costituiscono l'unità operativa:

```

entity Moltiplicatore is
    Port ( A : in STD_LOGIC_VECTOR(7 downto 0);
            B : in STD_LOGIC_VECTOR(7 downto 0);
            AperB : out STD_LOGIC_VECTOR(15 downto 0));
end Moltiplicatore;

architecture Behavioral of Moltiplicatore is
begin
    AperB <= std_logic_vector(unsigned(A) * unsigned(B));
end Behavioral;

entity Adder is
    Port ( A : in STD_LOGIC_VECTOR(15 downto 0);
            B : in STD_LOGIC_VECTOR(15 downto 0);
            ApiuB : out STD_LOGIC_VECTOR(15 downto 0));
end Adder;

architecture Behavioral of Adder is
begin

```

```

Ap1uB <= std_logic_vector(unsigned(A) + unsigned(B));

end Behavioral;

entity Registro is
  Port ( CLK : in STD_LOGIC;
         EN: in STD_LOGIC;
         RESET : in STD_LOGIC;
         D : in STD_LOGIC_VECTOR(15 downto 0);
         Q : out STD_LOGIC_VECTOR(15 downto 0)
      );
end Registro;

architecture Structural of Registro is

  component FlipFlop is
    port( CLK : in STD_LOGIC;
          EN: in STD_LOGIC;
          RESET: in STD_LOGIC;
          D : in STD_LOGIC;
          Q : out STD_LOGIC
        );
  end component;

begin

  reg15to0: for i in 15 downto 0 generate
    FF: FlipFlop
      Port map( CLK => CLK,
                EN => EN,
                RESET => RESET,
                D => D(i),
                Q => Q(i)
              );
  end generate;

end Structural;

entity FlipFlop is
  Port ( CLK : in STD_LOGIC;
         EN : in STD_LOGIC;
         RESET: in STD_LOGIC;
         D : in STD_LOGIC;
         Q : out STD_LOGIC
      );
end FlipFlop;

```

```

architecture rtl of FlipFlop is
begin
    ff: process( CLK )
    begin
        if( CLK'event and CLK = '1' ) then
            if( RESET = '1' ) then
                Q <= '0';
            elsif(EN= '1') then
                Q <= D;
            end if;
        end if;
    end process;

end rtl;

```

Di seguito è riportato il codice dell'unità di controllo:

```

entity unita_controllo is
    Port ( CLK : in STD_LOGIC;
            START : in STD_LOGIC;
            EN : out STD_LOGIC;
            RESET : out STD_LOGIC;
            A : out STD_LOGIC_VECTOR(7 downto 0);
            B : out STD_LOGIC_VECTOR(7 downto 0);
            abilitazione_risultato : out STD_LOGIC;
            reset_risultato : out STD_LOGIC);
end unita_controllo;

architecture Behavioral of unita_controllo is

signal PC_next, PC, Next_Prod : unsigned(3 downto 0) := "0000";

Component MicroROM is
    Port ( PC : in unsigned(3 downto 0);
            PC_next : out unsigned(3 downto 0);
            A : out STD_LOGIC_VECTOR(7 downto 0);
            B : out STD_LOGIC_VECTOR(7 downto 0);
            EN : out STD_LOGIC;
            RESET : out STD_LOGIC
    );
end component;

begin

    rom: MicroROM
    PORT MAP(PC => PC,
              PC_next => PC_next,
              A => A,
              B => B,

```

```

        EN => EN,
        RESET => RESET
    );

reg_PC: PROCESS(CLK)
variable counter : integer range 0 to 6 := 0;
BEGIN
    if(CLK'event and CLK = '1') then
        if(start = '1') then
            PC <= "0001";
            counter := 0;
            reset_risultato <= '1';
        else
            reset_risultato <= '0';
            PC<= PC_next;
            counter := counter +1;
            if(counter = 3) then
                Next_Prod <= PC_next;
                abilitazione_risultato <= '1';
            end if;
            if(counter = 4) then
                abilitazione_risultato <= '0';
            end if;
            if(counter = 5) then
                PC <= "0000";
            end if;
            if(counter = 6) then
                PC <= Next_Prod;
                counter := 0;
            end if;
        end if;
    end if;
end process reg_PC;

end Behavioral;

```

Di seguito è riportati il codice della MicroRom:

```

entity MicroROM is
    Port ( PC : in  unsigned(3 downto 0);
           PC_next : out unsigned(3 downto 0);
           A : out STD_LOGIC_VECTOR(7 downto 0);
           B : out STD_LOGIC_VECTOR(7 downto 0);
           EN : out STD_LOGIC;
           RESET : out STD_LOGIC
    );
end MicroROM;

architecture synth of MicroROM is

```

```

type Controllo_type is record
    PC_next      : unsigned(3 downto 0);
    A            : std_logic_vector(7 downto 0);
    B            : std_logic_vector(7 downto 0);
    EN           : std_logic;
    RESET        : std_logic;
end record;

constant idle : Controllo_type := (
    PC_next => "0000",
    A    => "00000000",
    B    => "00000000",
    EN   => '0',
    RESET=> '1'
);
constant k00 : Controllo_type := (
    PC_next => "0010",
    A => "00000011",
    B    => "00000101",
    EN   => '1',
    RESET=> '0'
);
constant k01 : Controllo_type := (
    PC_next => "0011",
    A => "00000010",
    B    => "00000110",
    EN   => '1',
    RESET=> '0'
);
constant k02 : Controllo_type := (
    PC_next => "0100",
    A => "00000001",
    B    => "00000011",
    EN   => '1',
    RESET=> '0'
);
constant k10 : Controllo_type := (
    PC_next => "0101",
    A    => "00000100",
    B    => "00000110",
    EN   => '1',
    RESET=> '0'
);
constant k11 : Controllo_type := (
    PC_next => "0110",
    A    => "00000011",
    B    => "00000111",
    EN   => '1',
    RESET=> '0'
);

```

```

    );
constant k12 : Controllo_type := (
    PC_next => "0111",
    A => "00000010",
    B => "00000100",
    EN => '1',
    RESET => '0'
);
constant k20 : Controllo_type := (
    PC_next => "1000",
    A => "00000101",
    B => "00000111",
    EN => '1',
    RESET => '0'
);
constant k21 : Controllo_type := (
    PC_next => "1001",
    A => "00000001",
    B => "00001000",
    EN => '1',
    RESET => '0'
);
constant k22 : Controllo_type := (
    PC_next => "0000",
    A => "00000101",
    B => "00000011",
    EN => '1',
    RESET => '0'
);

type ROM_TYPE is ARRAY(0 to 9) of Controllo_type;

constant ROM : ROM_TYPE := (
    0 => idle,
    1 => k00,
    2 => k01,
    3 => k02,
    4 => k10,
    5 => k11,
    6 => k12,
    7 => k20,
    8 => k21,
    9 => k22
);

signal Controllo : Controllo_type;

begin

```

```

    Controllo <= ROM(conv_integer(PC));

    PC_next <= Controllo.PC_next;
    A         <= Controllo.A;
    B         <= Controllo.B;
    EN        <= Controllo.EN;
    RESET     <= Controllo.RESET;

end synth;

```

7.3.2 Implementazione con handshake e contatore

Top module:

```

entity prodotto_scalare is
  Generic (n : positive := 8);
  Port ( clk : in std_logic;
  rst : in std_logic;
  start : in std_logic;
  M : in unsigned(n-1 downto 0);
  a : in unsigned(n-1 downto 0);
  b : in unsigned(n-1 downto 0);
  ack_end_op : in std_logic;
  ack : out std_logic;
  end_op : out std_logic;
  y : out unsigned(3*n - 1 downto 0));
end prodotto_scalare;

architecture Structural of prodotto_scalare is

signal current_sum : unsigned(3*n - 1 downto 0) := (others => '0');
signal buff_a : unsigned(n-1 downto 0) := (others => '0');
signal buff_b : unsigned(n-1 downto 0) := (others => '0');
signal internal_rst : std_logic := '0';
signal rst_state : std_logic := '0';
signal step : std_logic := '0';
signal molt_out : unsigned(3*n - 1 downto 0) := (others => '0');
signal op_R : unsigned(3*n - 1 downto 0) := (others => '0');
signal add_out : unsigned(3*n - 1 downto 0) := (others => '0');
signal clk_en : std_logic := '0';

component cntrl_unit is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         step : in STD_LOGIC;
         en_S : out STD_LOGIC;
         en_R : out STD_LOGIC);
end component;

component consumatore is
  Generic(n : positive := 8);

```

```

Port ( clk : in STD_LOGIC;
       rst : in STD_LOGIC;
       syn_in : in STD_LOGIC;
       ack_in : in STD_LOGIC;
       a : in unsigned(n-1 downto 0);
       b : in unsigned(n-1 downto 0);
       M : in unsigned(n-1 downto 0);
       result_in : in unsigned(3*n - 1 downto 0);
       syn_out : out STD_LOGIC;
       ack_out : out STD_LOGIC;
       y : out unsigned(3*n - 1 downto 0);
       out_a : out unsigned(n-1 downto 0);
       out_b : out unsigned(n-1 downto 0);
       rst_out : out STD_LOGIC;
       step : out STD_LOGIC);
end component;

component registro is
  Generic(n : positive := 8);
  Port ( d : in unsigned(n-1 downto 0);
         rst : in STD_LOGIC;
         en : in STD_LOGIC;
         q : out unsigned(n-1 downto 0));
end component;

component sommatore is
  Generic (n : positive := 8);
  Port ( a : in unsigned(n-1 downto 0);
         b : in unsigned(n-1 downto 0);
         clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         y : out unsigned(n-1 downto 0));
end component;

component moltiplicatore is
  Generic(n : positive := 8);
  Port ( a : in unsigned(n-1 downto 0);
         b : in unsigned(n-1 downto 0);
         clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         y : out unsigned(3*n - 1 downto 0));
end component;

begin

frontend : consumatore generic map(n) port map(
clk => clk, rst => rst, syn_in => start, ack_in => ack_end_op, M => M, a
=> a,
result_in => current_sum, syn_out => end_op, ack_out => ack, y => y, b =>
b,

```

```

out_a => buff_a, out_b => buff_b, rst_out => internal_rst, step => step
);

rst_state <= rst or internal_rst;

clk_en <= clk and step;

molt : moltiplicatore generic map(n) port map(
a => buff_a, b => buff_b, y => molt_out, clk => clk, rst => rst_state
);

reg_R : registro generic map(3*n) port map(
d => molt_out, rst => rst_state, en => clk_en, q => op_R
);

reg_S : registro generic map(3*n) port map(
d => add_out, rst => rst_state, en => clk_en, q => current_sum
);

somma : sommatore generic map(3*n) port map(
a => current_sum, b => op_R, y => add_out, clk => clk, rst => rst_state
);

end Structural;

```

Consumatore:

```

entity consumatore is
Generic(n : positive := 8);
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         syn_in : in STD_LOGIC;
         ack_in : in STD_LOGIC;
         a : in unsigned(n-1 downto 0);
         b : in unsigned(n-1 downto 0);
         M : in unsigned(n-1 downto 0);
         result_in : in unsigned(3*n - 1 downto 0);
         syn_out : out STD_LOGIC;
         ack_out : out STD_LOGIC;
         y : out unsigned(3*n - 1 downto 0); -- è sufficiente se M <
n
         out_a : out unsigned(n-1 downto 0);
         out_b : out unsigned(n-1 downto 0);
         rst_out : out STD_LOGIC;
         step : out STD_LOGIC);
end consumatore;

architecture Structural of consumatore is

signal overflow : std_logic := '0';

```

```

signal set : std_logic := '0';
signal par_cnt : unsigned(n-1 downto 0) := (others => '0');
type state is (idle, data_in, end_op);
signal current : state := idle;

component contatore is
  Generic(n : positive := 8);
  Port ( clk : in STD_LOGIC;
          set : in STD_LOGIC;
          par : in unsigned(n-1 downto 0);
          rst : in STD_LOGIC;
          overflow : out STD_LOGIC);
end component;

begin

count : contatore generic map(n) port map(
clk => clk, set => set, par => par_cnt, rst => rst, overflow => overflow
);

out_a <= a;
out_b <= b;
y <= result_in;

cons : process(clk, rst)
constant max_cont : unsigned(n-1 downto 0) := (others => '1');
begin
if (rst = '1') then
current <= idle; syn_out <= '0'; ack_out <= '0'; rst_out <= '0';
elsif (clk = '1' and clk'event) then
case current is
when idle =>
if (syn_in = '1') then
current <= data_in; ack_out <= '1'; rst_out <= '1'; step <= '0';
set <= '1'; par_cnt <= max_cont - M + 1;
else
current <= idle; rst_out <= '0'; step <= '0';
end if;
when data_in =>
if (overflow = '1') then
current <= end_op; set <= '0'; step <= '0'; rst_out <= '0';
else
current <= data_in; set <= '0';
step <= '1'; rst_out <= '0'; ack_out <= '0';
end if;
when end_op =>
if (ack_in = '0') then
current <= end_op; syn_out <= '1';
step <= '0';
else

```

```

current <= idle; rst_out <= '0'; step <= '0'; syn_out <= '0';
end if;
when others =>
current <= idle; rst_out <= '1';
end case;
end if;
end process;

end Structural;

```

Contatore:

```

entity contatore is
Generic(n : positive := 8);
Port ( clk : in STD_LOGIC;
       set : in STD_LOGIC;
       par : in unsigned(n-1 downto 0);
       rst : in STD_LOGIC;
       overflow : out STD_LOGIC);
end contatore;

architecture Behavioral of contatore is

signal cnt : unsigned(n-1 downto 0) := (others => '0');

begin
count : process(clk, set, rst)
constant maximum : unsigned(n-1 downto 0) := (others => '1');
begin
if (rst = '1') then
cnt <= (others => '0'); overflow <= '0';
elsif (set = '1') then
cnt <= par; overflow <= '0';
elsif (clk = '1' and clk'event) then
if (cnt = maximum) then
cnt <= (others => '0');
overflow <= '1';
else
cnt <= cnt + 1;
overflow <= '0';
end if;
end if;
end process;

end Behavioral;

```

Moltiplicatore:

```

entity moltiplicatore is
Generic(n : positive := 8);

```

```

Port ( a : in unsigned(n-1 downto 0);
       b : in unsigned(n-1 downto 0);
clk : in std_logic;
rst : in std_logic;
       y : out unsigned(3*n - 1 downto 0));
end moltiplicatore;

architecture Behavioral of moltiplicatore is
-- diviso 2 perché la moltiplicazione automaticamente ritorna un
risultato
-- che ha dimensione in bit doppia dell'ingresso
signal zero : unsigned(n/2 - 1 downto 0) := (others => '0');

begin
molt : process(clk, rst)
begin
if (rst = '1') then
y <= (others => '0');
elsif (clk'event and clk = '1') then
y <= (zero & a) * (zero & b);
end if;
end process;

end Behavioral;

```

Registro:

```

entity registro is
Generic(n : positive := 8);
Port ( d : in unsigned(n-1 downto 0);
       rst : in STD_LOGIC;
       en : in STD_LOGIC;
       q : out unsigned(n-1 downto 0));
end registro;

architecture Behavioral of registro is

begin
memory : process(rst, en)
begin
if (rst = '1') then
q <= (others => '0');
elsif (en = '1' and en'event) then
q <= d;
end if;
end process;

end Behavioral;

```

Sommatore:

```
entity sommatore is
  Generic (n : positive := 8);
  Port ( a : in unsigned(n-1 downto 0);
          b : in unsigned(n-1 downto 0);
        clk : in std_logic;
        rst : in std_logic;
        y : out unsigned(n-1 downto 0));
  -- dev'essere questa la dimensione dell'uscita altrimenti non si può
  mettere in feedback nel registro
end sommatore;

architecture Behavioral of sommatore is

begin
  somma : process(clk, rst)
  begin
    if (rst = '1') then
      y <= (others => '0');
    elsif (clk'event and clk = '1') then
      y <= a + b;
    end if;
  end process;

end Behavioral;
```

7.4 Simulazione

7.4.1 Implementazione con vettori in rom

Per la simulazione e la verifica di funzionamento del sistema viene creato il *testbench*, il cui codice è mostrato di seguito:

```
ENTITY testbench IS
  END testbench;

ARCHITECTURE behavior OF testbench IS

  -- Component Declaration
  COMPONENT ProdScal
  PORT(
    CLK : IN std_logic;
    START : IN std_logic;
    AB : OUT std_logic_vector(15 downto 0)
  );
  END COMPONENT;

  SIGNAL START : std_logic := '0';
  SIGNAL CLK : std_logic := '0';
```

```

    SIGNAL AB : std_logic_vector(15 downto 0) := (others => '0');
    constant CLK_period : time := 10 ns;

BEGIN

-- Component Instantiation
    uut: ProdScal PORT MAP(
        CLK => CLK,
        START => START
    );

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

-- Test Bench Statements
tb : PROCESS
BEGIN

    wait for 100 ns;
    START <= '1';
    wait for CLK_period;
    START <= '0';
    wait for 4*CLK_period;
    assert AB="0000000000011110"
    report "errore0"
    severity failure;
    wait for 4*CLK_period;
    assert AB="0000000000110101"
    report "errore1"
    severity failure;
    wait for 4*CLK_period;
    assert AB="0000000000111010"
    report "errore2"
    severity failure;

    wait;
END PROCESS tb;
-- End Test Bench

END;

```

Il sistema è stato testato con le seguenti coppie di vettori già precaricate in memoria:

```
A0 = ["00000011", "00000010", "00000001"]
B0 = ["00000101", "00000110", "00000011"]
```

```
A1 = ["00000100", "00000011", "00000010"]  
B1 = ["00000110", "00000111", "00000100"]
```

```
A2 = ["00000101", "00000001", "00000101"]  
B2 = ['00000111', "00001000", "00000011"]
```

I risultati della simulazione sono riportati nella seguente figura:



7.4.2 Implementazione con handshake e contatore

Testbench:

```
ENTITY prod_scalare_testbench IS
END prod_scalare_testbench;

ARCHITECTURE behavior OF prod_scalare_testbench IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT prodotto_scalare
        PORT(
            clk : IN std_logic;
            rst : IN std_logic;
            start : IN std_logic;
            M : IN unsigned(7 downto 0);
            a : IN unsigned(7 downto 0);
            b : IN unsigned(7 downto 0);
            ack_end_op : IN std_logic;
            ack : OUT std_logic;
            end_op : OUT std_logic;
            y : OUT unsigned(23 downto 0)
        );
    END COMPONENT;

    --Inputs
    signal clk : std_logic := '0';
    signal rst : std_logic := '0';
    signal start : std_logic := '0';
    signal M : unsigned(7 downto 0) := (others => '0');
    signal a : unsigned(7 downto 0) := (others => '0');
    signal b : unsigned(7 downto 0) := (others => '0');
    signal ack_end_op : std_logic := '0';

    --Outputs
    signal ack : std_logic;
    signal end_op : std_logic;
    signal y : unsigned(23 downto 0);

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

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: prodotto_scalare PORT MAP (
        clk => clk,
        rst => rst,
```

```

        start => start,
        M => M,
        a => a,
        b => b,
        ack_end_op => ack_end_op,
        ack => ack,
        end_op => end_op,
        y => y
    );
}

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

-- Processo client (produttore)
produttore: process
begin
    -- hold reset state for 100 ns.
    wait for 100 ns;
rst <= '1';
    wait for clk_period;
rst <= '0';
    wait for clk_period;

M <= to_unsigned(3, 8);
start <= '1';
-- il protocollo è sincrono ma ci vuole una sincronizzazione di fase
iniziale
wait until (ack = '1');
start <= '0';
wait for clk_period;
a <= to_unsigned(2, 8);
b <= to_unsigned(4, 8);
wait for clk_period;
a <= to_unsigned(10, 8);
b <= to_unsigned(2, 8);
wait for clk_period;
a <= to_unsigned(7, 8);
b <= to_unsigned(4, 8);
wait until (end_op = '1'); -- il protocollo è semisincrono
ack_end_op <= '1';
assert (y = to_unsigned(56, 24));
wait until (end_op = '0'); -- il protocollo è semisincrono
ack_end_op <= '0';

```

```

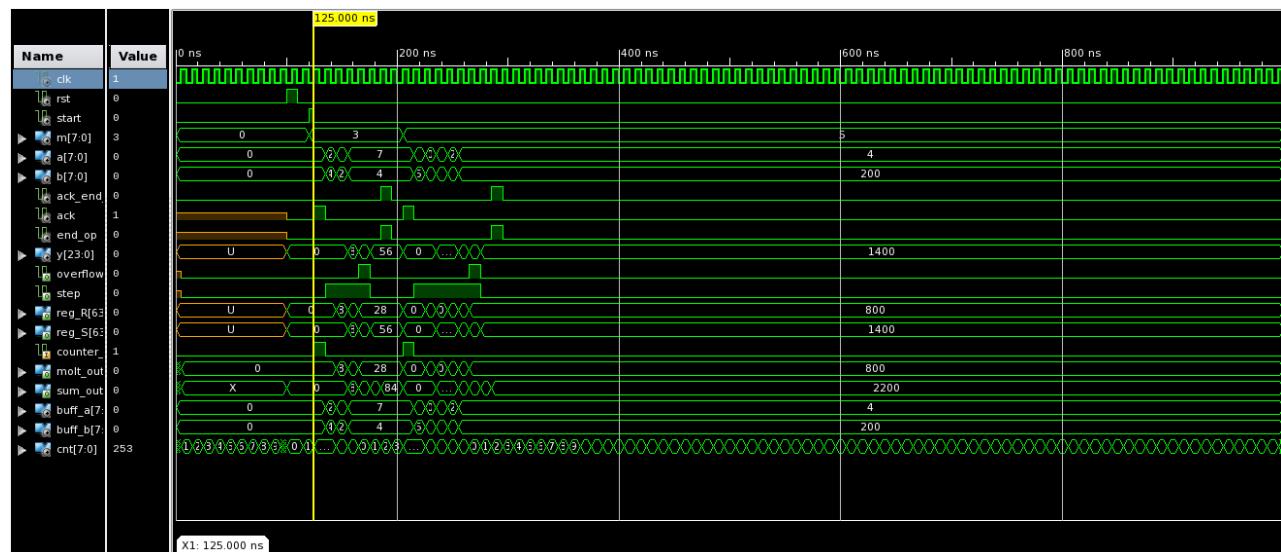
wait for clk_period;

M <= to_unsigned(5, 8);
start <= '1';
wait until (ack = '1');
start <= '0';
wait for clk_period;
a <= to_unsigned(60, 8);
b <= to_unsigned(5, 8);
wait for clk_period;
a <= to_unsigned(0, 8);
b <= to_unsigned(200, 8);
wait for clk_period;
a <= to_unsigned(10, 8);
b <= to_unsigned(10, 8);
wait for clk_period;
a <= to_unsigned(2, 8);
b <= to_unsigned(100, 8);
wait for clk_period;
a <= to_unsigned(4, 8);
b <= to_unsigned(200, 8);
wait until (end_op = '1');
ack_end_op <= '1';
assert (y = to_unsigned(1400, 24));
wait until (end_op = '0');
ack_end_op <= '0';

wait;
end process;

```

END;



Esercizio 8

8.1 Traccia

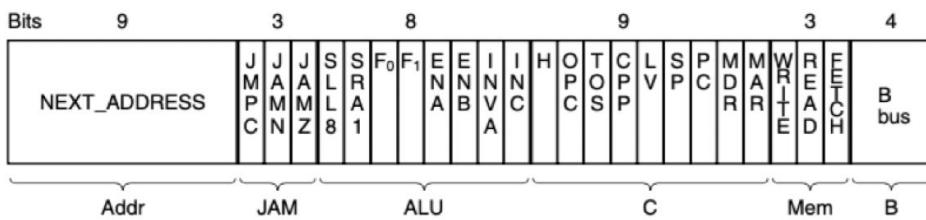
A partire dall'implementazione fornita di un processore operante secondo il modello IJVM, si proceda all'analisi dell'architettura mediante simulazione e si approfondisca lo studio del suo funzionamento per due istruzioni a scelta, si modifichi un codice operativo a scelta, documentando tutte le modifiche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

8.2 Richiamo sull'architettura

Prima di procedere con l'analisi mediante simulazione, facciamo un breve richiamo sull'architettura del processore MIC-1.

Il processore MIC-1 è una macchina a stack: data una procedura in esecuzione, esiste uno spazio di memoria indirizzato a partire da un indirizzo base, nel quale vengono scritte le variabili locali della procedura. Essendo a stack, lo spiazzamento rispetto all'indirizzo base si riferisce sempre all'elemento che è considerato il top dello stack. Un'architettura di questo tipo è molto conveniente per realizzare operazioni aritmetiche. Dunque, nel datapath ci aspettiamo un registro che contiene l'indirizzo base dello stack (**LV**) ed un indirizzo che contiene lo spiazzamento (**SP**). Inoltre, per ridurre il numero di accessi da fare in memoria e quindi velocizzare le operazioni, conviene avere anche un registro in cui è presente il valore dell'elemento presente in cima allo stack (**TOS**). Supponiamo di avere una ALU a due operandi, e di avere un solo bus per leggere dai registri, i quali vi accederanno in mutua esclusione: come facciamo ad effettuare un'operazione tra valori presenti in due diversi registri? E' chiaro che ci serve un registro direttamente connesso alla ALU (**H**), in cui bisogna pre-caricare un operando prima di poter effettuare l'operazione desiderata a due operandi; tale registro è comunemente detto "registro tampone". Poi, tipicamente chi programma in assembly ha la possibilità di definire delle costanti: per questo motivo è necessario anche un registro che mantiene l'indirizzo base dell'area di memoria in cui sono presenti le costanti (**CPP**), e di nuovo si avrà un indirizzamento del tipo "indirizzo base" + "spiazzamento", dove lo spiazzamento sarà passato come operando di un'istruzione assembly, eventualmente sarà fatto uno shift per implementare una logica del tipo "indirizzo relativo multiplo di N bit" e la somma per ottenere l'indirizzo finale sarà ovviamente fatta dalla ALU. Poi ci servono alcuni registri tipici di ogni processore: uno che contiene indirizzi assoluti per effettuare operazioni di lettura/scrittura con la memoria (**MAR**), il quale viene scritto in seguito ad una somma fatta dalla ALU (ad esempio $LV + H$), ed il cui valore va verso l'esterno insieme a dei **segnali di controllo della memoria**, come *read enable* e *write enable*; accoppiato a tale registro, ci serve un altro registro (**MDR**) che contiene il risultato della lettura dalla memoria oppure dualmente il valore da scrivere in memoria; poi, supponendo di avere un programma assembly caricato in memoria RAM, è imperativo avere un registro che contiene l'indirizzo della prossima istruzione (**PC**), il quale serve come il MAR a fare operazioni di I/O, ma in questo caso si effettuano solo letture e quindi si ha un solo segnale di controllo, detto *fetch*; il risultato della lettura dell'istruzione dalla RAM è salvato in un ulteriore registro (**MBR**). Tra le possibili istruzioni assembly, ci sono quelle di salto: anche i salti sono gestiti mediante indirizzo base più spiazzamento; tuttavia, a questo punto è necessaria una nota sul parallelismo del datapath: i registri sono su 32 bit, tranne l'MBR che è a 8 bit, e la ALU vuole operandi a 32 bit. Questo vuol dire che non viene letta un'intera istruzione con una singola fetch: si preleva con la prima fetch il codice operativo, poi l'operando con una seconda fetch. Dunque, non si può saltare direttamente rispetto al PC, ma è necessario salvare in un registro un "vecchio" valore base del PC relativamente al quale saltare

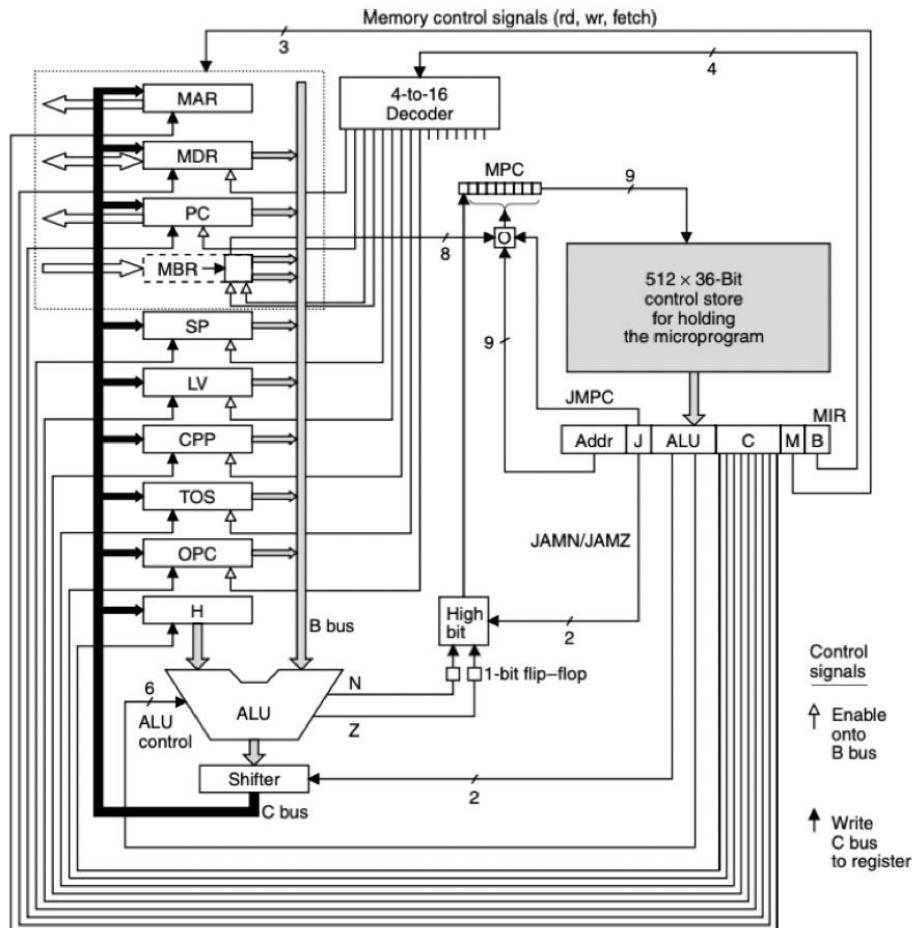
(**OPC**); di nuovo si ha la logica del tipo indirizzo base più spiazzamento. L'MBR può contenere sia valori signed sia valori unsigned, e deve essere esteso a 32 bit: quindi ci sono due modi per accedere al valore di MBR, uno si chiama "MBR" ed è signed, quindi si fa un'estensione con segno (con bit pari a 0 se il segno è positivo altrimenti con bit pari ad 1), e l'altro si chiama "MBRU" ed è unsigned, quindi prevede un'estensione con bit pari a 0. Il bus di lettura è detto bus B e come anticipato si può leggere da un solo registro alla volta, quindi ci aspettiamo l'utilizzo o di un mux o di un decoder per selezionare il registro dal quale leggere, mentre il bus di scrittura è detto bus C e si può scrivere su più registri nello stesso colpo di clock, ad eccezione del registro MBR, ovviamente. La ALU può effettuare diverse operazioni e quindi c'è un vettore di segnali di controllo per specificare l'operazione da effettuare (**ALU control**), inoltre in alcuni casi è necessario fare degli shift e quindi dopo la ALU c'è uno **shifter**, anch'esso con dei segnali di controllo. Per implementare la logica dei salti condizionati, la ALU fornisce dei bit relativi al risultato dell'ultima operazione (**N** e **Z**), i quali saranno controllati al colpo di clock successivo mediante una logica combinatoria (**high bit**); essendo controllati al colpo di clock successivo, è necessario salvare questi due flag in dei flip-flop. Finora abbiamo descritto poco più dell'unità operativa: l'unità di controllo è realizzata in logica microprogrammata, dunque le sequenze di segnali di abilitazione sono salvati in una memoria **ROM**, e sono sequenze a 36 bit. Di questi, 9 sono per l'indirizzo successivo nella ROM, che quindi può contenere un massimo di 512 sequenze, 3 sono segnali di controllo interno relativi a salti condizionati / incondizionati (le istruzioni relative a salti condizionati sono tutte implementate con il bit più significativo alto, per questo motivo la logica combinatoria relativa ai flag della ALU si chiama **high bit**; quindi si ha un massimo di 256 sequenze per le istruzioni relative ai salti condizionati e 256 sequenze per tutte le altre istruzioni, comprese quelle che istruiscono la logica combinatoria riguardo al controllo dei flag della ALU ovvero quelle per l'inizio del salto condizionato), 8 sono segnali di controllo dello shifter e della ALU, 9 sono segnali di selezione dei registri in cui scrivere il contenuto del bus C, 3 sono segnali di controllo della memoria e 4 sono segnali che vanno in ingresso al decoder per selezionare da quale registro leggere il prossimo valore del bus B. Riscontriamo quanto detto nella seguente figura, la quale rappresenta la **control word**:



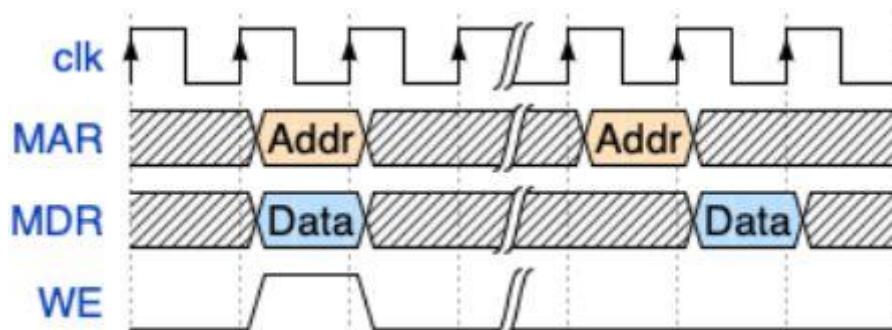
Per quanto riguarda le **funzioni della ALU**:

F₀	F₁	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	\bar{A}
1	0	1	1	0	0	\bar{B}
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B - A
1	1	0	1	1	0	B - 1
1	1	1	0	1	1	-A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	-1

A questo punto non dovremmo avere problemi a visualizzare l'**architettura complessiva**:



L'unica cosa che ci manca è la tempificazione delle operazioni di I/O, importante per comprendere come le istruzioni assembly sono programmate in termini di micro-istruzioni nella ROM. L'ipotesi che si fa è di avere un cache hit di 100%, ovvero ogni volta che si effettua un'operazione di lettura e scrittura, si ha a disposizione il risultato nel colpo di clock successivo, quindi si ha un protocollo sincrono. L'indirizzo ed il dato da scrivere devono essere disponibili in corrispondenza dello stesso colpo di clock: quando si schedula l'operazione di scrittura al tempo T , al tempo $T + \Delta$ cioè in corrispondenza del colpo di clock successivo si ha il segnale di *write enable* (WE) per la memoria e quindi si scrive il contenuto del registro MDR all'indirizzo specificato in MAR; l'operazione sarà conclusa nel colpo di clock ancora successivo. In fase di lettura, similmente si schedula l'operazione di lettura per il colpo di clock successivo, non è necessario un *read enable* per la memoria (notare la differenza tra l'*enable* interno alla CPU e l'*enable* per la memoria), e l'operazione sarà conclusa nel colpo di clock ancora successivo, in cui il valore presente nell'indirizzo specificato nel registro MAR risulterà disponibile nel registro MDR.



8.3 Analisi in simulazione

Al fine di analizzare il flusso di esecuzione scaturito dalle istruzioni **BIPUSH**, **IADD** e **ISTORE** è stato considerato il programma, program.ajvm, scritto in linguaggio assembly specifico per il processore:

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

Questo programma dopo essere stato assemblato viene posto nella memoria RAM dalla quale verrà prelevato, istruzione per istruzione dai registri PC e MBR del processore.

Ogni istruzione è codificata come una stringa di bit che specificano l'indirizzo della prima microistruzione presente all'interno della control store dell'unità di controllo.

Nel caso considerato le micro-procedure relative alle istruzioni **BIPUSH**, **IADD** e **ISTORE** sono le seguenti:

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

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

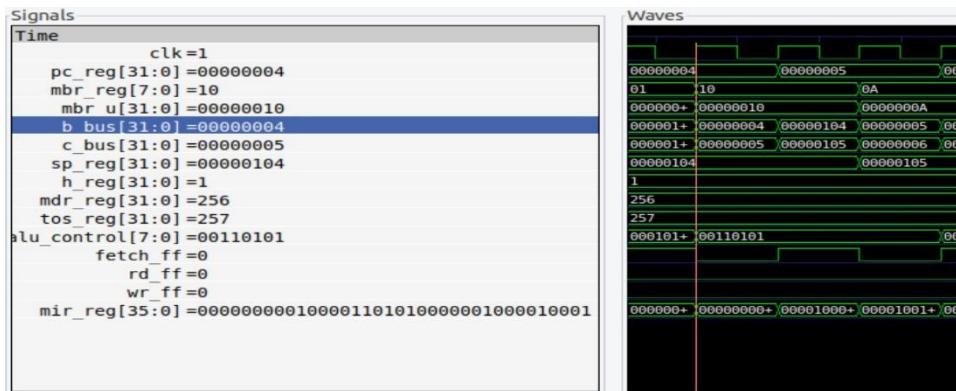
istore = 0x36:
    H=LV
    MAR = MBRU + H
istore_cont:
    MDR=TOS;wr
    SP = MAR = SP - 1; rd
    PC = PC + 1; fetch
    TOS = MDR; goto main
```

Si consideri il flusso di esecuzione della **BIPUSH**; al tempo 225 ns della simulazione, viene eseguita la microistruzione relativa al main:

main:

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

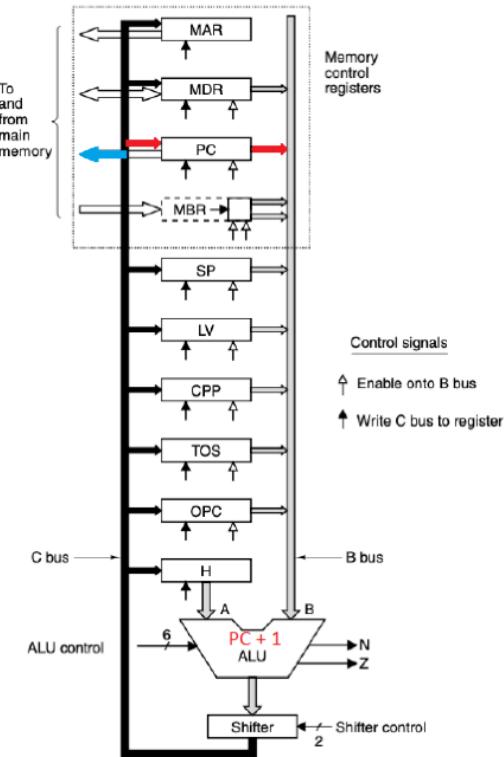
Questa ha lo scopo di fare il fetch dell'istruzione, in questo caso saltando alla prima micro-istruzione di **BIPUSH**. Lo si può vedere, in simulazione, dal valore di MBR pari a 00010000 (mbr_reg[7:0] = 10 in figura, essendo rappresentato in esadecimale), coincidente con l'indirizzo della control store nel quale è memorizzata la prima micro-istruzione di **BIPUSH**.



Dalla figura è possibile osservare i valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 110101, coincidenti con l'operazione B + 1;
- Per il bus C i nove bit di controllo 000000100, coincidenti con il registro PC;
- Per le operazioni in memoria i tre bit di controllo 001, coincidenti con una fetch;
- Per i bit codificati del bus B i quattro bit di controllo 0001, coincidenti con il registro PC.

Quindi, viene data l'abilitazione al registro PC per scrivere sul bus B il proprio contenuto, questo arriva alla ALU, la quale incrementa di 1 il suo valore e lo trasmette al bus C. Mediante il bus C quindi si sovrascrive il valore del registro PC, che al termine dell'operazione avrà il valore aggiornato. Viene fatta la fetch in memoria utilizzando il valore aggiornato di PC e, dopo due colpi di clock rispetto a quando viene schedulata la fetch, il registro MBR conterrà l'indirizzo prelevato.



8.3.1 Analisi Bipush

All’istante 235ns è letta l’informazione dall’MBR che punta alla locazione di memoria della control store nella quale si trova la prima istruzione, o meglio il primo byte, della prima operazione di BIPUSH.

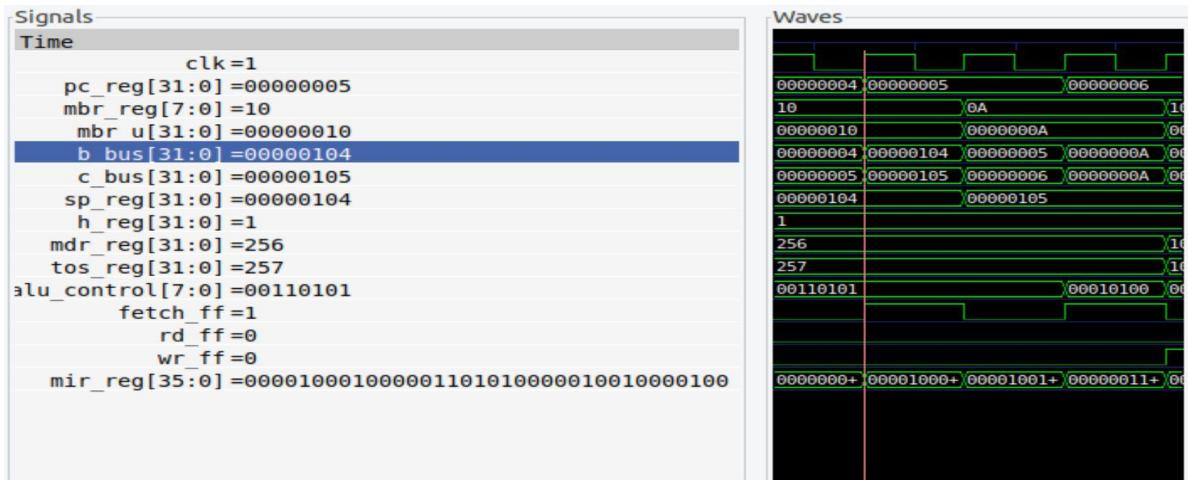
L’obiettivo dell’operazione è caricare nello stack un dato valore passato nell’istruzione stessa e quindi caricato nel memory byte register (MBR), nel caso in esame si fa una push di un numero esadecimale A che in decimale è reso come 10. Il codice dell’operazione in microistruzioni è:

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

Vediamo cosa accade per ogni microistruzione.

- SP = MAR = SP +1

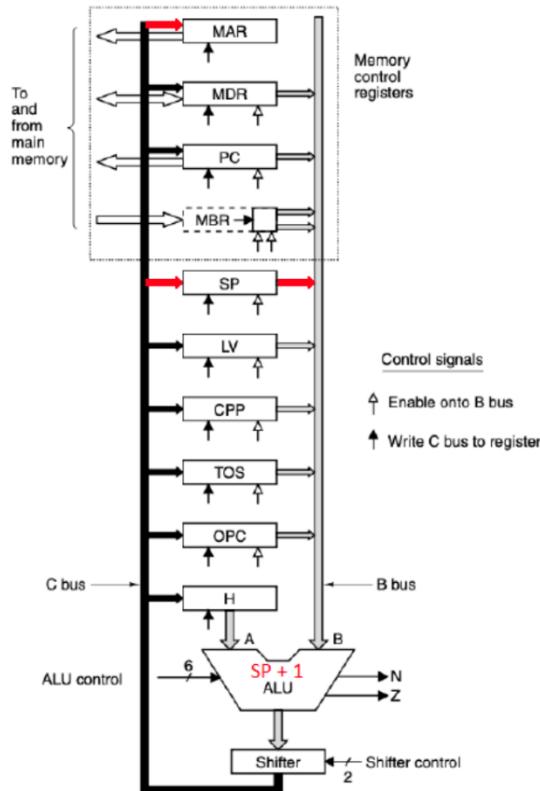
La prima microistruzione della BIPUSH avviene al marker 235ns e prevede di posizionare nel registro SP e MAR il valore di SP + 1, questo perché dovendo inserire un nuovo valore dello stack è necessario incrementare il puntatore alla testa dello stack, dovendo poi posizionare l’operando in memoria inizializziamo anche il registro MAR di modo che sappiamo già dove dover scrivere il dato che per ora non abbiamo ancora caricato nel MDR. Analizziamo con il tool gtkwave cosa accade nel nostro processore.



Dalla figura è possibile osservare i valori dei bit della control word prelevata dalla control store:

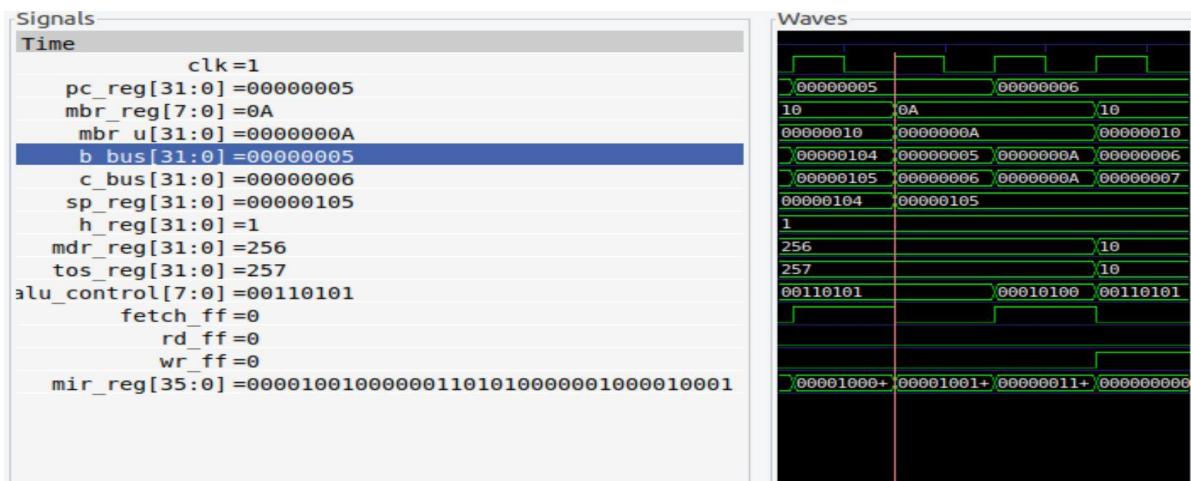
- Per la ALU i sei bit di controllo 110101, coincidenti con l'operazione B + 1;
- Per il bus C i nove bit di controllo 000001001, coincidenti con i registri SP e MAR;
- Per le operazioni in memoria i tre bit di controllo 000, ovvero nessuna operazione sulla memoria deve essere eseguita;
- Per i bit codificati del bus B i quattro bit di controllo 0100, coincidenti con il registro SP;
- Per il campo next address leggiamo i bit 000010001 ovvero 17 in decimale.

Analizzando le parole di controllo ritroviamo che il PC dovrà puntare alla prossima microistruzione, dunque, deve essere incrementato di 1 e passa da 16 a 17. Sul bus B abilitiamo la lettura del dato presente in SP che poi passa alla ALU, la quale ne incrementerà il valore di uno: aggiungendo un nuovo dato, il puntatore alla testa dello stack deve essere incrementato, in effetti stiamo facendo una push. Infine, il dato elaborato passa nei registri abilitati in scrittura dal bus C: SP, per aggiornare il puntatore alla testa dello stack, e MAR, dato che dovremo comunicare con la memoria scrivendo il nuovo valore di testa.



b. $PC = PC + 1$; fetch

All'istante 245 ns è eseguita la seconda micro-istruzione della BIPUSH che prevede il semplice aggiornamento del PC con lo scheduling del segnale di fetch. Analizziamo con il tool gtkwave cosa accade nel nostro processore.

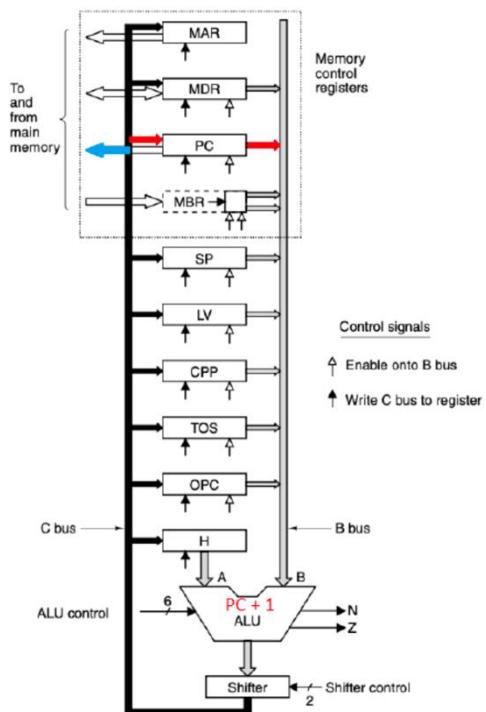


Dalla figura è possibile osservare i valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 110101, coincidenti con l'operazione B + 1;
- Per il bus C i nove bit di controllo 000000100, coincidenti con il registro PC;

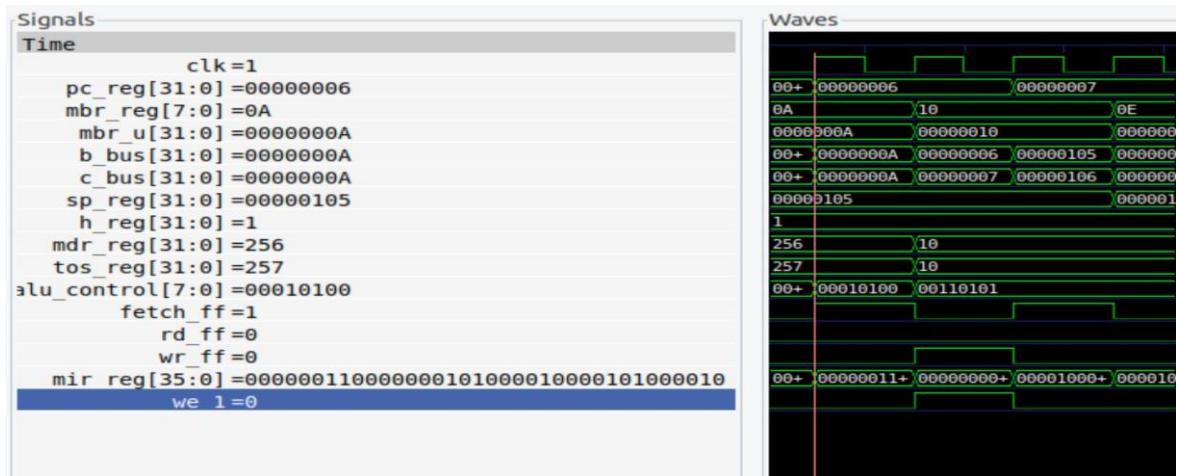
- Per le operazioni in memoria i tre bit di controllo 001, è schedulata l'operazione di fetch;
- Per i bit codificati del bus B i quattro bit di controllo 0001, coincidenti con il registro PC;
- Per il campo next address leggiamo i bit 000010010 ovvero 18 in decimale.

Analizzando le parole di controllo ritroviamo che il PC dovrà puntare alla prossima microistruzione, dunque, deve essere incrementato di 1 e passa da 17 a 18. Sul bus B abilitiamo la lettura del dato in PC che poi passa alla ALU che incrementerà il valore di uno. Il dato elaborato entra nel registro abilitato dal bus C ovvero PC. Da notare il flag di fetch alto; l'azione della fetch non serve per segnalare il caricamento del dato da salvare in memoria nel registro MBR, bensì ad aggiornare il registro inserendo la prossima operazione alla quale il main dovrà puntare. L'operazione di fetch dopo essere schedulata impiega due colpi di clock ad essere eseguita, nel senso che il risultato della fetch sarà disponibile nel registro MBR solo due colpi di clock dopo lo scheduling della fetch: ciò comporta da parte del processore un anticipo dell'operazione. Infatti, nel main, quando si fa goto (MBR) non si salta all'indirizzo relativo alla fetch appena schedulata nel main, perché tale indirizzo non è stato ancora letto, ma si salta all'indirizzo che era precedentemente presente in MBR; gli "indirizzi" di cui parliamo sono in realtà codici operativi, e sono usati per indirizzare la ROM. Un'istruzione in generale è composta da un codice operativo e da operandi: se con ciascuna fetch si preleva un byte, per le istruzioni che prevedono operandi, in generale, ci aspettiamo una o più fetch all'interno della relativa procedura MAL, per garantire che quando si ritorna al main si abbia in MBR il codice operativo della prossima istruzione. Tornando alla BIPUSH, mentre l'operazione di fetch del prossimo codice operativo è schedulata, viene letto il dato da salvare in memoria, cioè l'operando, che era stato precedentemente caricato nel registro MBR per mezzo della prima fetch fatta nel main al marker 225.



c. MDR = TOS = MBR; wr; goto main

All'istante 255 ns è eseguita la terza ed ultima microistruzione della BIPUSH, questa prevede di aggiornare il registro TOS con il nuovo dato salvato nel registro MBR e di posizionare il dato anche nell'MDR, schedulando il segnale di write in modo che la push sullo stack sia salvata in memoria. Infine, vi è il ritorno al main per eseguire le prossime operazioni; ricordiamo che nel programma considerato andremo ad eseguire una seconda BIPUSH per caricare il secondo operando. Analizziamo con il tool gtkwave cosa accade nel nostro processore.

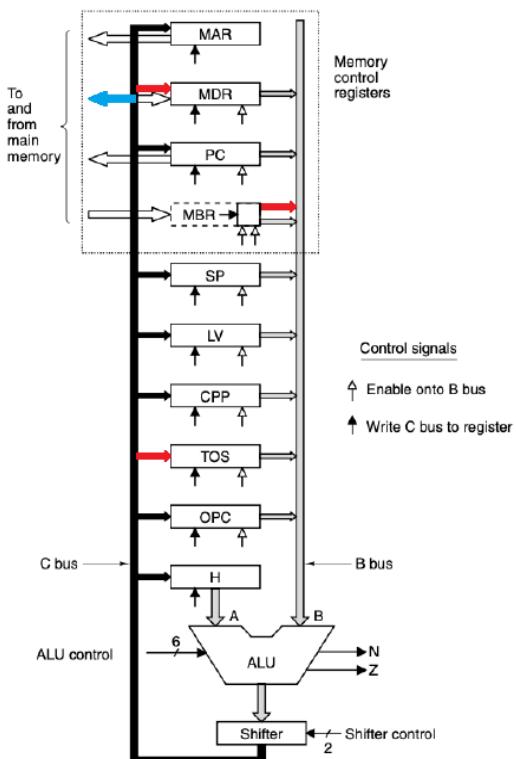


Dalla figura è possibile osservare i valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 010100, coincidenti con l'operazione B, ovvero il dato passa immutato;
- Per il bus C i nove bit di controllo 001000010, coincidenti con i registri TOS e MDR;
- Per le operazioni in memoria i tre bit di controllo 100, è schedulata l'operazione di write;
- Per i bit codificati del bus B i quattro bit di controllo 0010, coincidenti con il registro MBR, notando che si punta al registro MBR e non MBRU, questo perché l'operando può anche essere un valore negativo;
- Per il campo next address leggiamo i bit 000000110 ovvero 6 in decimale.

Analizzando le parole di controllo ritroviamo che il PC dovrà puntare alla microistruzione main, dunque, di fatti il campo next address punta alla locazione di memoria 6 dove è salvata l'informazione relativa al main. Sul bus B abilitiamo la lettura del dato presente in MBR che poi passa per la ALU senza subire variazioni ed è scritto nei registri TOS e MDR, abilitati in scrittura dal bus C, inoltre notiamo che il flag di scrittura in memoria è schedulato alto. Cerchiamo di capire nel dettaglio cosa accade: innanzitutto, dopo un colpo di clock dall'ultima microistruzione della BIPUSH si ritorna al main e vediamo nel mentre i valori aggiornati sia nel registro TOS, che ora non deve più essere modificato, sia in MDR, che contiene il valore

da scrivere in memoria. Notiamo inoltre che all’istante 265 ns, ovvero dopo un colpo di clock dall’ultima microistruzione della BIPUSH, è rispettato il protocollo di scrittura in memoria avendo un indirizzo nel registro MAR, ovvero l’indirizzo della nuova testa dello stack, un valore caricato nel registro MDR (in questo caso il valore A, che coincide con TOS) e il flag write enable della memoria alto (nella figura we_1). Dunque, dopo un **ulteriore** colpo di clock, ovvero all’istante 275 ns, avremo ultimato le operazioni previste dalla BIPUSH avendo anche aggiornato la memoria. Nel mentre, il nostro programma è nel main, ed ha puntato alla prossima istruzione da eseguire, che nel programma in esame è la seconda BIPUSH, il cui codice operativo è stato salvato nel registro MBR due colpi di clock dopo la fetch prevista dalla BIPUSH.



8.3.2 Analisi IADD

Non è stato riportato il comportamento nel dettaglio della seconda BIPUSH, essendo analogo al precedente, con l’unica differenza che l’operando caricato in testa allo stack in questo caso è 14 in decimale (0xE in esadecimale).

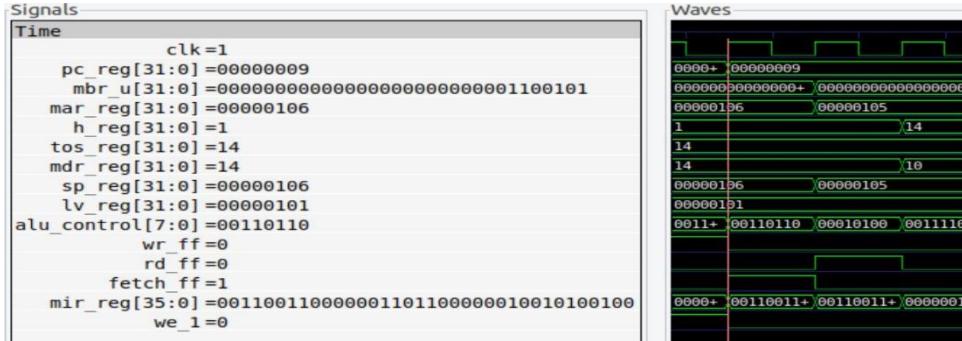
Al marker 315 inizia il flusso di esecuzione della IADD. L’istruzione IADD fa la pop dei due operandi presenti in testa allo stack, li somma e fa il push del risultato in testa allo stack. Volendo essere più precisi, per motivi di performance effettua una sola pop: il secondo operando lo legge solo, e poi lo sovrascrive con la somma tra i due operandi. Nel caso in esame, preleva dallo stack 10 in decimale (0xA in esadecimale), 14 in decimale (0xE in esadecimale), li somma e il risultato, 24 in decimale (0x18 in esadecimale), viene posto in cima allo stack. Il codice dell’operazione in microistruzioni è:

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

Si considerano una alla volta le microistruzioni.

a. $\text{MAR} = \text{SP} = \text{SP} - 1; \text{rd};$

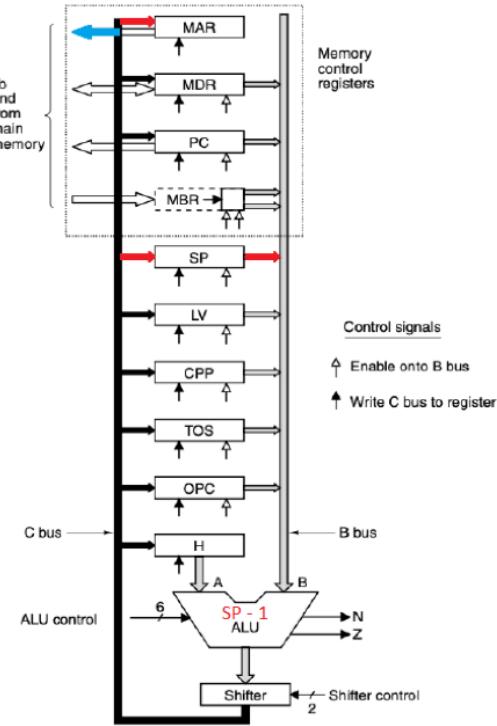
La prima microistruzione decremente lo Stack-Pointer per poter puntare al primo operando dell'operazione (10 in decimale), infatti viene puntata la posizione immediatamente inferiore alla testa dello stack.



I valori dei bit della control word prelevata dalla control store:

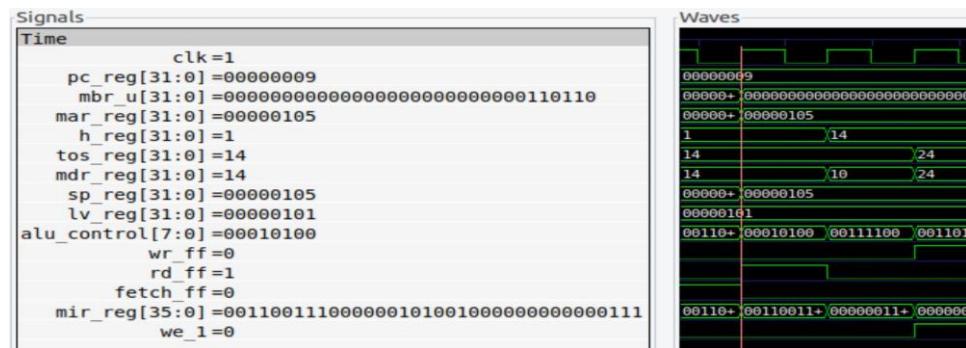
- Per la ALU i sei bit di controllo 110110, coincidenti con l'operazione B - 1;
- Per il bus C i nove bit di controllo 000001001, coincidenti con i registri MAR e SP;
- Per le operazioni in memoria i tre bit di controllo 010, è schedulata l'operazione di read;
- Per i bit codificati del bus B i quattro bit di controllo 0100, coincidenti con il registro SP.

Il registro SP viene abilitato a scrivere il proprio valore sul bus B. Il valore entra nella ALU, viene decrementato di uno e viene trasmesso al bus C. Dal bus C viene letto dai registri SP e MAR. Al termine dell'operazione MAR e SP saranno aggiornati con il valore SP - 1. Al colpo di clock successivo viene schedulata l'operazione read dalla memoria e dopo un ulteriore colpo di clock il valore letto dalla memoria è visibile in MDR.



b. $H = TOS$

Al marker 325 ns viene eseguita la seconda microistruzione, la quale memorizza il valore contenuto in TOS nel registro H. Il valore memorizzato in H coincide con il secondo operando (14 in decimale) poiché TOS contiene il valore della precedente testa dello stack.

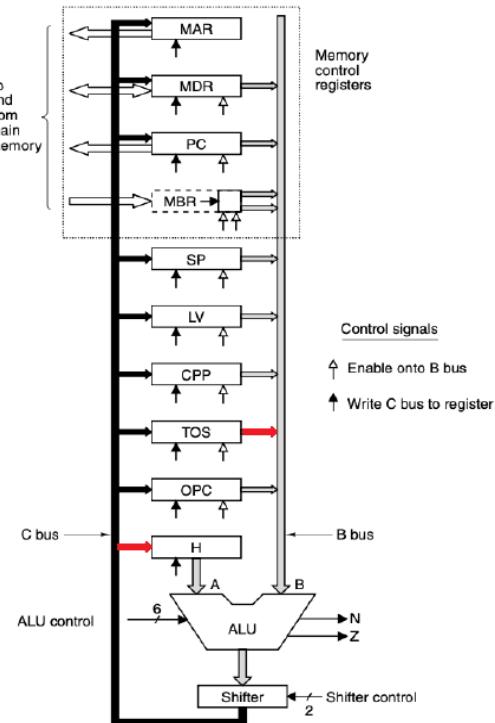


I valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 010100, coincidenti con l'operazione B;
- Per il bus C i nove bit di controllo 100000000, coincidenti con il registro H;
- Per le operazioni in memoria i tre bit di controllo 000, coincidenti con nessuna operazione in memoria;
- Per i bit codificati del bus B i quattro bit di controllo 0111, coincidenti con il registro TOS.

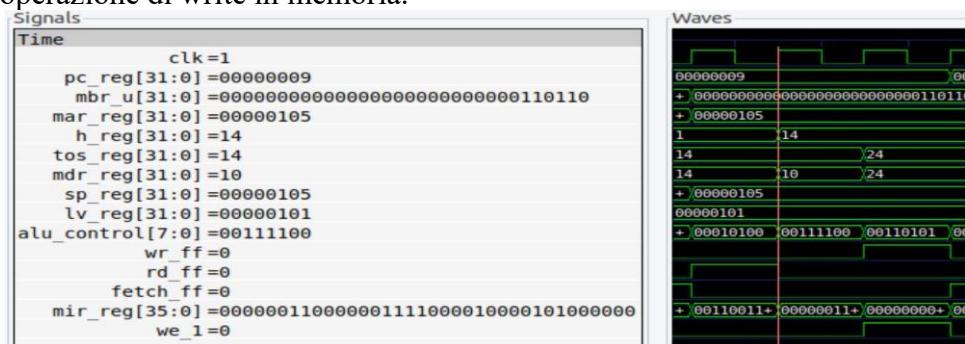
Quindi, viene abilitato il registro TOS in scrittura sul bus B, il valore da esso scritto passa immutato nell'ALU e arriva al bus C, dal quale viene letto dal registro H.

In figura è possibile osservare che in questo colpo di clock, oltre ad avere il secondo operando nel registro H (0x14 in esadecimale), risultato dall'esecuzione della seconda micro-istruzione appena analizzata, sono visibili anche gli effetti della prima micro-istruzione, infatti è possibile osservare che nel registro MDR è memorizzato il primo operando (0xA in esadecimale). Infatti, in questo colpo di clock sono avvenuti parallelamente la lettura dalla memoria e lettura dal bus C.



$$C. \quad MDR = TOS = MDR + H; \quad wr$$

Al marker 335 ns viene eseguita la terza microistruzione, questa effettua la somma tra i due operandi e scrive il risultato nel registro TOS e nel registro MDR per poterlo scrivere in memoria, abilitando a tale scopo l'operazione di write in memoria.



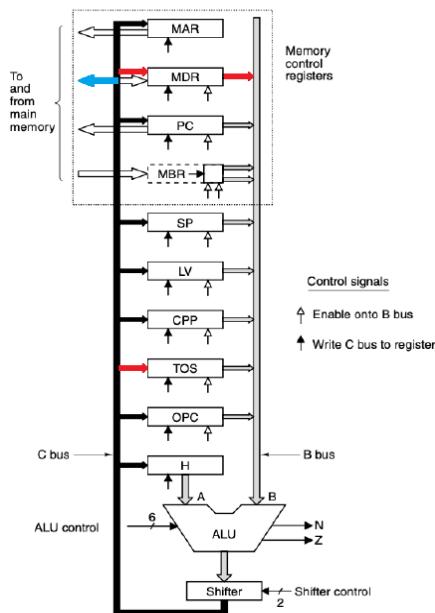
I valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 111100, coincidenti con l'operazione A + B;
- Per il bus C i nove bit di controllo 001000010, coincidenti con i registri TOS e MDR;

- Per le operazioni in memoria i tre bit di controllo 100, coincidenti con l'operazione di scrittura in memoria;
- Per i bit codificati del bus B i quattro bit di controllo 0000, coincidenti con il registro MDR.

Il registro MDR è abilitato a scrivere sul bus B e il valore da esso contenuto entra come secondo operando dell'ALU. L'ALU somma il contenuto del registro H con il valore presente sul bus B e trasferisce il risultato sul bus C, dal quale viene letto dai registri TOS e MDR.

Viene abilitata l'operazione di scrittura in memoria, quindi dopo un colpo di clock essa viene effettivamente schedulata e dopo un ulteriore colpo di clock il valore di MDR viene scritto all'indirizzo contenuto nel registro MAR in memoria, ovvero in testa allo stack.



8.3.3 Analisi ISTORE

Al marker 355 ns inizia il flusso di esecuzione della ISTORE. L'istruzione ISTORE memorizza la testa dello stack all'indirizzo passato dall'operando dell'istruzione stessa, facendo pop dallo stack. Nel caso in esame, si memorizza la testa dello stack, contenente il risultato della somma, 24 in decimale (0x18 in esadecimale), all'indirizzo passato dalla ISTORE, ‘a’. Il codice dell'operazione in microistruzioni è:

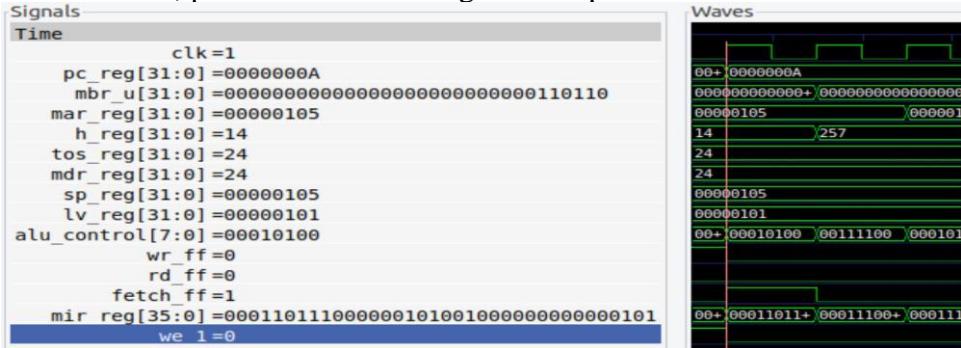
```

istore = 0x36:
    H=LV
    MAR = MBRU + H
istore_cont:
    MDR = TOS; wr
    SP = MAR = SP - 1; rd
    PC = PC + 1; fetch
    TOS = MDR; goto main
  
```

Si considerano una alla volta le microistruzioni.

a. $H = LV$

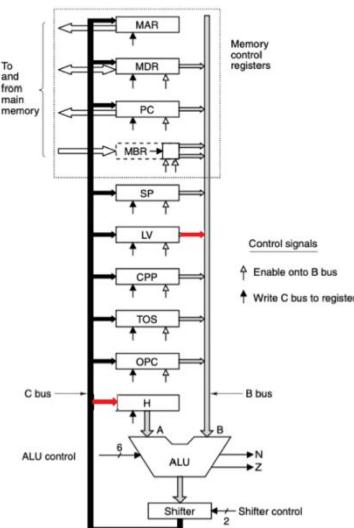
La prima microistruzione memorizza il valore contenuto nel registro LV nel registro H. Questa operazione viene effettuata allo scopo di “comporre” l’indirizzo assoluto in memoria dato dalla somma dell’indirizzo base contenuto in LV e lo scostamento, contenuto in MBRU. Al termine della prima microistruzione il primo operando per la composizione dell’indirizzo è quindi pronto ad essere sommato dall’ALU, poiché si trova nel registro tampone.



I valori dei bit della control word prelevata dalla control store:

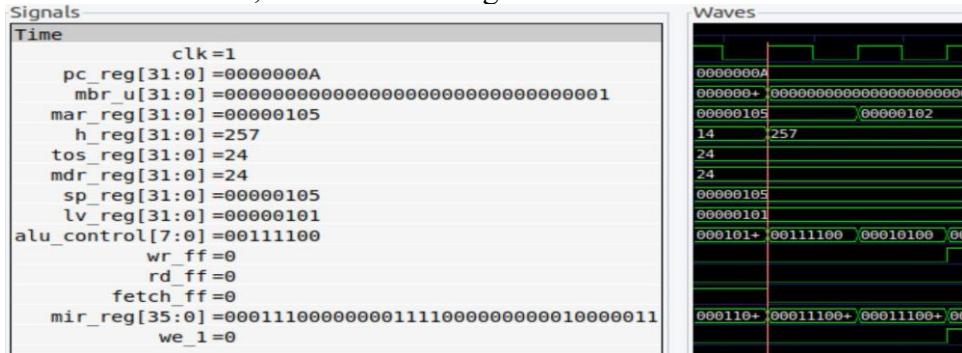
- Per la ALU i sei bit di controllo 010100, coincidenti con l’operazione B;
- Per il bus C i nove bit di controllo 100000000, coincidenti con il registro H;
- Per le operazioni in memoria i tre bit di controllo 000, coincidenti con nessuna operazione in memoria;
- Per i bit codificati del bus B i quattro bit di controllo 0101, coincidenti con il registro LV.

Il registro LV è abilitato a scrivere sul bus B, il suo contenuto, attraverso il bus B, arriva all’ALU che lo fa passare invariato verso il bus C e dal bus C viene letto dal registro tampone H.



b. $MAR = MBRU + H$

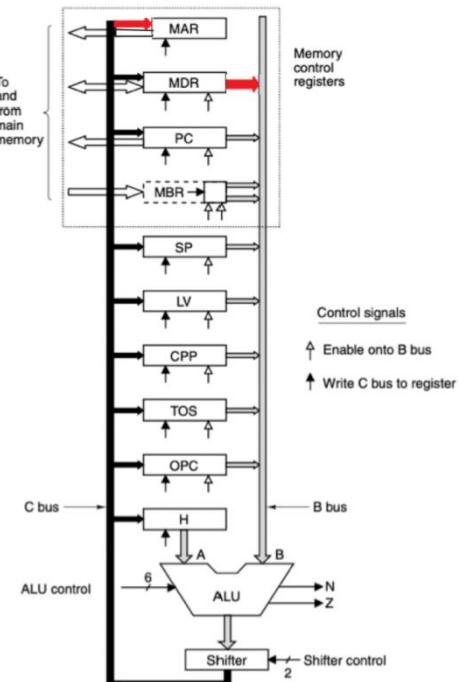
Al marker 365 ns viene eseguita la seconda micro-istruzione, questa effettua la somma dei registri MBRU e H e pone il risultato ottenuto, coincidente con l'indirizzo di memoria assoluto nel quale memorizzare la testa dello stack, all'interno del registro MAR.



I valori dei bit della control word prelevata dalla control store:

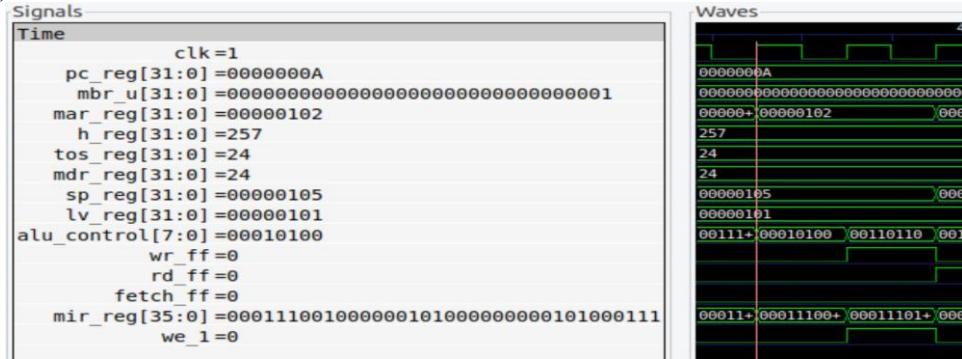
- Per la ALU i sei bit di controllo 111100, coincidenti con l'operazione A + B;
- Per il bus C i nove bit di controllo 000000001, coincidente con il registro MAR;
- Per le operazioni in memoria i tre bit di controllo 000, coincidenti con nessuna operazione in memoria;
- Per i bit codificati del bus B i quattro bit di controllo 0011, coincidenti con il registro MBRU, da non confondere con il registro MBR. Il registro MBRU contiene il valore di MBR esteso senza segno in modo tale da essere utilizzato per il calcolo degli indirizzi.

Viene abilitato il registro MBRU a scrivere sul bus B, il suo contenuto viene gestito dalla ALU come secondo operando da sommare al primo operando contenuto nel registro H. Il risultato viene trasferito sul bus C e viene letto dal registro MAR.



c. MDR = TOS; wr

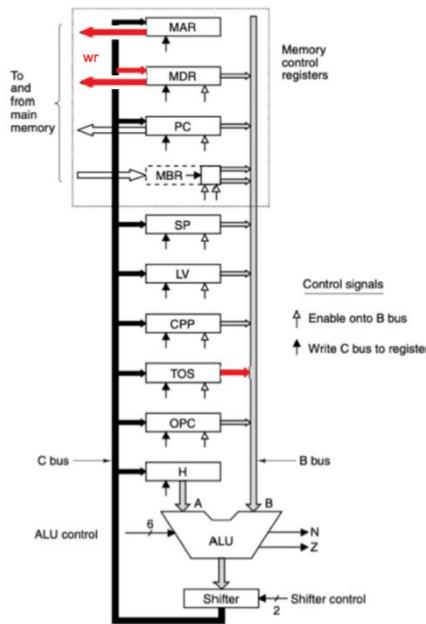
Al marker 375 ns viene eseguita la terza microistruzione, questa memorizza il valore di TOS all'interno del registro MDR per poi scriverlo in memoria. Il registro TOS contiene il valore in testa allo stack, nel caso in esame è il risultato della somma della IADD precedente (0x18 in esadecimale).



I valori dei bit della control word prelevata dalla control store:

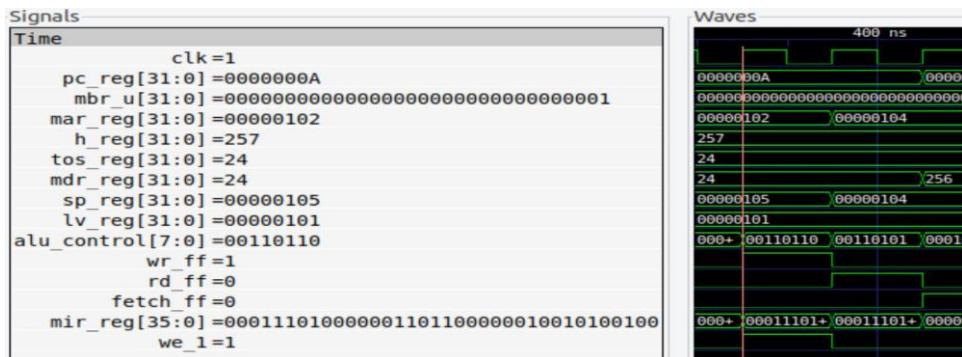
- Per la ALU i sei bit di controllo 010100, coincidenti con l'operazione B;
- Per il bus C i nove bit di controllo 000000010, coincidente con il registro MDR;
- Per le operazioni in memoria i tre bit di controllo 100, coincidenti con l'operazione di write in memoria;
- Per i bit codificati del bus B i quattro bit di controllo 0111, coincidenti con il registro TOS.

Il registro TOS è abilitato a scrivere sul bus B, il suo contenuto arriva all'ALU che lo fa passare immutato verso il bus C dal quale viene letto dal registro MDR. Viene abilitata l'operazione di scrittura in memoria del contenuto di MDR all'indirizzo indicato da MAR. Al colpo di clock successivo viene schedulata l'operazione di scrittura in memoria e aspettando un ulteriore colpo di clock il contenuto di MDR è effettivamente presente in memoria.



$$d. \quad SP = MAR = SP - 1; \quad rd$$

Al marker 385 viene eseguita la quarta microistruzione. Allo scopo di effettuare una pop dallo stack viene decrementato il valore del puntatore alla testa dello stack e viene memorizzato anche nel registro MAR per poi effettuare la lettura a quell'indirizzo di memoria.

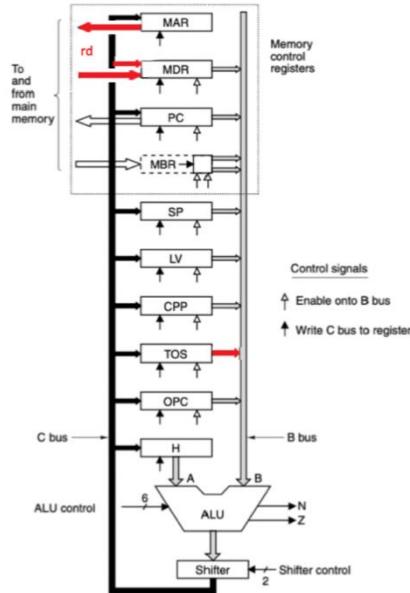


I valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 110110, coincidenti con l'operazione B - 1;
- Per il bus C i nove bit di controllo 000001001, coincidenti con i registri SP e MAR;
- Per le operazioni in memoria i tre bit di controllo 010, coincidenti con l'operazione di read in memoria;
- Per i bit codificati del bus B i quattro bit di controllo 0100, coincidenti con il registro SP.

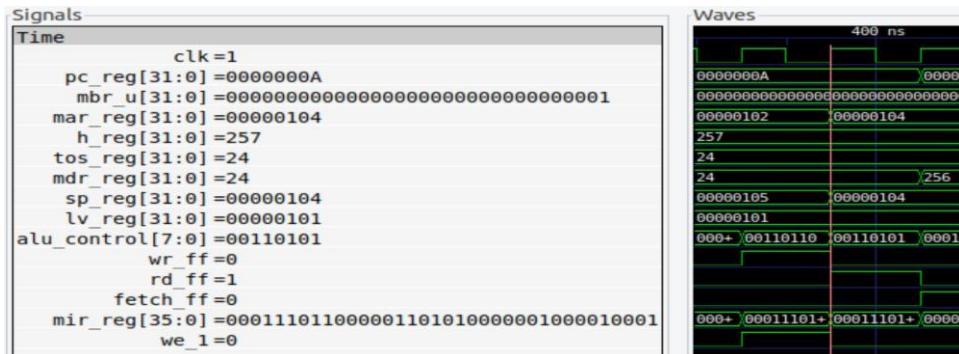
Il registro SP è abilitato a scrivere sul bus B il suo contenuto. Quest'ultimo, arrivato all'ALU, viene decrementato di uno e viene trasferito sul bus C, dal quale viene letto dai registri SP e MAR. Viene abilitata la read dalla memoria, quindi dopo un colpo di clock questa viene schedulata e dopo un

ulteriore colpo di clock, ovvero all'istante 405 ns, il contenuto della memoria puntato da MAR verrà memorizzato nel registro MDR.



e. $PC = PC + 1$; fetch

All'istante 395 ns è eseguita la quinta micro-istruzione della ISTORE che prevede il semplice aggiornamento del PC con lo scheduling del segnale di fetch. Questa viene effettuata con lo scopo di prelevare la prossima istruzione da eseguire poiché la fetch precedente effettuata nel main è stata utilizzata per prelevare l'operando 'a' della `istore` che indica l'indirizzo relativo utilizzato.

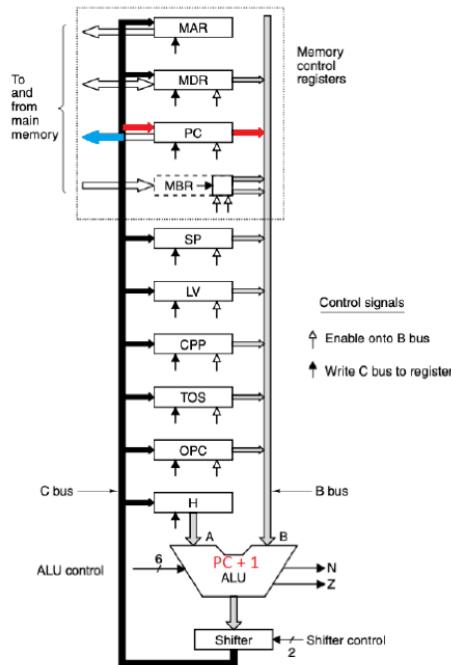


Dalla figura è possibile osservare i valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 110101, coincidenti con l'operazione B + 1;
- Per il bus C i nove bit di controllo 000000100, coincidenti con il registro PC;
- Per le operazioni in memoria i tre bit di controllo 001, è schedulata l'operazione di fetch;
- Per i bit codificati del bus B i quattro bit di controllo 0001, coincidenti con il registro PC;

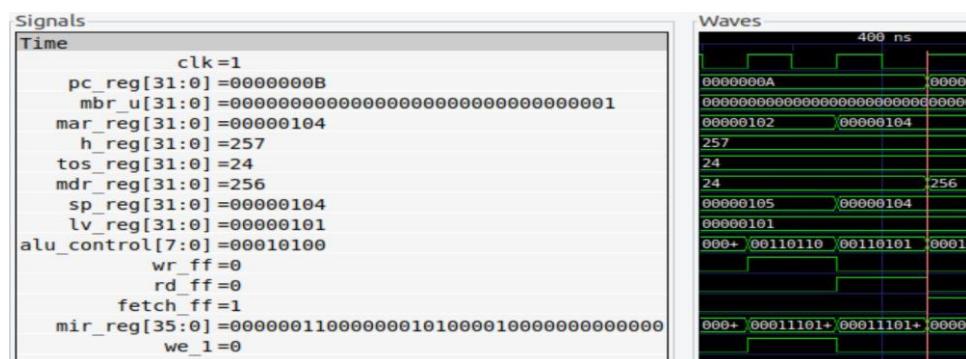
- Per il campo next address leggiamo i bit 000111011 ovvero 59 in decimale.

Il registro PC è abilitato a scrivere sul bus B e il suo contenuto arriva in ingresso all'ALU che lo incrementa di uno e lo passa al bus C dal quale viene letto dal registro PC. Viene attivata l'operazione di fetch con la memoria. Dopo un colpo di clock la fetch viene schedulata e dopo un ulteriore colpo di clock il contenuto della memoria è presente nel registro MBR.



f. $TOS = MDR$; goto main

All'istante 405 ns viene schedulata l'ultima microistruzione della ISTORE. In questo colpo di clock sono visibili gli effetti dell'operazione di lettura effettuata all'istante 385 ns (quarta microistruzione). Quindi il contenuto appena letto dalla memoria, presente in MDR viene memorizzato nel registro TOS. Al termine di questa operazione si conclude anche la parte di pop che coinvolge lettura dalla memoria e viene memorizzato quindi il valore della testa dello stack nel registro TOS.

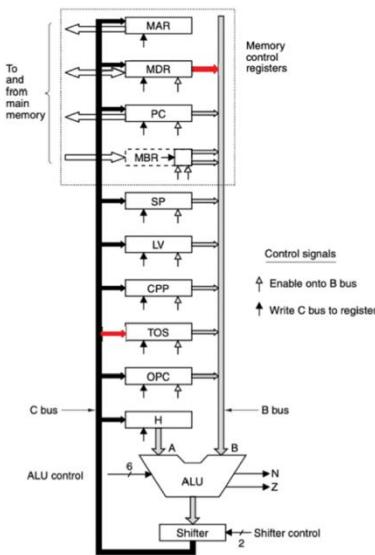


Dalla figura è possibile osservare i valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 010100, coincidenti con l'operazione B;

- Per il bus C i nove bit di controllo 001000000, coincidenti con il registro TOS;
- Per le operazioni in memoria i tre bit di controllo 000, coincidente con nessuna operazione in memoria;
- Per i bit codificati del bus B i quattro bit di controllo 0000, coincidenti con il registro MDR;

Il registro MDR è abilitato a scrivere sul bus B, il suo contenuto arriva all'ALU e viene trasferito, invariato, sul bus C dal quale viene letto dal registro TOS.



8.4 Modifica di un'istruzione e relativa analisi in simulazione

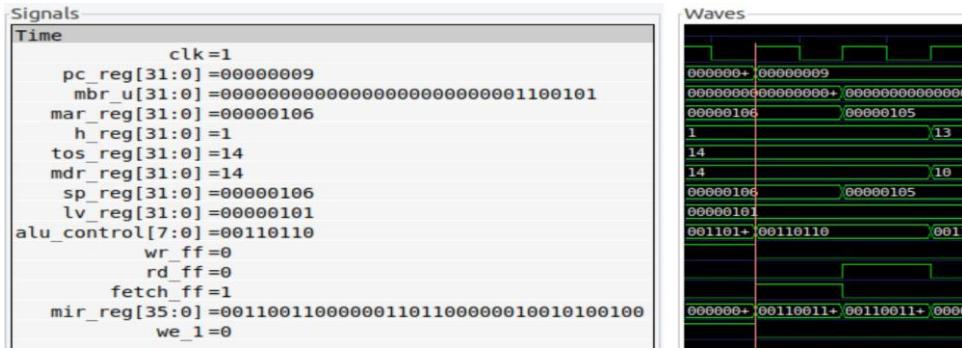
Nel paragrafo analizziamo l'operazione della IADD ponendo un parallelismo diretto tra una versione di IADD modificata e la versione non alterata. In primo luogo, presentiamo i due codici in termini di microistruzioni che descriveremo:

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

Alla destra troviamo il codice della iadd modificata ed a sinistra della iadd non modificata, le due versioni ritornano comunque il risultato corretto dell'operazione di somma, in quanto la versione modificata prevede un'operazione di sottrazione per uno ed un'operazione di somma per uno. Analizziamo istruzione per istruzione cosa accade e dove avviene la modifica; le due varianti della iadd sono state usate entrambe nel programma ai paragrafi precedenti.

a. $\text{MAR} = \text{SP} = \text{SP} - 1; \text{rd};$

La prima microistruzione decremente lo Stack-Pointer per poter puntare al primo operando dell'operazione (10 in decimale), infatti viene puntata la posizione immediatamente inferiore alla testa dello stack.



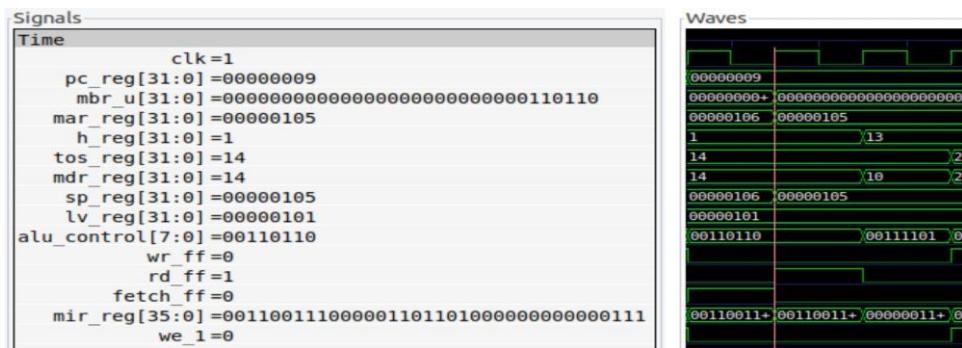
I valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 110110, coincidenti con l'operazione B - 1;
- Per il bus C i nove bit di controllo 000001001, coincidenti con i registri MAR e SP;
- Per le operazioni in memoria i tre bit di controllo 010, è schedulata l'operazione di read;
- Per i bit codificati del bus B i quattro bit di controllo 0100, coincidenti con il registro SP.

Il registro SP viene abilitato a scrivere il proprio valore sul bus B. Il valore entra nella ALU, viene decrementato di uno e viene trasmesso al bus C. Dal bus C viene letto dai registri SP e MAR. Al termine dell'operazione MAR e SP saranno aggiornati con il valore SP – 1. Al colpo di clock successivo viene schedulata l'operazione di read dalla memoria e dopo un ulteriore colpo di clock il valore letto dalla memoria è visibile in MDR. Si nota, dunque, che la prima microistruzione delle due versioni della iadd non ha alcuna modifica.

b. $H = TOS - 1$

Al marker 325 ns viene eseguita la seconda microistruzione, la quale memorizza il valore contenuto in TOS nel registro H. Il valore memorizzato in H coincide con il secondo operando (14 in decimale) poiché TOS contiene il valore della precedente testa dello stack.



I valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 110110, coincidenti con l'operazione B - 1;

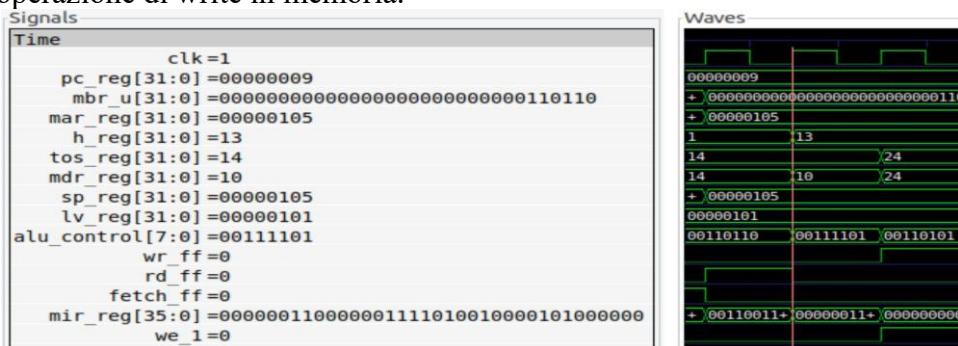
- Per il bus C i nove bit di controllo 100000000, coincidenti con il registro H;
- Per le operazioni in memoria i tre bit di controllo 000, coincidenti con nessuna operazione in memoria;
- Per i bit codificati del bus B i quattro bit di controllo 0111, coincidenti con il registro TOS.

Quindi, viene abilitato il registro TOS in scrittura sul bus B, il valore da esso scritto passa nell'ALU, è decrementato di uno e arriva al bus C, dal quale viene letto dal registro H.

In figura è possibile osservare che in questo colpo di clock, oltre ad avere il secondo operando nel registro H (0x14 in esadecimale), risultato dall'esecuzione della seconda microistruzione appena analizzata, sono visibili anche gli effetti della prima microistruzione, infatti è possibile osservare che nel registro MDR è memorizzato il primo operando (0xA in esadecimale). Infatti, in questo colpo di clock sono avvenuti parallelamente la lettura dalla memoria e lettura dal bus C. Notiamo dunque che le differenze tra la iadd modificata e la versione non modificata sono da riscontrare nella control word dell'alu che in un caso prevede l'operazione di decremento dell'operando B, nel caso standard il passaggio dell'operando B senza variazioni.

c. $MDR = TOS = MDR + H + 1; wr$

Al marker 335 ns viene eseguita la terza microistruzione, questa effettua la somma tra i due operandi e scrive il risultato nel registro TOS e nel registro MDR per poterlo scrivere in memoria, abilitando a tale scopo l'operazione di write in memoria.



I valori dei bit della control word prelevata dalla control store:

- Per la ALU i sei bit di controllo 111101, coincidenti con l'operazione A + B + 1;
- Per il bus C i nove bit di controllo 001000010, coincidenti con i registri TOS e MDR;
- Per le operazioni in memoria i tre bit di controllo 100, coincidenti con l'operazione di scrittura in memoria;
- Per i bit codificati del bus B i quattro bit di controllo 0000, coincidenti con il registro MDR.

Il registro MDR è abilitato a scrivere sul bus B e il valore da esso contenuto entra come secondo operando dell'ALU. L'ALU somma il contenuto del registro H con il valore presente sul bus B, aggiungendo uno, e trasferisce il risultato sul bus C, dal quale viene letto dai registri TOS e MDR.

Viene abilitata l'operazione di scrittura in memoria, quindi dopo un colpo di clock essa viene effettivamente schedulata e dopo un ulteriore colpo di clock il valore di MDR viene scritto all'indirizzo contenuto nel registro MAR in memoria, ovvero in testa allo stack. Notiamo dunque che le differenze tra la iadd modificata e la versione non modificata sono da riscontrare, anche in questo caso, nella control word dell'alu che in prevede l'operazione di somma tra gli operandi A e B nel caso standard, nella versione modificata è alto anche il bit di inc nella control word dell'alu, ciò vuol dire che oltre che sommare il contenuto dei registri A e B si somma anche un +1. Notiamo infine che le due operazioni di iadd restituiscono entrambe lo stesso risultato, dato che la versione modificata decremente il contenuto del registro H (l'operando A dell'alu) di 1, ma aggiunge un'unità alla somma degli operandi totali risultando dunque la somma tra i due operandi A e B senza variazioni. Ma è sempre vero che il risultato è lo stesso? Non è difficile convincersi che, nel caso in cui l'operando che si trova in testa allo stack sia il valore negativo con il valore più alto possibile (quindi il bit più significativo pari ad 1 e poi 31 bit pari a 0), sottraendo 1 si ottiene una condizione di overflow, ottenendo il valore positivo con il valore più alto possibile (in pratica il negato del precedente). Dunque, possono essere introdotte delle situazioni di overflow che normalmente non si avrebbero, ad esempio si può avere una situazione di overflow pur sommando un operando negativo ed un operando positivo: per una somma "normale", è impossibile avere overflow in caso di operandi di segno opposto.

Esercizio 9

9.1 Traccia

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

UART_TAPPO: il componente acquisisce una stringa di 8 bit (fornita attraverso gli switch della board di sviluppo) e la serializza tramite la sezione di trasmissione del dispositivo UART; l'output seriale della UART viene re-inviato in ingresso alla sezione di ricezione dello stesso dispositivo (configurazione a tappo), e il dato deserializzato viene visualizzato sui led della board di sviluppo.

2_UART: il componente acquisisce una stringa di 8 bit (fornita dall'utente tramite gli switch della board di sviluppo), la serializza tramite la sezione di trasmissione di un primo dispositivo UART, la deserializza tramite la sezione di ricezione di un secondo dispositivo UART collegato a valle del primo, e mostra le stringa led della board di sviluppo.

UART_PC (facoltativo): il componente realizza la comunicazione fra la board di sviluppo e un terminale seriale in esecuzione su PC (es. Termite), previa connessione di PC e board tramite dispositivo fisico RS232 (uno degli endpoint di comunicazione è rappresentato dal PC). Il componente deve poter acquisire una stringa di 8 bit che rappresenta un carattere in codifica ASCII (fornita attraverso gli switch della board di sviluppo), ed inviarla tramite il dispositivo UART al terminale in esecuzione sul PC, in cui il carattere viene visualizzato. Allo stesso modo, il componente deve essere in grado di ricevere attraverso lo stesso dispositivo UART (oppure una seconda UART) un carattere trasmesso dal terminale e mostrarlo sui led.

9.2 Richiamo sul funzionamento della periferica seriale

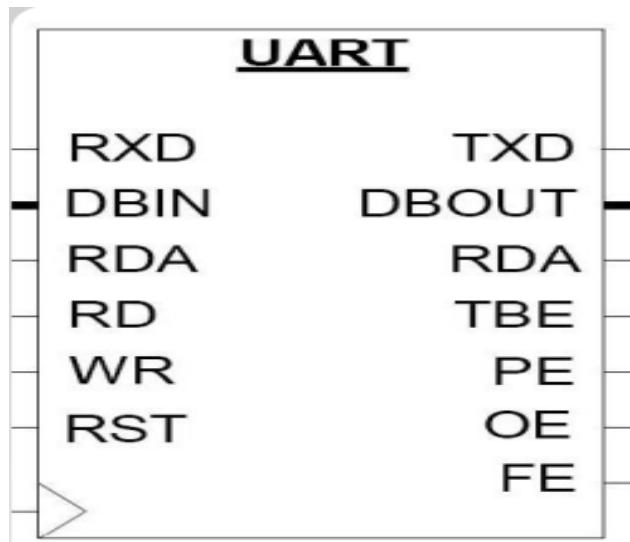
La periferica seriale fornisce un'interfaccia mediante la quale è possibile:

- Caricare un dato in parallelo ed abilitare la scrittura, in modo tale da trasmettere tale dato in maniera seriale, gestendo l'incapsulamento del dato in un frame con il quale è possibile effettuare un controllo degli errori lato ricevente e quindi ricevere informazioni di stato sull'operazione di trasmissione.
- Ricevere dati in maniera seriale e fornirli in uscita in parallelo al layer successivo, gestendo problematiche di sfasamento tra il clock di chi trasmette ed il clock di chi riceve mediante sovra-campionamento della linea ed effettuando un controllo degli errori sia per inviare informazioni di stato a chi trasmette sia per segnalare errori al layer che utilizza la periferica seriale per ricevere dati.
- Configurare parametri della comunicazione mediante un'unità di controllo come specificato dallo standard **RS-232**: si scrive in un registro MODE per specificare se si vuole effettuare trasmissione sincrona o asincrona, il tipo di parità, il numero di bit di parità, il numero di bit di stop, il numero di bit di dato, il numero di bit di sincronizzazione, e si usa una COMMAND word per “programmare” la seriale, ad esempio per avviare la trasmissione o abilitare/disabilitare la ricezione. In questo modo è possibile configurare la modalità di trasmissione (simplex, half duplex o full duplex) ed è possibile modificare il baud rate in un certo range.

È chiaro che, in generale, se definiamo il rendimento come il rapporto tra il numero di bit utili ed il numero di bit trasmessi, una comunicazione sincrona ha un rendimento più alto (più prossimo ad 1): questo perché è sufficiente una sincronizzazione iniziale, dopo la quale si trasmette un treno di messaggi, terminato da un messaggio finale (EOT – End Of Train). Tuttavia, non sempre è possibile effettuare la trasmissione in maniera sincrona: se la trasmissione è asincrona, è necessario un header

ed un trailer per ogni messaggio da inviare, ed il ricevente dovrà consumare caratteri piuttosto che consumare messaggi. I dispositivi che implementano tutte le funzionalità di una interfaccia RS-232, alcune delle quali abbiamo appena descritto, sono gli **USART** (Universal Synchronous-Asynchronous Receiver/Transmitter); essi sono dunque dispositivi configurabili. Se sappiamo che la comunicazione dovrà essere asincrona, ci possiamo accontentare di un dispositivo che rilassa alcuni vincoli sulla configurabilità, permettendo di configurare solo il formato dei dati e la velocità di trasmissione: un dispositivo di questo tipo è detto **UART**.

Concentriamoci su una specifica implementazione dell'UART, quella della Digilent. Vediamo la sua interfaccia nella seguente figura:

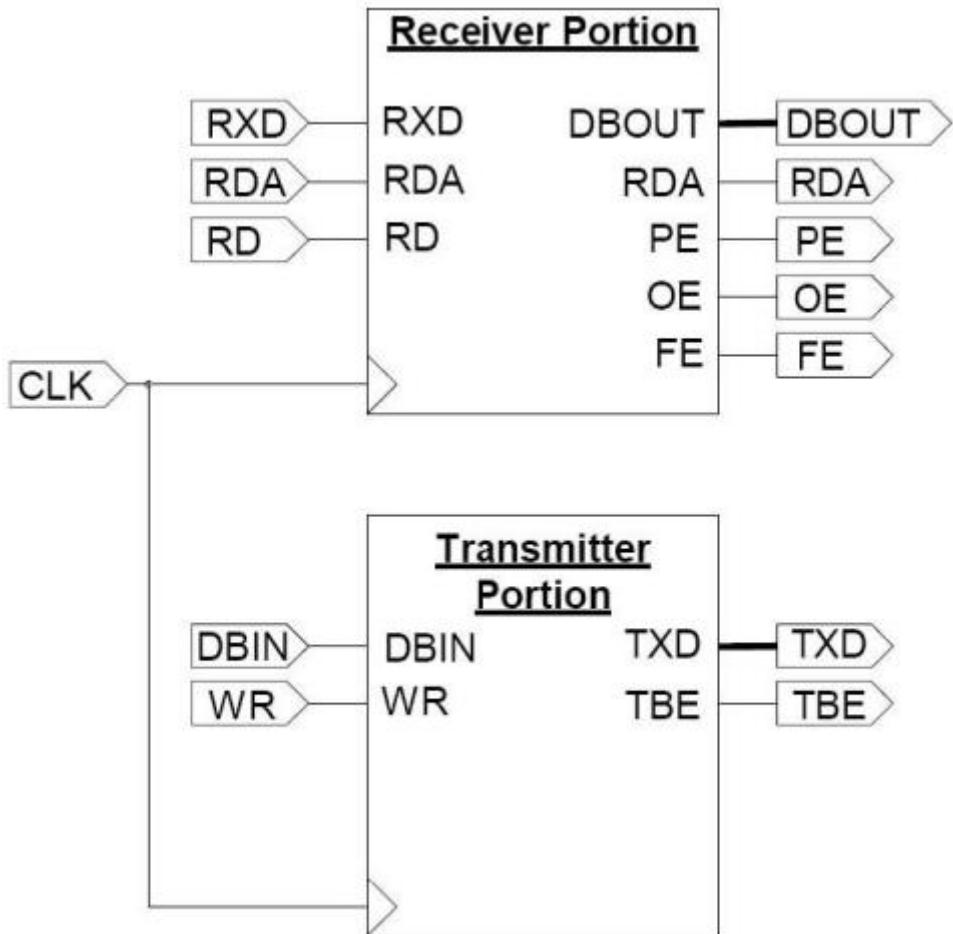


Analizziamo nel dettaglio i segnali presenti nell'interfaccia:

- RXD è l'ingresso seriale dei dati, i.e. i dati ricevuti dall'interfaccia seriale entrano nella porta relativa a RXD e sono scritti in uno shift register. Tali dati sono gestiti mediante sovraccampionamento a causa della deriva tra i clock di trasmettitore e ricevitore, quindi c'è un componente che usa un clock 16 volte più veloce e si pone sulla linea di mezzeria di ciascun bit, in corrispondenza della quale campiona il valore da scrivere nello shift register.
- DBIN è l'ingresso parallelo relativo alla word che si vuole trasmettere: quindi l'UART effettuerà una conversione parallelo-serie mediante un altro shift register.
- RDA (Read Data Available) è un segnale di ingresso/uscita, ovvero è collegato ad un bus dati, è posto alto quando termina la ricezione di un dato e dev'essere abbassato da chi legge il dato.
- RD è un segnale che serve per abilitare le uscite del ricevitore, ovvero per abilitare la lettura: esso è attivo basso, dunque quando vale '0' possono essere settati i segnali FE, OE, RDA, PE ed il registro collegato a DBOUT da parte del ricevitore, mentre quando vale '1' si comporta come un reset per i quattro segnali detti (quindi per tutti quelli di cui permette il set tranne il registro).
- WR serve per abilitare la scrittura, ovvero chi vuole utilizzare l'UART per trasmettere deve settare WR a 1; un'implementazione "corretta" dell'UART deve includere nell'automa a stati finiti del trasmettitore uno stato che aspetta che il segnale WR si abbassi, altrimenti si continua a trasmettere sempre lo stesso dato ("garbage transmission").
- RST è il classico segnale di reset, che viene utilizzato in maniera sincrona in questa implementazione.
- È presente un segnale di clock, ovviamente: la frequenza di tale clock deve essere conforme alla configurazione del baud rate della seriale; approfondiremo questo aspetto in seguito, in fase di progettazione dei componenti che utilizzano la seriale.

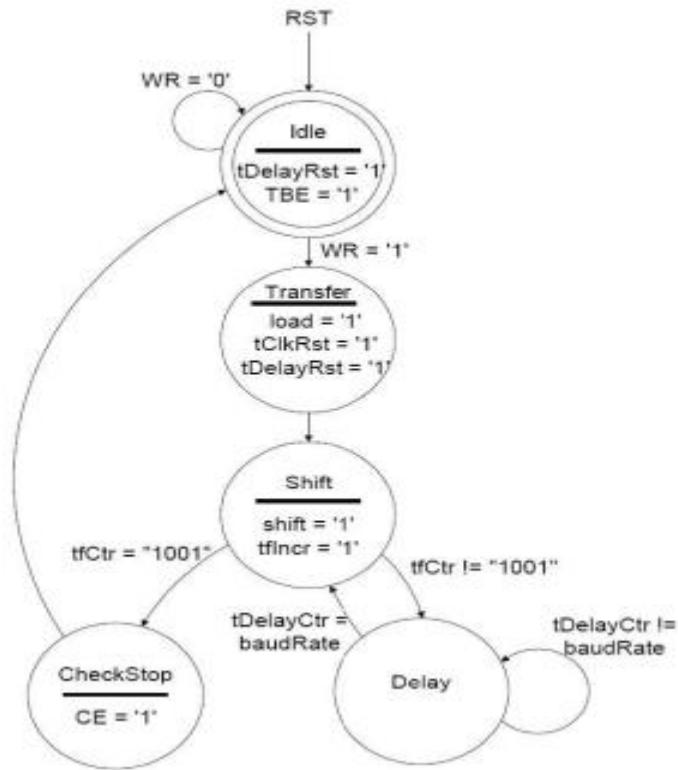
- TXD è l'uscita seriale dei dati: quando si setta WR a 1, si carica il contenuto di DBIN in un registro, e si carica in parallelo uno shift register con la seguente word: ('1' & par & tfReg(7 downto 0) &'0'); si trasmette prima il bit meno significativo dello shift register, che sarà '0', questo perché la linea è attiva bassa e quindi TXD è sempre '1' quando non si trasmette (quindi chi riceve capisce che l'altro sta trasmettendo quando legge '0' su RXD); dopodiché si trasmettono i dati a partire dal bit meno significativo, poi il bit di parità ed infine un bit di stop. La seriale gestisce opportunamente le tempificazioni delle attività in modo tale sia che nel registro sia già presente il valore di DBIN sia che il bit di parità sia calcolato quando si effettua il caricamento parallelo nello shift register, quindi ci aspettiamo che lo shift register sia caricato in modo parallelo due colpi di clock dopo lo scheduling di WR pari ad 1.
- DBOUT è l'uscita parallela dei dati: al termine di una ricezione, i bit presenti nello shift register corrispondenti ai dati ricevuti sono letti in parallelo e scritti in un registro, il quale coincide con l'uscita DBOUT.
- TBE (Text Buffer Empty) serve per segnalare al trasmettitore che non c'è nessuna trasmissione in corso, e quindi che la macchina è pronta a trasmettere.
- PE (Parity Error) corrisponde ad un errore di parità, controllato mediante xor tra i bit del dato ricevuto, xor del risultato con il bit di parità, e negazione di quest'ultimo risultato.
- OE (Overrun Error) è un segnale che si alza al termine della ricezione di un dato se e solo se il segnale RDA è alto: in pratica corrisponde alla situazione di errore che si ha quando un dato viene ricevuto, non viene letto dal layer successivo e poi si riceve un altro dato, il quale sovrascrive il precedente.
- FE (Framing Error) è alto se il bit di stop, il quale è l'ultimo bit ad essere ricevuto, è basso.

Internamente, la seriale si suddivide in maniera modulare in una porzione dedicata alla trasmissione e in una porzione dedicata alla ricezione, dunque l'interfaccia che abbiamo presentato si suddivide come mostrato nella seguente figura:



A questo punto ci resta da capire come sono fatti gli automi di trasmettitore e ricevitore. Entrambi gli automi implementano quanto detto per le rispettive interfacce, quindi non ripeteremo il discorso riguardante lo stimolo iniziale ed i segnali di stato.

Per **implementare il trasmettitore**, abbiamo bisogno di un contatore per gestire il ritardo tra una trasmissione e la successiva, il quale ha il compito di garantire che la trasmissione avvenga con un certo baud rate; inoltre, abbiamo bisogno di un contatore per tenere traccia del numero di trasmissioni effettuate: una configurazione tipica è quella con 8 bit di dato, 1 bit di parità ed 1 bit di stop, quindi dopo 10 trasmissioni si evolve verso una terminazione che poi riporta allo stato idle, in cui si segnala che non c'è alcuna trasmissione in corso. Si usa un segnale di *load* per il caricamento parallelo nello shift register usato in trasmissione, si usano dei segnali di reset interni per i contatori, si usa un segnale di *shift* per far shiftare lo shift register usato in trasmissione e per ogni shift si usa anche un segnale di abilitazione del contatore di caratteri inviati. Si ha dunque il seguente automa:

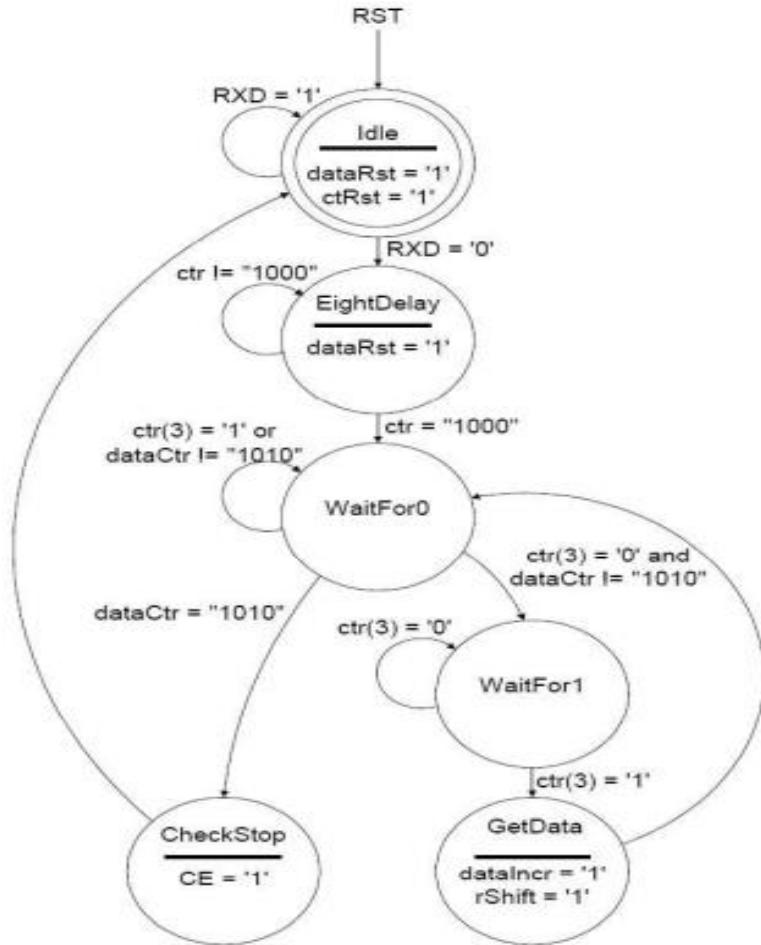


La gestione del segnale TBE è in realtà più complessa, in quanto è realizzata con un altro automa, il quale aspetta che il segnale WR si abbassi prima di tornare nello stato idle e quindi di segnalare TBE alto: lo stato di attesa di WR si chiama “WaitWrite”. Dunque l'automa visto sottintende la presenza di un altro automa dedicato alla gestione del segnale TBE: in realtà è quest'ultimo automa ad essere sensibile al segnale WR, mentre l'automa che gestisce il trasferimento “parte” quando TBE si abbassa. Questo vuol dire che il segnale WR comanda l'automa che gestisce TBE, il quale comanda l'automa che gestisce il trasferimento mediante il segnale TBE, appunto. Tuttavia, è più semplice comprendere la fase di trasferimento considerando questo automa semplificato.

Per **implementare il ricevitore**, ci serve una rete di controllo relativa ai dati ricevuti serialmente, un contatore regolato da un clock più veloce di quello usato per gli altri componenti (in genere 16 volte più veloce, in modo da campionare 16 volte ciascun bit ricevuto in ingresso), un contatore del numero di bit ricevuti (che da protocollo dovranno essere pari a quelli trasmessi, chiaramente), uno shift register per memorizzare i bit ricevuti ed un circuito combinatorio per effettuare il controllo di parità. Il valore di ciascun bit può essere deciso in base a diversi approcci, in teoria: uno consiste nello scegliere il valore del bit in base alla *moda* dei campioni, quindi se il numero di campioni pari ad 1 è maggiore o uguale di 8 allora scelgo 1, altrimenti scelgo 0; l'altro consiste nel decidere per il valore statisticamente più stabile, che è quello che si trova “in mezzo”, dunque la *mediana*.

L'implementazione che stiamo considerando sceglie la mediana: dunque, dopo aver campionato uno '0' (il quale indica che la linea è attiva), si pone in uno stato in cui resetta il contatore dei dati ricevuti ed aspetta 8 campioni. Visto che stiamo considerando ancora lo start bit, dobbiamo in pratica aspettare 8 campioni per porci sulla cella centrale dello start bit, poi altri 16 per porci sulla cella centrale del bit successivo. Tuttavia, è poco pratico contare 16 se stiamo sulla cella centrale di un contatore a 4 bit: la misura più pratica consiste nel contare 16 contando 8 alla volta, e tale misura si realizza efficacemente notando che la cella centrale è 1000 e dopo 8 campioni si ha 0000. Quindi, semplicemente, i primi 8 campioni a partire della cella centrale di ciascun bit successivo allo start si contano aspettando che il bit più significativo del contatore passi da 1 a 0, poi i successivi 8 campioni si contano aspettando che il bit più significativo del contatore passi da 0 a 1, dopodiché è possibile abilitare sia lo shift del registro a scorrimento in ricezione mediante il segnale *rShift* sia il segnale di abilitazione del contatore del numero di bit ricevuti mediante il segnale *dataIncr*. Questa “procedura” termina quando si ricevono 10 bit (nella configurazione descritta prima), e si entra in

uno stato in cui si alza RDA e si segnalano eventuali condizioni di errore. L'automa che realizza il ricevitore è dunque rappresentato nella seguente figura:



Per ribadire il concetto del sovra-campionamento, vediamo un diagramma temporale che ci dovrebbe chiarire ulteriormente le idee:



9.3 Progettazione, implementazione, simulazione e sintesi su FPGA dei componenti

9.3.1 Progettazione UART tappo e 2_UART

Per progettare i due componenti richiesti, è stata necessaria una fase preliminare di ragionamento riguardante il clock ed il baud. Il *baud rate* è definito come il rapporto tra la frequenza del clock che pilota la trasmissione dei bit ed il baud, cioè il numero di simboli trasmessi al secondo. Il *baud*

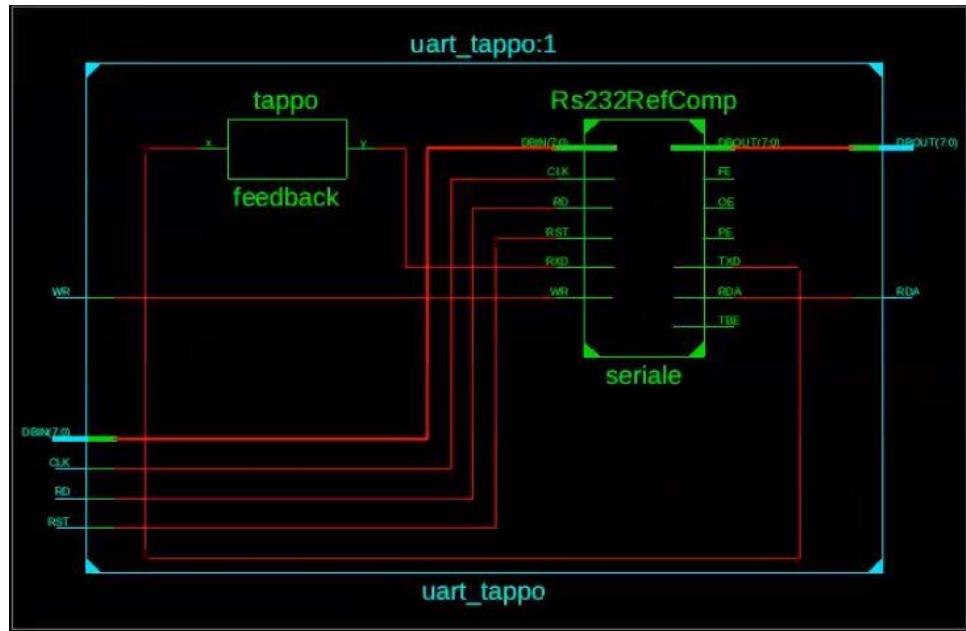
divide corrisponde ad una frazione del baud rate, che dipende dal numero di campioni presi per ciascun bit: nel nostro caso il numero di campioni è 16, quindi il baud divide è pari al baud rate diviso 16. Dunque, si avrà che il clock di ricezione dipende direttamente dal baud rate, mentre il clock di trasmissione dipende dal baud divide, quindi il clock di ricezione è 16 volte più veloce del clock di trasmissione. Nella pratica, visto che per implementare i clock si usano dei contatori, si usa come segnale del clock di trasmissione il bit in posizione 3 (quarto bit meno significativo) del contatore del clock di ricezione: in questo modo il clock di trasmissione è effettivamente una divisione di 16 del clock di ricezione. Tenendo in mente sia il fatto che dobbiamo fare la sintesi su FPGA sia il fatto che abbiamo intenzione di fare una simulazione, scegliamo clock e baud in modo tale da avere un clock non troppo elevato (useremo il componente clock filter per l'FPGA per ridurre la frequenza da 100 MHz a quella desiderata) e un baud divide piccolo, per osservare la simulazione in modo più semplice (il baud divide viene usato dal contatore di delay nell'automa della trasmissione, quindi per ogni bit da inviare, nello stato di delay si deve contare un numero di colpi di clock del clock di trasmissione pari al baud divide). Per non ripetere più volte i conti, abbiamo definito uno script in Python che prende in ingresso i parametri frequenza e baud desiderati e fornisce in uscita il baud rate ed il baud divide:

```
import sys

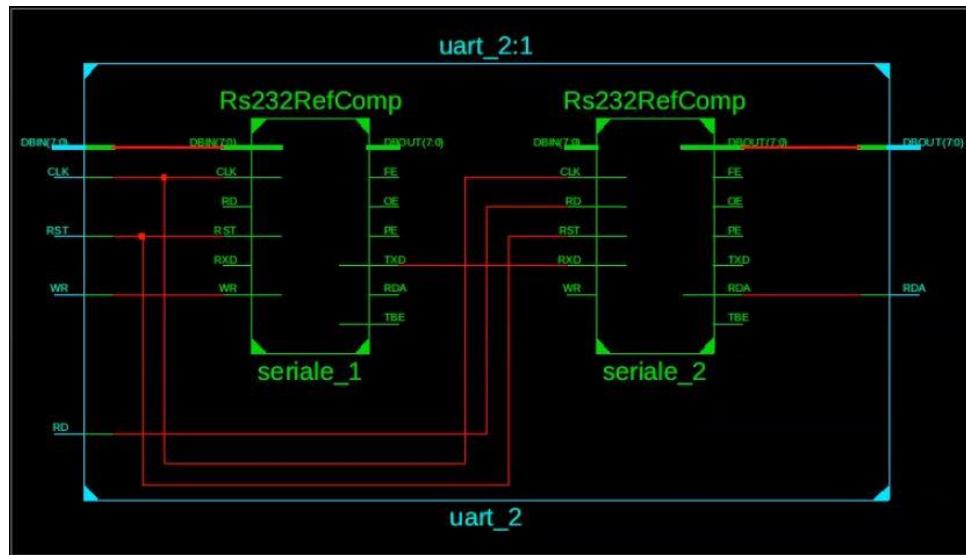
if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Error: provide frequency and baud")
    else:
        freq = int(sys.argv[1])
        baud = int(sys.argv[2])
        baud_rate = freq / baud
        baud_divide = int(baud_rate / 16)
        print("Baud rate: {}, baud divide: {}".format(baud_rate, baud_divide))
```

Alla fine, abbiamo scelto di usare una frequenza di 10 MHz e un baud di 115200, ottenendo un baud rate di 86.8 e un baud divide di 5; dal punto di vista pratico, bisogna mettere un clock a frequenza 10 MHz in ingresso al componente che implementa la seriale ed impostare la costante baudDivide pari a 5 all'interno del componente stesso.

Per quanto riguarda le implementazioni dei componenti, è più immediato vederne le figure e poi il relativo codice VHDL. Questa è quella relativa ad “UART tappo”:



Per quanto riguarda “2 UART”:



9.3.2 Implementazione in VHDL

Il componente tappo:

```
entity tappo is
  Port ( x : in STD_LOGIC;
         y : out STD_LOGIC);
end tappo;
```

```
architecture Behavioral of tappo is
```

```
begin
  y <= x;
end Behavioral;
```

Quindi, l'uart_tappo:

```

entity uart_tappo is
Port(
    CLK  : in  std_logic;
    DBIN : in  std_logic_vector (7 downto 0);
    DBOUT : out std_logic_vector (7 downto 0);
    RST  : in  std_logic:= '0';
    WR   : in  std_logic;
    RD   : in  std_logic;
    RDA  : inout std_logic
);
end uart_tappo;

architecture Behavioral of uart_tappo is

signal tx, rx : std_logic;
-- signal rd : std_logic := '0'; -- serve per abilitare la diagnostica

component tappo is
    Port ( x : in STD_LOGIC;
           y : out STD_LOGIC);
end component;

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

begin

seriale : Rs232RefComp port map(
    CLK => CLK, RST => RST, DBIN => DBIN, WR => WR, DBOUT => DBOUT,
    TXD => tx, RXD => rx, RD => '0', RD => RD, RDA => RDA
);

feedback : tappo port map(
    x => tx, y => rx
);

```

```
end Behavioral;
```

Il componente uart_2 è simile:

```
entity uart_2 is
Port(
    CLK  : in  std_logic;
    DBIN : in  std_logic_vector (7 downto 0);
    DBOUT : out std_logic_vector (7 downto 0);
    RST  : in  std_logic := '0';
    WR   : in  std_logic;
    RD   : in  std_logic;
    RDA  : inout std_logic
);
end uart_2;

architecture Behavioral of uart_2 is

signal tx : std_logic := '1';

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

begin

seriale_1 : Rs232RefComp port map(
    CLK => CLK, RST => RST, DBIN => DBIN, WR => WR,
    TXD => tx, RXD => '1', RD => '1'
);

seriale_2 : Rs232RefComp port map(
    CLK => CLK, RST => RST, WR => '0', RDA => RDA, DBIN => "00000000",
    RXD => tx, RD => RD, DBOUT => DBOUT
);
```

```
end Behavioral;
```

Notare che i due componenti hanno la stessa interfaccia; questo ci suggerisce che possiamo sfruttare il binding dinamico di più architecture alla stessa entity in fase di sintesi su FPGA, dove includeremo questi componenti in una control unit.

9.3.3 Simulazione

I due testbench sono praticamente identici.

Testbench di uart_tappo:

```
ENTITY uart_tappo_testbench IS
END uart_tappo_testbench;

ARCHITECTURE behavior OF uart_tappo_testbench IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT uart_tappo
        PORT(
            CLK : IN std_logic;
            DBIN : IN std_logic_vector(7 downto 0);
            DBOUT : OUT std_logic_vector(7 downto 0);
            RST : IN std_logic;
            WR : IN std_logic;
            RD : IN std_logic;
            RDA : inout std_logic
        );
    END COMPONENT;

    --Inputs
    signal CLK : std_logic := '0';
    signal DBIN : std_logic_vector(7 downto 0) := (others => '0');
    signal RST : std_logic := '0';
    signal WR : std_logic := '0';
    signal RD : std_logic := '0';

    --Outputs
    signal DBOUT : std_logic_vector(7 downto 0);
    signal RDA : std_logic := '0';

    -- Clock period definitions
    constant CLK_period : time := 100 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: uart_tappo PORT MAP (
```

```

CLK => CLK,
DBIN => DBIN,
DBOUT => DBOUT,
RST => RST,
WR => WR,
RDA => RDA,
RD => RD
);

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

-- Stimulus process
stim_proc: process
begin
  wait for 100 ns;
  RST <= '1';
  wait for CLK_period;
  RST <= '0';
  DBIN <= "01000111"; -- 71
  WR <= '1';
  wait until RDA = '1';
  wait for CLK_period;
  RD <= '1';
  WR <= '0';
  assert DBOUT = "01000111";

  wait;
end process;

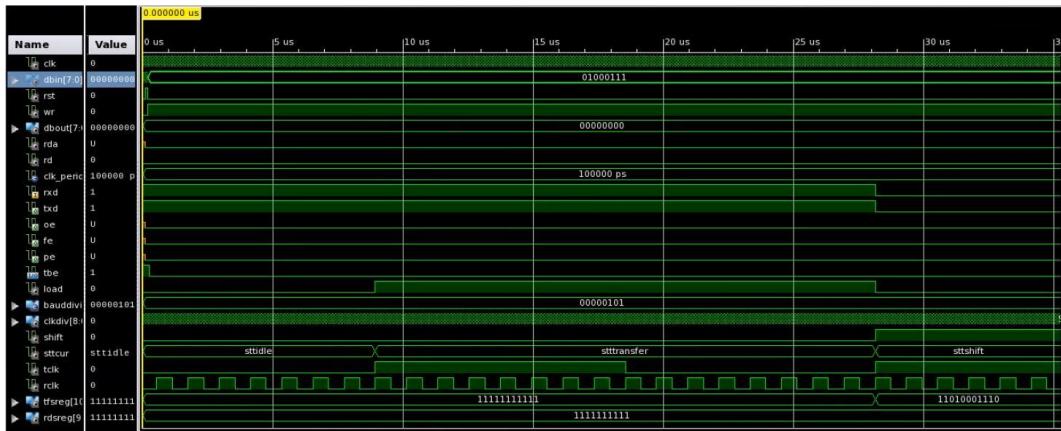
END;

```

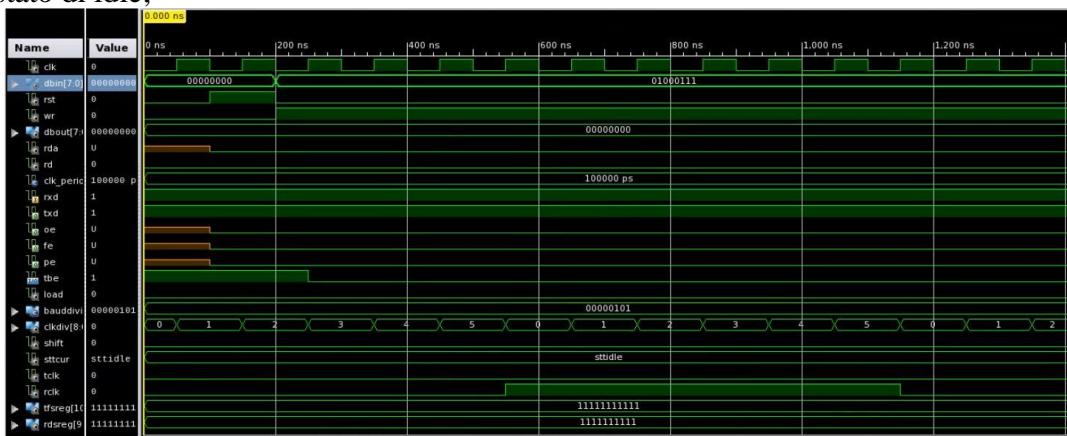
Risultato della simulazione:

Abbiamo eseguito più test sulla periferica UART facendo notare tre aspetti secondo noi significativi:

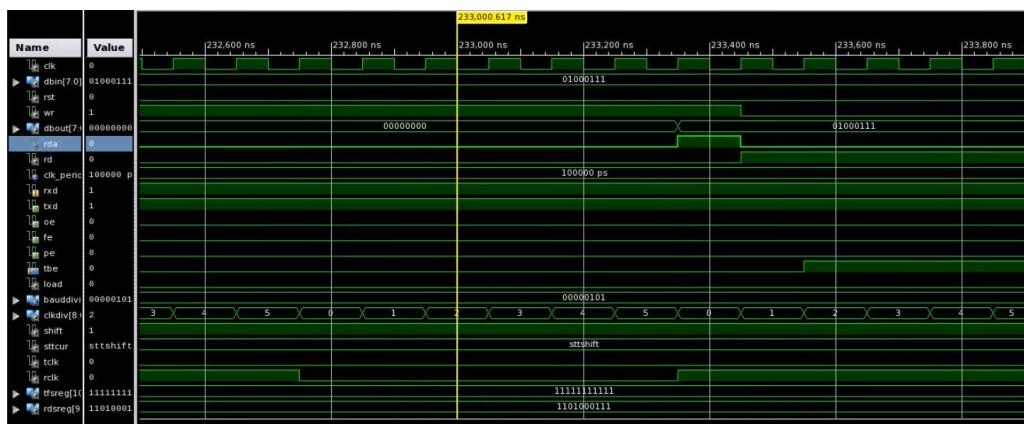
- Testing del clock, abbiamo voluto verificare che il clock del ricevitore fosse 16 volte più veloce del clock del trasmettitore, come si evince dal grafico di cui sotto tale relazione è rispettata;



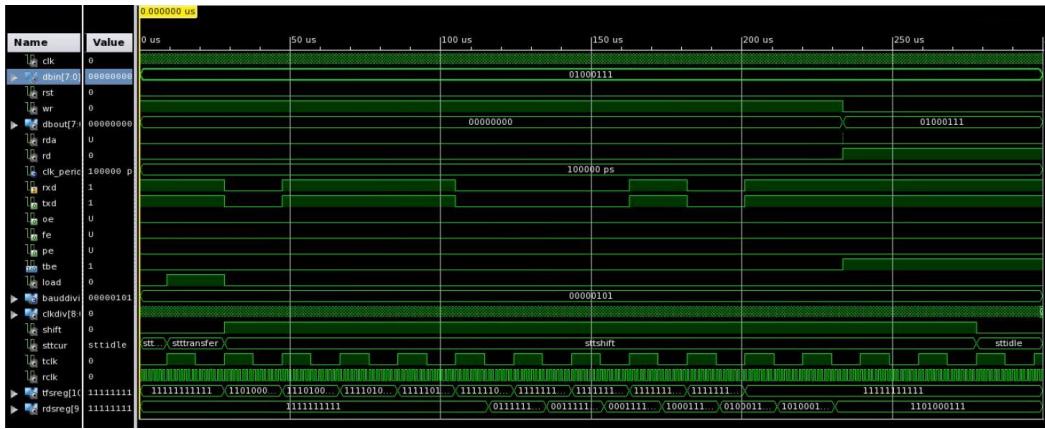
- Testing del reset, verificando che annulli ogni tipo di operazione e riporti la macchina in uno stato di idle;



- Test rda, verificando che il segnale di rda sia alto e che il client comunicando con la uart alzi il segnale rd che resetta i segnali di controllo errori ed rda;



- Infine, vediamo i risultati del testing generale notando che dbout e dbin hanno lo stesso valore.



Testbench di uart_2:

```
ENTITY uart_2_testbench IS
END uart_2_testbench;

ARCHITECTURE behavior OF uart_2_testbench IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT uart_2
PORT(
    CLK : IN std_logic;
    DBIN : IN std_logic_vector(7 downto 0);
    DBOUT : OUT std_logic_vector(7 downto 0);
    RST : IN std_logic;
    WR : IN std_logic;
    RD : IN std_logic;
    RDA : INOUT std_logic
);
END COMPONENT;

--Inputs
signal CLK : std_logic := '0';
signal DBIN : std_logic_vector(7 downto 0) := (others => '0');
signal RST : std_logic := '0';
signal WR : std_logic := '0';
signal RD : std_logic := '0';

--BiDirs
signal RDA : std_logic;

--Outputs
signal DBOUT : std_logic_vector(7 downto 0);

-- Clock period definitions
constant CLK_period : time := 100 ns;

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: uart_2 PORT MAP (
    CLK => CLK,
    DBIN => DBIN,
    DBOUT => DBOUT,
    RST => RST,
    WR => WR,
    RDA => RDA,
```

```

        RD => RD
    );

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

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100 ns.
    wait for 100 ns;
    RST <= '1';
    wait for CLK_period;
    RST <= '0';
    DBIN <= "01000111"; -- 71
    WR <= '1';
    wait until RDA = '1';
    wait for CLK_period;
    RD <= '1';
    WR <= '0';
    assert DBOUT = "01000111";

    wait;
end process;

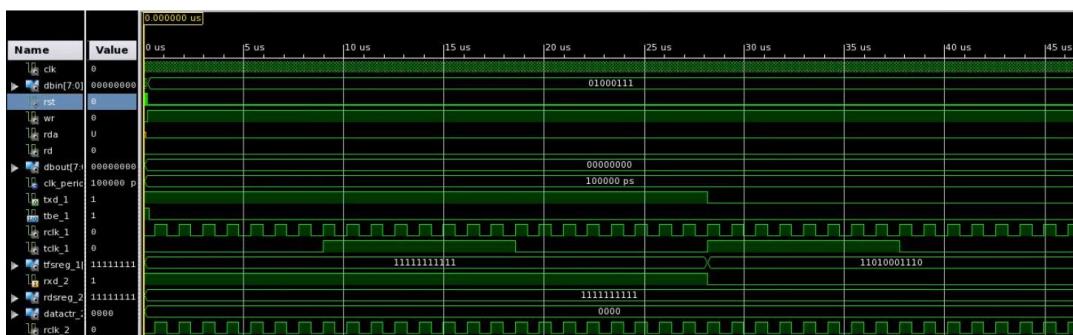
```

END;

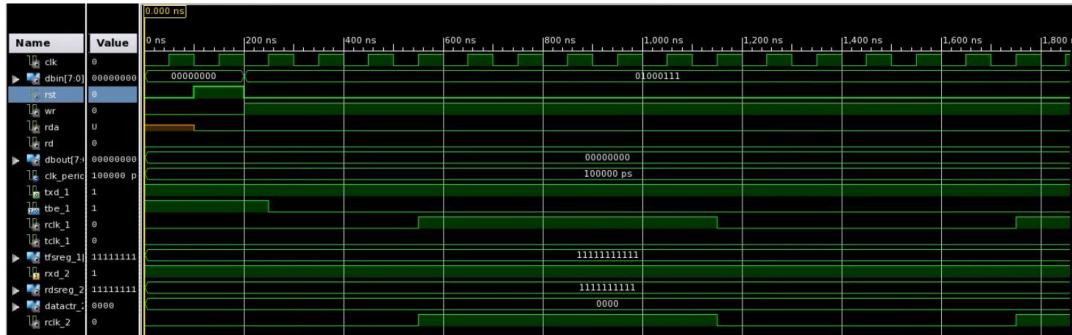
Risultato della simulazione:

Abbiamo eseguito più test sulla periferica UART facendo notare tre aspetti secondo noi significativi:

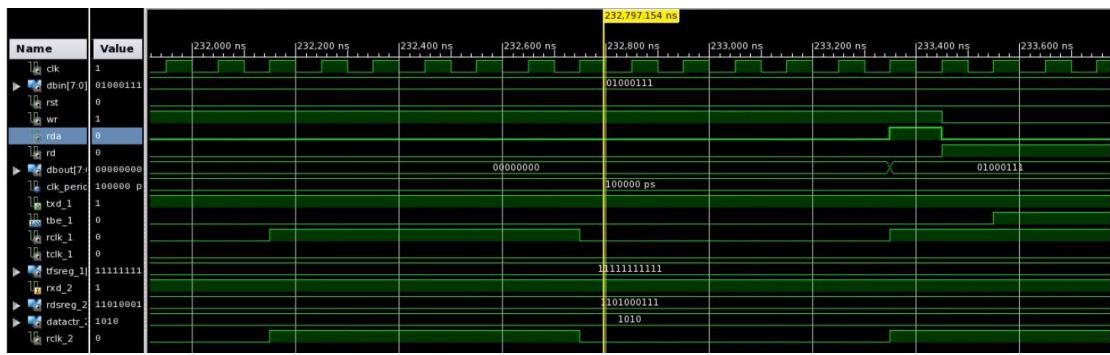
- Testing del clock, abbiamo voluto verificare che il clock del ricevitore fosse 16 volte più veloce del clock del trasmettitore, in questo caso usando due dispositivi uart abbiamo dovuto controllare la sincronia tra i due clock e abbiamo indicato con il suffisso uno i clock relativi al trasmettitore e con il suffisso due i clock relativi al ricevitore;



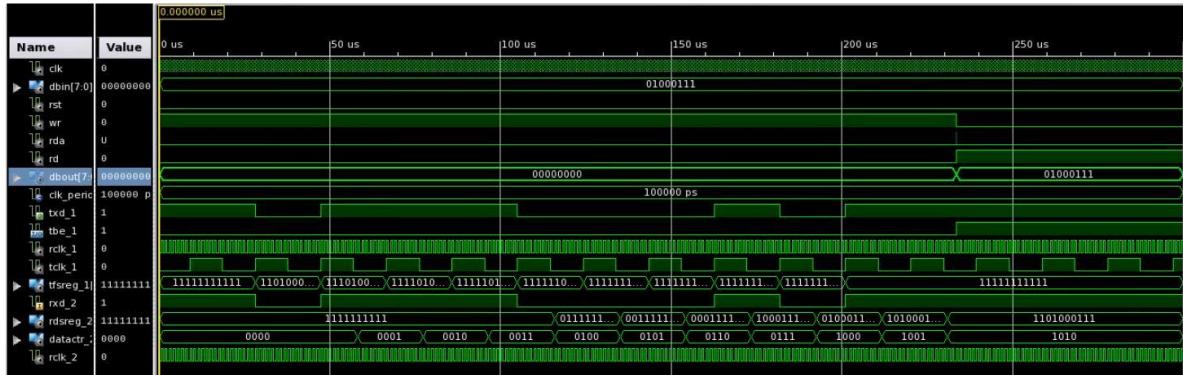
- Testing del reset, verificando che annulli ogni tipo di operazione e riporti le macchine in uno stato di idle;



- Test rda, verificando che il segnale di rda sia alto e che il client comunicando con la uart usata come receiver alzi il segnale rd che resetta i segnali di controllo errori ed rda;



- Infine, abbiamo un test generale per verificare il funzionamento di due uart una usata come trasmettitore una usata come ricevitore.



9.3.4 Sintesi su FPGA

La sintesi della UART sulla scheda ha previsto un accorgimento riguardante le frequenze a cui le componenti implementate dovevano lavorare, essendo queste tarate su una frequenza di 10MHz si è organizzato un filtraggio del clock della board che è di 100MHz.

Abbiamo gestito l'input da dare alla board mediante 8 switch ed un bottone: gli 8 switch servono per fornire un dato in parallelo alla seriale, la quale effettua quindi la conversione parallelo-serie quando le si fornisce il segnale di write (mediante il bottone); si ottiene l'output quando il componente che contiene l'uart in configurazione tappo o le due uart collegate tra loro fornisce il segnale RDA alto. A tal punto, bisognerebbe abbassare write ed alzare read, in modo da far tornare la macchina in uno stato tale da accettare una nuova trasmissione. Abbiamo fatto in modo che questo aspetto fosse gestito

in maniera automatica da un processo nel top module, *client_protocol*; in pratica al posto di usare direttamente il bottone per il segnale di write, abbiamo definito un signal *actual_wr*, che viene settato pari al valore campionato dal bottone quando RDA è 0, mentre viene settato a 0 quando RDA è 1, per non avere errore di overrun. Per quanto riguarda il segnale di read, l'uscita parallela della seriale è direttamente collegata agli 8 bit meno significativi del vettore di bit che pilota i catodi, dunque in pratica il sistema legge ad ogni colpo di clock; per questo motivo, quando RDA si alza, si può supporre che il sistema abbia letto, e quindi si alza il segnale di read quando si campiona RDA pari ad 1. In caso contrario, il segnale di read resta pari a 0, quindi la soluzione più naturale consiste nell'assegnare il valore di read pari al valore di RDA.

Senza addentrarci nei componenti forniti dai Docenti (clock filter, cathodes manager ecc.) riportiamo la control unit, responsabile dei valori da fornire ai catodi, ed il top module cioè *display on board*. I componenti uart tappo e uart 2 sono stati inclusi nella stessa entity control unit sfruttando il fatto che si possono definire più architetture per la stessa entity, e poi nel top module è stato utilizzato il binding dinamico, importando “work.all”.

Control unit:

```

entity control_unit is
  Port (
    clock : in STD_LOGIC;
    reset_n : in STD_LOGIC;
    wr : in STD_LOGIC;
    dbin : in STD_LOGIC_VECTOR(7 downto 0);
    value : out STD_LOGIC_VECTOR(31 downto 0);
    enable : out STD_LOGIC_VECTOR(7 downto 0);
    rda : inout STD_LOGIC;
    rd : in STD_LOGIC
  );
end control_unit;

architecture tappo of control_unit is

signal reset : std_logic;
signal dbout : std_logic_vector(7 downto 0);

component uart_tappo is
Port(
  CLK  : in std_logic;                                --Master Clock
  DBIN : in std_logic_vector (7 downto 0); --Data Bus in
  DBOUT : out std_logic_vector (7 downto 0); --Data Bus out
  RST  : in std_logic := '0'; --Master Reset
  WR   : in std_logic; --Write Strobe(se 1 significa "scrivi")
  RDA  : inout std_logic; --Read Data Available( alto quando il
                         --dato disponibile nel registro rdReg)
  RD   : in std_logic
);
end component;

begin

reset <= not reset_n;

```

```

enable <= "00000011";
value(31 downto 8) <= (others => '0');
value(7 downto 0) <= dbout;

seriale : uart_tappo port map(
    CLK => clock, DBIN => dbin, DBOUT => dbout, RST => reset, WR => wr,
    RDA => rda, RD => rd
);

end tappo;

architecture due_seriali of control_unit is

signal reset : std_logic;
signal dbout : std_logic_vector(7 downto 0);

component uart_2 is
    Port(
        CLK : in std_logic;      --Master Clock
        DBIN : in std_logic_vector (7 downto 0); --Data Bus in
        DBOUT : out std_logic_vector (7 downto 0); --Data Bus out
        RST : in std_logic := '0'; --Master Reset
        WR : in std_logic; --Write Strobe(se 1 significa
        "scrivi")
        RDA : inout std_logic; --Read Data Available( alto quando
        il dato disponibile nel registro rdReg)
        RD : in std_logic
    );
end component;

begin

reset <= not reset_n;
enable <= "00000011";
value(31 downto 8) <= (others => '0');
value(7 downto 0) <= dbout;

seriale : uart_2 port map(
    CLK => clock, DBIN => dbin, DBOUT => dbout, RST => reset, WR => wr,
    RDA => rda, RD => RD
);

end due_seriali;

```

Display on board:

```

entity display_on_board is
Port(

```

```

clock : in STD_LOGIC;
reset : in STD_LOGIC;
wr : in STD_LOGIC;
dbin : in STD_LOGIC_VECTOR(7 downto 0);
anodes : out STD_LOGIC_VECTOR (7 downto 0); --anodi e catodi delle cifre
cathodes : out STD_LOGIC_VECTOR (7 downto 0)
);
end display_on_board;

architecture Structural of display_on_board is

COMPONENT display_seven_segments
GENERIC(
clock_frequency_in : integer := 50000000; --questi parametri servono a
configurare
clock_frequency_out : integer := 5000000 --il clock filter
);
PORT(
    clock : IN std_logic;
    reset_n : IN std_logic;
    value : IN std_logic_vector(31 downto 0); --6 nibble da mostrare
    enable : IN std_logic_vector(7 downto 0); --abilitazione delle 4
cifre
    dots : IN std_logic_vector(7 downto 0); --punti
    anodes : OUT std_logic_vector(7 downto 0);
    cathodes : OUT std_logic_vector(7 downto 0)
);
END COMPONENT;

component control_unit is
Port (
    clock : in STD_LOGIC;
    reset_n : in STD_LOGIC;
    wr : in STD_LOGIC;
    dbin : in STD_LOGIC_VECTOR(7 downto 0);
    value : out STD_LOGIC_VECTOR(31 downto 0);
    enable : out STD_LOGIC_VECTOR(7 downto 0);
    rda : out STD_LOGIC
);
end component;

component clock_filter is
generic(
    clock_frequency_in : integer := 50000000;
    clock_frequency_out : integer := 5000000
);
Port ( clock_in : in STD_LOGIC;

```

```

        reset_n : in STD_LOGIC;
        clock_out : out  STD_LOGIC);
end component;

signal reset_n : std_logic;
signal cu_value : std_logic_vector(31 downto 0);
signal cu_enable : std_logic_vector(7 downto 0);
signal cu_dots : std_logic_vector(7 downto 0) := (others => '0');
signal clk_seriale : std_logic;
signal actual_wr : std_logic;
signal rda : std_logic;
signal rd : std_logic := '0';

begin

reset_n <= not reset;

client_protocol : process(clk_seriale)
begin
    if (clk_seriale'event and clk_seriale = '1') then
        rd <= rda;
        if rda = '1' then
            actual_wr <= '0';
        else
            actual_wr <= wr;
        end if;
    end if;
end process;

clk_filter : clock_filter GENERIC MAP(clock_frequency_in => 100000000,
clock_frequency_out => 10000000)
    PORT MAP( clock_in => clock, reset_n => reset_n, clock_out =>
clk_seriale);

seven_segment_array: display_seven_segments GENERIC MAP(
clock_frequency_in => 100000000, --qui inserisco i parametri effettivi
(clock della board e clock in uscita desiderato)
clock_frequency_out => 50000
)
PORT MAP(
    clock => clock,
    reset_n => reset_n,
    value => cu_value,
    enable => cu_enable,
    dots => cu_dots,
    anodes => anodes,
    cathodes => cathodes
);
cu: entity work.control_unit(due_seriali) PORT MAP(

```

```

    clock => clk_seriale,
    reset_n => reset_n,
    wr => actual_wr,
    dbin => dbin,
    value => cu_value,
    enable => cu_enable,
    rda => rda,
    rd => rd
);

end Structural;

```

Riportiamo la parte che abbiamo utilizzato dei **constraints**:

```

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

##Switches
set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports {
dbin[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports {
dbin[1] }]; #IO_L3N_T0_DQS_EMCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 } [get_ports {
dbin[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 } [get_ports {
dbin[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 } [get_ports {
dbin[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMOS33 } [get_ports {
dbin[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMOS33 } [get_ports {
dbin[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMOS33 } [get_ports {
dbin[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]

##7 segment display
set_property -dict { PACKAGE_PIN T10      IOSTANDARD LVCMOS33 } [get_ports {
cathodes[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10      IOSTANDARD LVCMOS33 } [get_ports {
cathodes[1] }]; #IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16      IOSTANDARD LVCMOS33 } [get_ports {
cathodes[2] }]; #IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13      IOSTANDARD LVCMOS33 } [get_ports {
cathodes[3] }]; #IO_L17P_T2_A26_15 Sch=cd

```

```

set_property -dict { PACKAGE_PIN P15    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[6] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17    IO_STANDARD LVCMOS33 } [get_ports {
anodes[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18    IO_STANDARD LVCMOS33 } [get_ports {
anodes[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9     IO_STANDARD LVCMOS33 } [get_ports {
anodes[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14    IO_STANDARD LVCMOS33 } [get_ports {
anodes[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14    IO_STANDARD LVCMOS33 } [get_ports {
anodes[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14    IO_STANDARD LVCMOS33 } [get_ports {
anodes[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2     IO_STANDARD LVCMOS33 } [get_ports {
anodes[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13    IO_STANDARD LVCMOS33 } [get_ports {
anodes[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons
set_property -dict { PACKAGE_PIN N17    IO_STANDARD LVCMOS33 } [get_ports {
reset }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18    IO_STANDARD LVCMOS33 } [get_ports {
wr }]; #IO_L4N_T0_D05_14 Sch=btnu

```

Esercizio 10

10.1 Traccia

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

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).

(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).

(Opzionale) Si implementi un protocollo di handshaking semplice regolato da una coppia di segnali (pronto a inviare/pronto a ricevere) per l'invio di ciascun messaggio fra due nodi.

10.2 Richiamo generale sugli switch

Ci sono diversi approcci per l'implementazione degli switch, che sono sistemi in grado di interconnettere tra loro N host. Come al solito, partiamo dall'interfaccia. Un host dev'essere in grado di raggiungere un altro host specificandone l'indirizzo di destinazione. In generale, è richiesto anche di specificare l'indirizzo sorgente, ma questo aspetto può variare a seconda dell'implementazione dello switch: è possibile infatti che un host sia collegato ad una certa "porta" dello switch, e quindi che la specifica dell'indirizzo sorgente non sia *necessaria* per il routing del messaggio. In tal caso, l'indirizzo sorgente sarà incluso da chi trasmette come parte del contenuto informativo del messaggio, ed interessa al ricevitore, più che allo switch.

Ora, esaminiamo due categorie di switch:

- A **connessione diretta**, costituito da un mux seguito da un demux: questo tipo di switch richiede di specificare sia l'indirizzo sorgente sia l'indirizzo destinazione; ci sono varie problematiche: i dati "scorrono in un solo verso", non c'è parallelismo perché il mux ha una sola linea in uscita, e realizzare il parallelismo sarebbe costoso in quanto richiederebbe replicazione dell'hardware per un numero di volte pari al numero di connessioni simultanee che si vogliono avere. Inoltre, pur replicando l'hardware, non ci sono meccanismi di gestione delle collisioni, ovvero se più host cercano di scrivere allo stesso destinatario, ad un certo punto si troverebbero a scrivere sulla stessa linea. Lo switch a connessione diretta può quindi sembrare obsoleto, ma in realtà se si considera uno switch a connessione diretta con 2 ingressi e 2 uscite, questo può essere utilizzato come componente base per switch più complessi.
- Ad **N stadi (indiretti)**, in cui l'idea è di utilizzare dispositivi costituiti da due ingressi e due uscite in ciascun stadio, interconnettendoli opportunamente con gli ingressi, tra di loro e con le uscite. Se si utilizza un bit di indirizzamento per l'uscita di ogni stadio, allora per N stadi è necessario un totale di N bit di indirizzamento. Se consideriamo un host collegato ad uno specifico dispositivo, con il primo bit di indirizzamento seleziona una tra le due possibili uscite; con il secondo bit di indirizzamento seleziona una tra le due possibili uscite dello specifico dispositivo al quale la prima uscita ha portato, quindi in totale usando due bit di indirizzamento seleziona una tra quattro possibili uscite. Iterando questo ragionamento, è facile rendersi conto che con N bit di indirizzamento, un host seleziona una tra 2^N possibili uscite. In effetti, **ciascun host vede un albero binario**. Ora, dal punto di vista funzionale, uno switch deve interconnettere k host; affinché ciò sia possibile, sono necessari $\log(k)$

stadi, supponendo il logaritmo in base 2 ed approssimandolo per eccesso. Supponendo per semplicità di avere sempre k come potenza di 2, una tecnica per realizzare questa logica si chiama **perfect shuffling**: si basa sul concetto di dividere un mazzo di carte in due parti uguali e poi di ricombinare le due parti mettendo insieme la prima carta della prima metà con la prima della seconda metà e così via. Effettuando lo shuffling ad ogni stadio, se ci sono 2^N "carte", dopo N stadi si riottiene il mazzo originale. Dato che si "mischia perfettamente", per ogni host si ottiene la vista ad albero binario desiderata, e quindi è possibile raggiungere qualsiasi host a partire da qualsiasi host, usando come bit di indirizzamento proprio la rappresentazione binaria dell'host che si vuole raggiungere (inteso come "carta" con valore appartenente all'intervallo che va da 0 a $2^N - 1$).

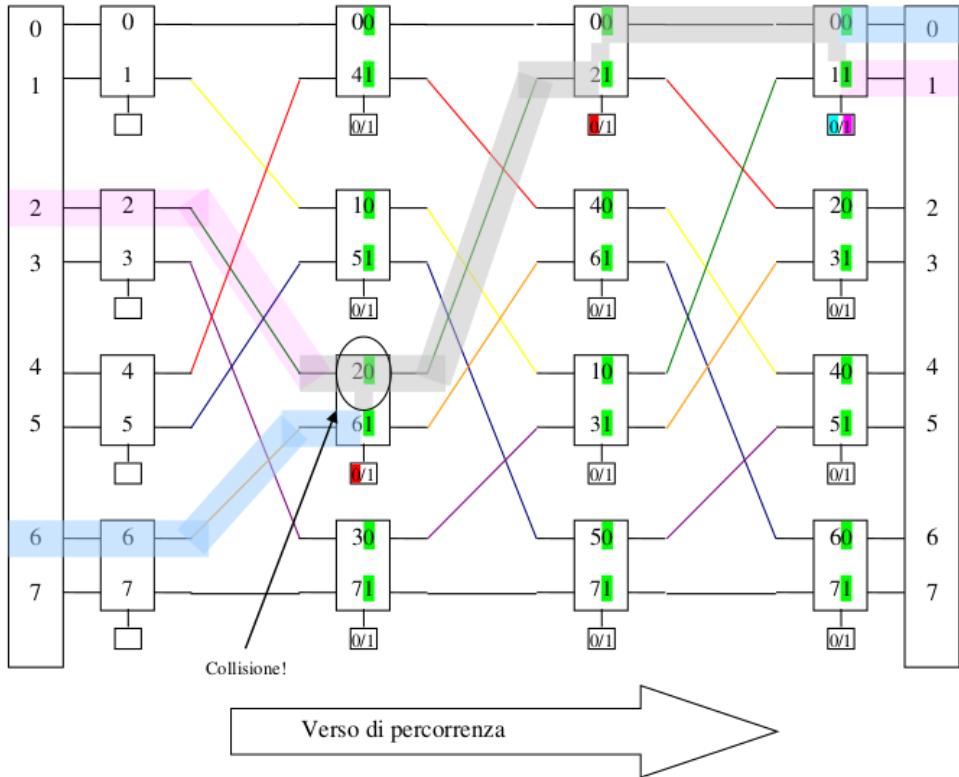
Lo switch ad N stadi che fa uso del perfect shuffling è detto **omega network**. È una soluzione efficiente perché si ha il massimo parallelismo con il minimo dispendio di hardware, ma non è esente da problemi di collisione. Tuttavia, a differenza dello switch a connessione diretta, dove l'indirizzo di destinazione è valutato in un solo passo e quindi non si possono gestire conflitti se non a monte con schemi di priorità, negli switch multistadio la logica è distribuita tra i nodi e l'indirizzo di destinazione è valutato un bit alla volta. È chiaro che anche per switch multistadio è possibile risolvere i conflitti a monte, ad esempio con uno schema a priorità fissa, ma non è la soluzione più efficiente. Per definire una logica di risoluzione di conflitti distribuiti, bisogna prima definire qual è il modello di comunicazione.

- Nel modello **wormhole**, si ha un meccanismo di comunicazione a pipeline: ogni nodo riceve un messaggio contenente un header e un payload, e mediante l'header decide dove instradare il messaggio ed eventualmente come gestire i conflitti; quindi, i nodi sono responsabili solo dell'instradamento dei messaggi: in generale non ne effettuano la memorizzazione. La pipeline è dovuta al fatto che non si mandano "grossi pacchetti", ma piccoli messaggi mediante i quali il ricevitore ricostruisce il messaggio complessivo, e ciascun piccolo messaggio viene subito inoltrato, quindi si ha un ritardo molto piccolo. Tale tecnica si può applicare con un approccio "pessimistico", come nelle reti a commutazione di circuito, dove si effettua una negoziazione di percorso preliminare alla comunicazione e poi il percorso è assegnato staticamente a tale comunicazione per tutta la durata della comunicazione stessa: si usa quando c'è un'esigenza di accoppiamento tra i comunicanti; oppure si può applicare con un approccio "ottimistico", in cui la negoziazione avviene dinamicamente, ovvero ciascun nodo decide in base ai messaggi che riceve se instradare oppure no un certo messaggio, ed una sequenza di messaggi lungo uno stesso percorso definisce un "serpentone" che corrisponde ad un percorso stabilito dinamicamente.
- Nel modello **store & forward**, non si ragiona a livello di percorso, ma a livello di tratta tra due nodi (anche gli host sono considerati come nodi): l'unità trasmissibile è il pacchetto, il quale contiene tutte le informazioni necessarie ad un nodo per poterlo instradare. Quando un nodo riceve un pacchetto, lo memorizza in un buffer (si può pensare ad un registro a scorrimento dal punto di vista hardware, o ad una coda dal punto di vista delle strutture dati), e prima di instradarlo esegue un *protocollo* mediante il quale effettua la negoziazione di tratta con il nodo verso il quale vuole effettuare l'instradamento. È chiaro che, una volta completato il protocollo, sarà trasmesso il pacchetto per intero. Dunque, non c'è un vero e proprio pipelining, quindi si introducono ritardi nella comunicazione, ma migliora l'occupazione complessiva di banda (*throughput*).

Nell'omega network si usa tipicamente un modello wormhole ottimistico. Avendo definito il modello di comunicazione, possiamo occuparci della gestione dei conflitti. Intanto classifichiamo il tipo di conflitto che si ha con questo modello di comunicazione: il conflitto avviene in caso di "intersezione" tra due serpentoni. Vediamolo meglio nella seguente figura, approfittandone per mostrare anche i collegamenti all'interno di un'omega network con 8 host.

Esempio: Dal nodo 2 si vuole andare al nodo 1; indirizzo 001; Dal nodo 6 si vuole andare al nodo 0 indirizzo 000:

N.B. L'indirizzo va considerato dal bit più significativo a quello meno significativo



In questo caso, la collisione è avvenuta al primo stadio. Il nodo in cui avviene la collisione deve prendere una decisione, e tale decisione è indipendente dallo stato degli altri nodi; in altre parole, ciascun nodo deve prendere una decisione arbitraria. Tuttavia, un requisito è che si deve evitare ciò che avviene in questa figura: il primo stadio decide di far passare il messaggio inviato dal nodo 2, il secondo stadio decide di far passare il messaggio inviato dal nodo 2 e poi il terzo stadio decide di far passare il messaggio che gli arriva verso l'indirizzo specificato dal nodo 6. Questa situazione è facilmente evitabile se i bit di indirizzamento fanno parte dell'header del messaggio; in caso contrario, bisogna prendere degli accorgimenti per garantire che una volta che un nodo decide di scartare un messaggio, tutti i nodi successivi devono ignorare l'indirizzamento relativo al messaggio scartato. L'ultima osservazione da fare sulle collisioni prima di presentare le tecniche per gestirle riguarda il dispositivo usato per realizzare le interconnessioni, ovvero l'unità operativa presente all'interno del nodo. Se tale dispositivo è realizzato con uno switch 2:2 a connessione diretta, si hanno collisioni sia in ingresso sia in uscita, perché è presente una sola linea tra il mux e il demux. Una soluzione più efficiente consiste nell'usare due multiplexer per gestire le uscite: ciascuno di questi mux ha come ingressi entrambi gli ingressi, ed il bit di selezione di ciascun mux è regolato da un'unità di controllo cablata nel nodo; se arrivano due messaggi con diversi bit di indirizzamento relativi a tale stadio, allora vuol dire che uno sarà l'uscita di un mux, e l'altro l'uscita dell'altro mux, quindi si possono far passare tali messaggi in parallelo, senza alcun conflitto. Il conflitto si ha solo se entrambi vogliono uscire sulla stessa linea. Inoltre, per ridurre ulteriormente il numero di conflitti, conviene introdurre anche un bit di abilitazione della linea relativa ad un certo messaggio, altrimenti si rilevano erroneamente conflitti quando invece i bit di indirizzamento erano relativi ad una vecchia trasmissione, ad esempio.

Detto questo, presentiamo le tecniche per gestire le collisioni:

- **Tecnica bloccante**, consiste nel fermare la pipeline in caso di conflitti, finché non si risolve il problema e si può riprendere la trasmissione; tale tecnica è conservativa e non è per niente efficiente.

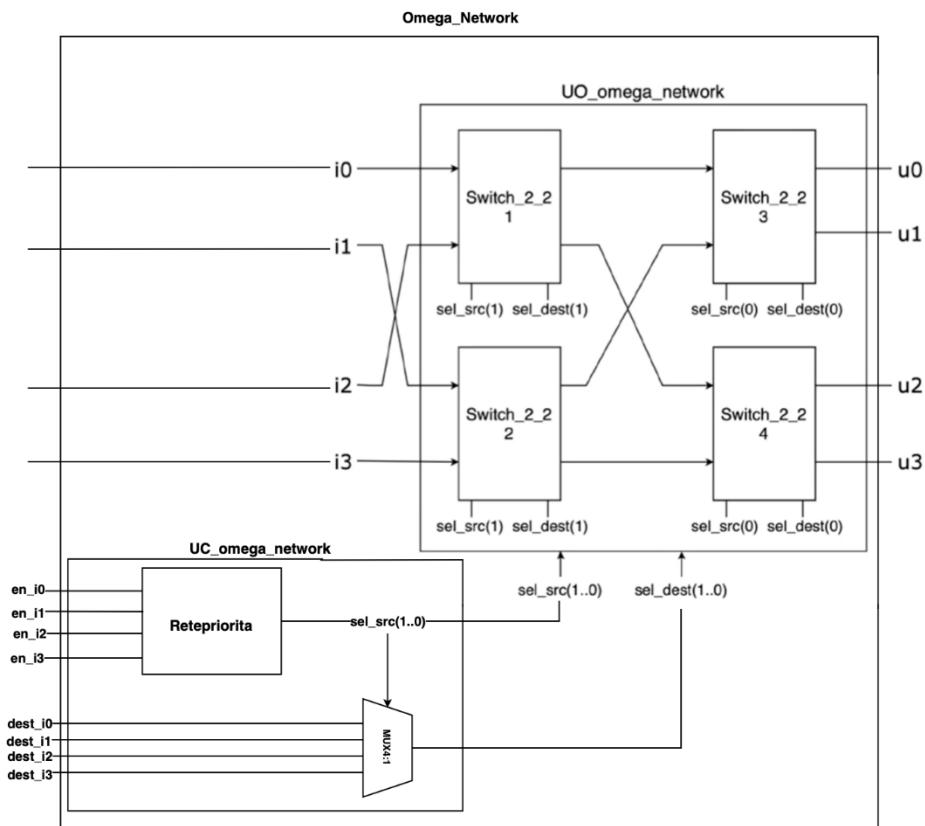
- **Rete con perdita di dati**, consiste nel gestire la collisione decidendo di inoltrare un messaggio piuttosto che l'altro, quindi ciascun nodo implementa a livello microscopico uno schema a priorità fissa tra due messaggi (in realtà se si considera una macchina sequenziale, la priorità può anche essere dinamica, ed in tal caso si ha uno *scheduler*).
- **Re-instradamento**, consiste nell'inoltrare il messaggio lungo percorsi alternativi; è possibile solo se le tabelle di routing non sono statiche e se esistono percorsi alternativi, quindi ad esempio non è possibile nell'omega network. Inoltre, se alcuni messaggi sono re-instradati, i messaggi possono arrivare in disordine al destinatario, e quindi nell'header del messaggio è necessario aggiungere dei numeri di sequenza, in modo tale che il destinatario li possa ricomporre nell'ordine giusto.
- **Cut-through**, consiste nel gestire i conflitti in modo simile alla rete con perdita di dati, solo che al posto di “scartare” i messaggi che non vengono inoltrati, tali messaggi sono gestiti in una memoria, dalla quale sono inoltrati dopo la risoluzione del conflitto; per l'implementazione sono necessari uno shift register ed un contatore di messaggi presenti nello shift register. Dunque, si ha una filosofia store & forward all'interno del wormhole. L'ovvio inconveniente è che il buffer può riempirsi a tal punto da provocare la perdita di dati, e quindi le prestazioni possono degradare verso quelle della tecnica della rete con perdita di dati: tuttavia, nel caso medio si hanno comunque prestazioni buone. È una tecnica più dispendiosa in termini di complessità e di hardware, ma è vantaggiosa se si assume che mediamente un conflitto non si protraiga per un tempo tale da provocare perdita di dati.

A questo punto, abbiamo fornito una panoramica di quello che è noto come “piano dati” di uno switch. L'ultima osservazione che ci resta da fare riguarda il parallelismo delle linee che interconnettono host e switch, nonché nodi dello switch tra loro. Supponiamo di avere uno switch in grado di interconnettere fino a 2^N host. Se supponiamo di avere un protocollo in cui per ogni messaggio trasmettiamo un bit di abilitazione, N bit per l'indirizzo destinazione, N bit per l'indirizzo sorgente ed M bit di dato, allora ogni messaggio ha una lunghezza pari a $2^N + M + 1$. Dal punto di vista logico, tra ciascun stadio e lo stadio successivo ci sono 2^N linee per i messaggi. In totale ci sono N stadi, più quello di ingresso, quindi in totale $(N + 1) * 2^N$ linee. Se queste linee sono parallele, abbiamo la massima velocità di trasmissione all'interno dello switch, ma abbiamo un totale di $(N + 1) * 2^N * (2^N + M + 1)$ fili hardware. Se le linee sono seriali, allora abbiamo “solo” $(N + 1) * 2^N$ fili hardware, ma è necessario che ogni nodo sia in grado di effettuare una conversione serie-parallelo in fase di ricezione ed una conversione parallelo-serie in fase di invio, e lo deve fare parallelamente su 2 linee in ingresso e 2 linee in uscita, per un totale di 4 shift register. Sarà necessario un numero di flip-flop pari a $2 * N * 2^N * (2^N + M + 1)$; inoltre sarà necessario sincronizzare invio e ricezione con un clock, e quindi in generale ci vuole la logica di periferiche seriali sincrone. Usando fili seriali, si ha solo un apparente risparmio di hardware, perché poi si hanno nodi più complessi sia in termini di hardware sia in termini di logica; inoltre, si è introdotta tale logica *all'interno di una pipeline*: si sa che la velocità di una pipeline è limitata superiormente dalla velocità del suo componente più lento, e la periferica seriale non può andare più veloce del clock, neanche se configurata in maniera sincrona. Dunque, con la configurazione parallela, dato che si tratta di parallelismo hardware si ottiene uno speedup lineare con $2^N + M + 1$, e si ha un dispendio di hardware simile o addirittura minore di quello che si ha con la configurazione seriale. Quindi sicuramente all'interno dello switch conviene tale configurazione. Per quanto riguarda le linee in ingresso e in uscita, ovvero i collegamenti con gli host, non è difficile rendersi conto che o si collega direttamente l'host in parallelo, ed in tal caso non ci sono problemi, oppure si può collegare l'host in maniera seriale e si disaccoppia il funzionamento interno dello switch dal collegamento con gli host mediante componenti che effettuano una sola volta la conversione serie/parallelo in ingresso e parallelo/serie in uscita: in questo modo lo switch lavora alla massima velocità e le periferiche seriali si trovano solo all'interfaccia dello switch (com'è giusto che sia, dato che si chiamano “periferiche”). In altre parole, la periferica seriale che si trova in ingresso trasmette verso lo switch assumendo che esso sia sempre disponibile, e lo switch trasmette in uscita verso la

periferica seriale che si trova in uscita assumendo che essa sia sempre disponibile. Quindi si hanno delle code in ingresso ed in uscita.

10.3 Soluzione con schema a priorità fissa

Il sistema è stato suddiviso in unità operativa e unità di controllo, in particolare, la prima realizza la rete di switch, secondo il modello *Omega Network*, mentre la seconda realizza la rete a priorità. L'unità operativa presenta come ingressi quattro messaggi che i nodi in comunicazione vogliono trasmettere (**i0**, **i1**, **i2**, **i3**), due bit di selezione (**sel_src**, **sel_dest**), forniti dall'unità di controllo per gestire la comunicazione e presenta come uscite i messaggi ricevuti da ciascun nodo (**u0**, **u1**, **u2**, **u3**). Sia i messaggi in ingresso e in uscita che i segnali di selezione sono costituiti da due bit ciascuno, i primi dovuti alla traccia, mentre per quanto riguarda i secondi sono dovuti ai possibili indirizzi dei nodi, che essendo 4 sono rappresentabili su 2 bit.

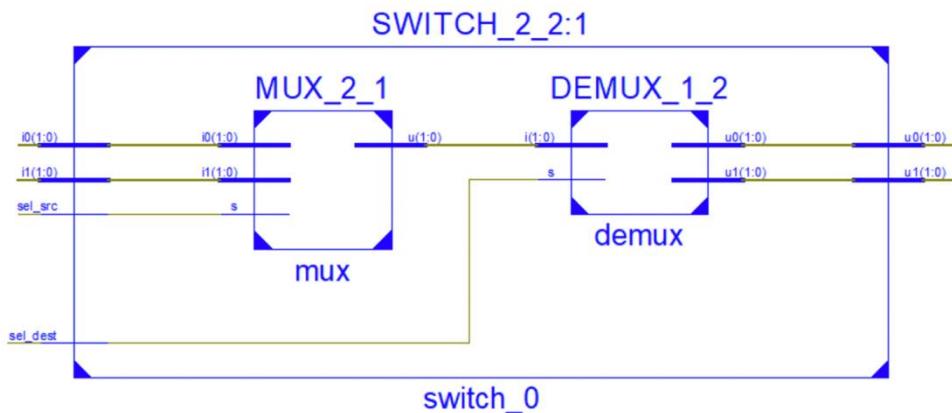


L'elemento alla base della rete è lo switch. Esso è realizzato con un approccio *structural*, tramite la composizione di un multiplexer 2:1 e un demultiplexer 1:2.

Lo switch riceve in ingresso un bit di indirizzo sorgente, **sel_src**, e un bit di indirizzo destinazione **sel_dest**, essi rappresentano rispettivamente il segnale di selezione del multiplexer e il segnale di selezione del demultiplexer.

Lo switch riceve in ingresso il messaggio, **i0** o **i1**, a seconda del bit di selezione del multiplexer, e lo inoltra all'uscita, **u0** o **u1**, a seconda del bit di selezione del demultiplexer.

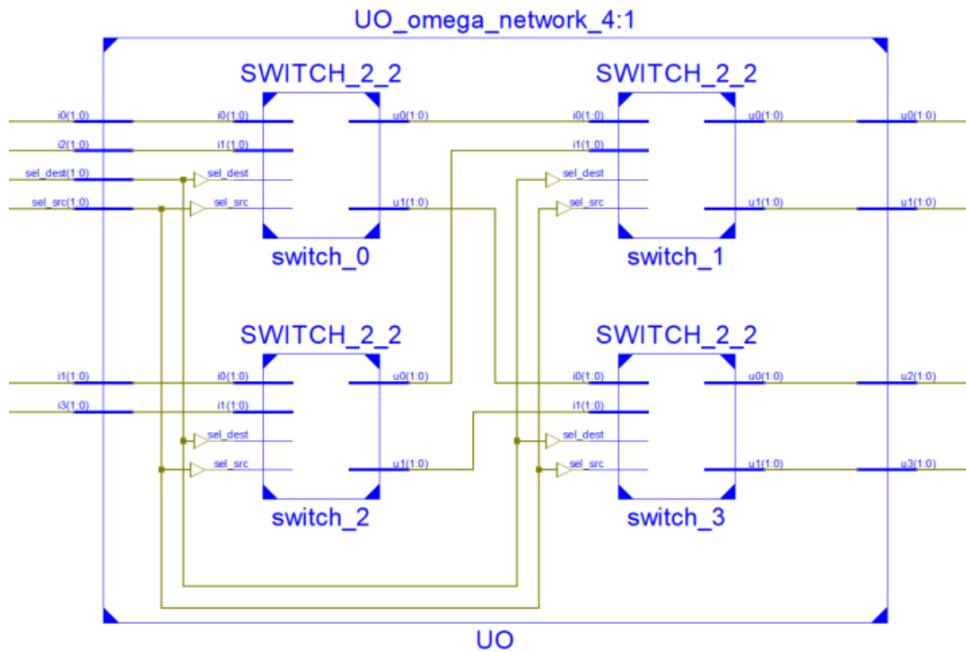
In figura è rappresentato lo schematico dello switch generato.



L'Omega Network è realizzata interconnettendo i quattro nodi seguendo uno schema ottenuto attraverso la tecnica del *Perfect Shuffling*. In particolare, per collegare quattro nodi servono $\log_2(4)$ stadi. La seguente tabella sintetizza le operazioni del *Perfect Shuffling*.

Ordinamento (Stato del mazzo di carte)	Schematizzazione (2 metà del mazzo di carte)	Accoppiamento per stadio
0 1 2 3	I metà del mazzo di carte	I stadio 0 con 2 1 " 3
	0 1	
	II metà del mazzo di carte	
	2 3	
0 2 1 3	I metà del mazzo di carte	II stadio 0 con 1 2 " 3
	0 2	
	II metà del mazzo di carte	
	1 3	
0 1 2 3	I metà del mazzo di carte	Si è ripristinato l'ordine iniziale
	0 1	
	II metà del mazzo di carte	
	2 3	

Si ottiene in questo modo il seguente schema della rete, corrispondente all'unità operativa del sistema.



Dalla figura è possibile osservare che gli ingressi-dato dello *switch_0* sono, rispettivamente, l'uscita del nodo '0' e l'uscita del nodo '2' (i_0 e i_2) e gli ingressi-dato dello *switch_2* sono, rispettivamente, l'uscita del nodo '1' e l'uscita del nodo '3' (i_1 e i_3). Entrambi sono allineati con i risultati della tabella, dato che rappresentano gli switch che compongono lo stadio I della rete. Essi presentano come ingressi di selezione il bit più significativo dell'indirizzo sorgente e dell'indirizzo destinatario forniti in ingresso alla rete dall'unità di controllo.

Per quanto riguarda gli switch dello stadio II essi sono: *switch_1*, il quale presenta come ingressi-dato, rispettivamente, l'uscita '0' dello *switch_0* e l'uscita '0' dello *switch_2*; *switch_3*, il quale presenta come ingressi-dato, rispettivamente, l'uscita '1' dello *switch_0* e l'uscita '1' dello *switch_2*; L'unità di controllo, invece, permette ad un solo nodo alla volta di effettuare la comunicazione, scegliendo il nodo a priorità maggiore. Essa prende in ingresso i nodi che vogliono comunicare, indicati con en_i_0 , en_i_1 , en_i_2 e en_i_3 , e le destinazioni dei messaggi che i quattro nodi vogliono inviare, $dest_i_0$, $dest_i_1$, $dest_i_2$, $dest_i_3$.

Se ad esempio il nodo '0' vuole trasmettere un messaggio al nodo '2', esso, oltre a porre in i_0 il payload del messaggio che vuole effettivamente inviare, deve alzare il bit en_i_0 ($en_i_0 = '1'$) per indicare che vuole trasmettere e deve porre l'indirizzo del nodo '2' in $dest_i_0$, ovvero, $dest_i_0 = "10"$.

Anche l'unità di controllo è implementata con un approccio *structural*, essa è costituita da una rete a priorità e da un multiplexer 4:1.

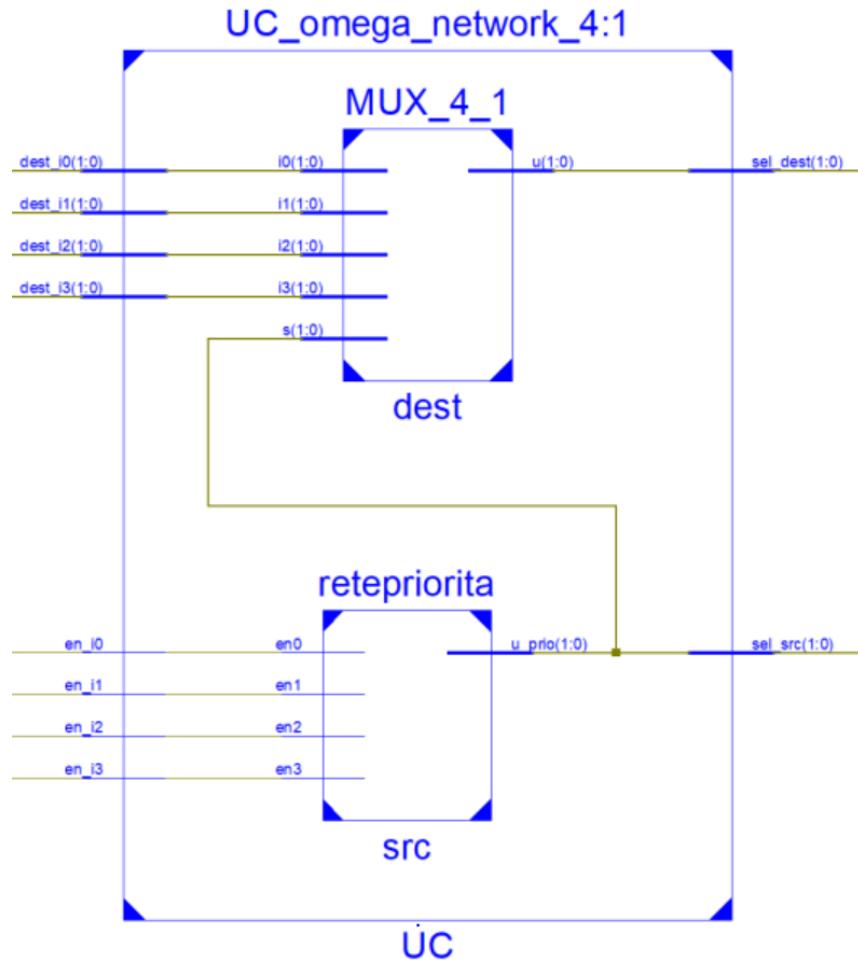
La rete a priorità prende in ingresso i bit relativi ai nodi che vogliono comunicare, en_i_0 , en_i_1 , en_i_2 ed en_i_3 , e trasmette in uscita l'indirizzo del nodo sorgente a maggiore priorità, tra quelle che vogliono trasmettere il proprio messaggio. È stata definita una priorità decrescente, quindi il nodo più prioritario è il nodo '0', mentre quello meno prioritario è il nodo '3'.

Se ad esempio sono alti en_i_1 , en_i_2 e en_i_3 , ovvero vogliono trasmettere sia il nodo '1', che il nodo '2' e il nodo '3', ma non vuole trasmettere il nodo '0', l'indirizzo della sorgente restituito in uscita dalla rete a priorità (sel_src) è l'indirizzo del nodo '1', ovvero "01", indicando che solo questo nodo può procedere alla trasmissione e bloccando gli altri.

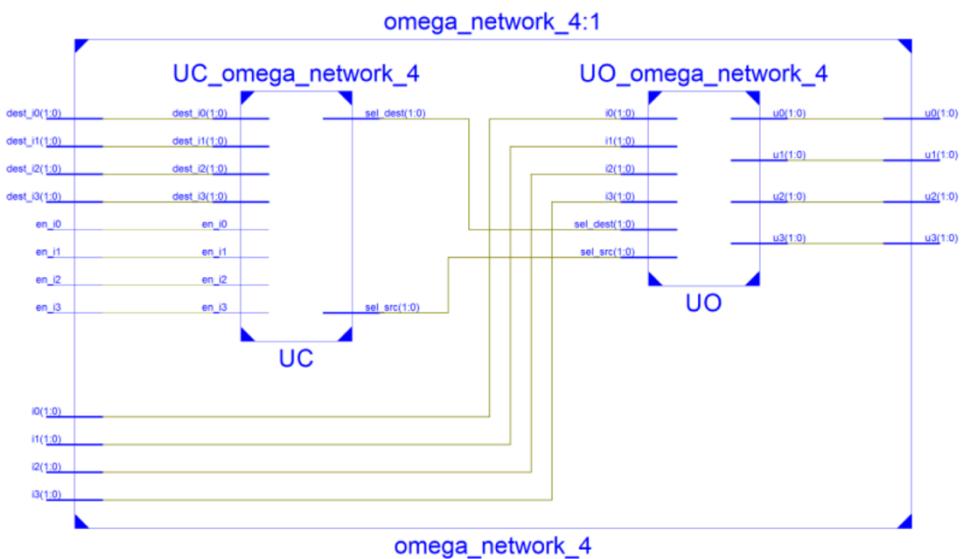
Il multiplexer 4:1 ha il compito di assegnare, sulla base del nodo abilitato alla trasmissione dalla rete a priorità, l'indirizzo destinazione corretto, relativamente al sorgente della trasmissione. Viene implementato questo comportamento prendendo in ingresso tutti gli indirizzi destinazione delle ipotetiche trasmissioni, e restituendo in uscita quello opportuno in base al segnale di selezione del multiplexer. Quest'ultimo è rappresentato dall'indirizzo sorgente abilitato dalla rete a priorità.

Se, ad esempio, viene selezionato il nodo ‘1’ per la trasmissione, in uscita al multiplexer viene restituito il valore assunto da *dest_i1*.

In figura è rappresentato lo schematico della rete di controllo.



Di seguito è raffigurato lo schematico relativo al sistema complessivo.



10.4 Soluzione generic con gestione delle collisioni

Il sistema è realizzato per mezzo di componenti chiamate *node* che implementano al loro interno un'unità operativa ed un'unità di controllo, collegando opportunamente i nodi della rete per mezzo dei *generic* ed il costrutto *for Generate*. La rete è in grado di:

- gestire le collisioni con uno schema prioritario nel singolo nodo, lasciando passare in caso di collisioni sul filo d'uscita sempre l'host con indice minore (es. i messaggi di Host_0 e Host_3 collidono, passa solo il messaggio di Host_0). È adottata dunque una politica di scarto del messaggio (rete con perdita di dati);
- far passare due messaggi in parallelo in uno stesso switch se hanno due destinatari diversi;
- definire in modo arbitrario la dimensione dei pacchetti assegnando un parametro N che definisce la dimensione del pacchetto da trasmettere;
- definire un parametro M che indica il numero di host codificati in binario (es. 8 host = 3 bit, M è di tre bit), dato M in modo implicito sono scelti anche il numero di nodi nella rete essendo questa un'*omega network*.

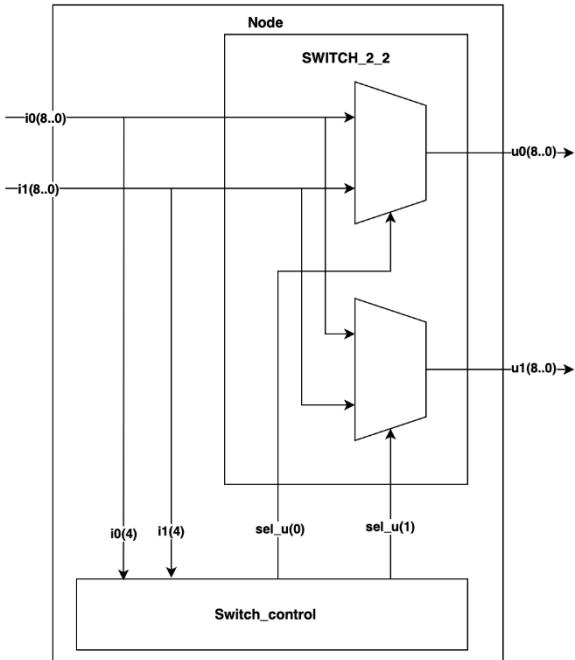
L'architettura così definita è generica e può avere un numero non determinato a priori di nodi interni e un numero non determinato a priori di host, con dimensioni variabili del pacchetto da trasmettere. Per agevolare la trattazione definiamo il numero di host connessi alla rete ed il protocollo usato per la creazione del pacchetto. Si suppone che la rete abbia otto host connessi, dunque gli indirizzi di ogni host sono codificabili su 3 bit, dunque il parametro M sarà pari a tre. Il pacchetto invece è composto da header e payload, nell'header includiamo un bit di abilitazione alla trasmissione attivo alto e senza il quale l'host non può trasmettere, 6 bit dedicati alla specifica dell'indirizzo destinazione e sorgente e due bit per il payload per un totale di 9 bit a pacchetto, dunque il parametro N sarà settato pari a 9 all'interno della rete.

Per capire meglio come è organizzato il pacchetto, supponendo l'interpretazione dei bit little endian, avremo:

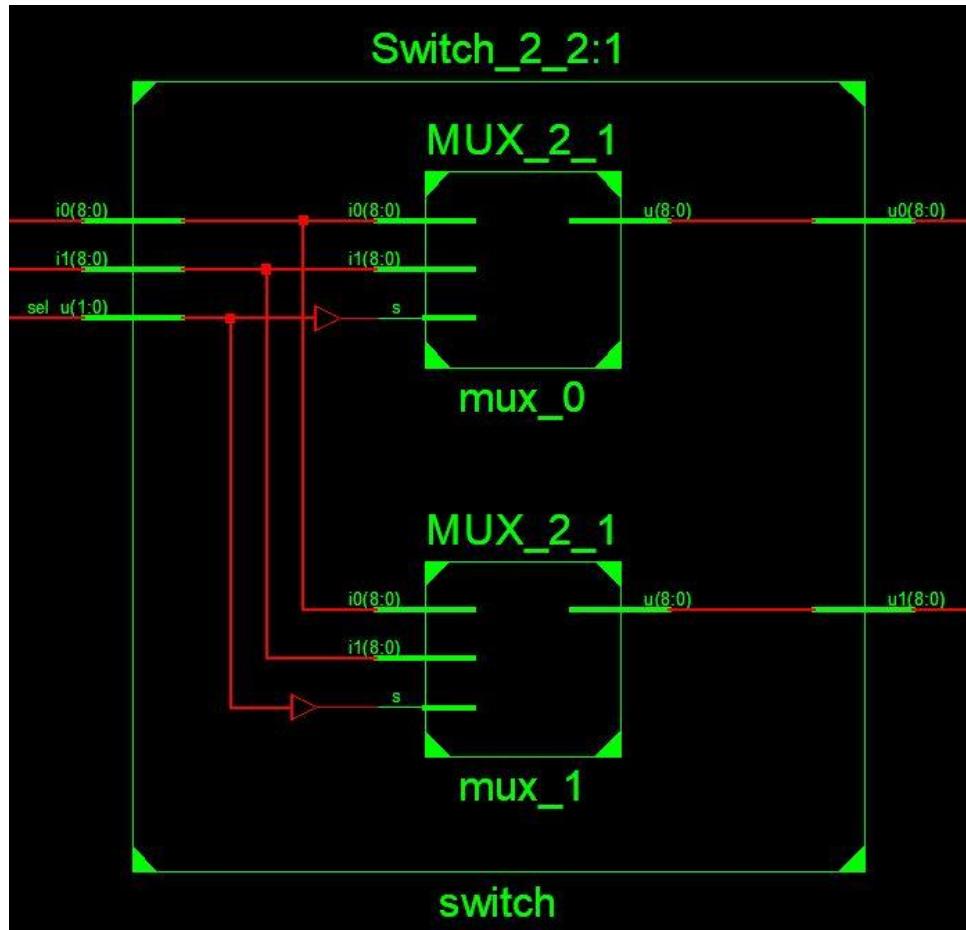
- il bit più significativo ovvero il bit in nona posizione è l'abilitazione;
- i tre bit dall'ottava alla sesta posizione sono l'indirizzo destinazione;
- i tre bit dalla quinta alla terza posizione sono l'indirizzo del mittente;
- i restanti due bit sono il payload.

en_i (1 bit)	dest_i (3 bit)	src_i (3 bit)	payload_i (2 bit)
-----------------	----------------	---------------	-------------------

Vediamo, dunque, come è realizzata la rete partendo dal componente cardine della nostra architettura ovvero il nodo. Come detto disponiamo di un'unità operativa ed un piccolo controllore, come in figura.



Notiamo che in ingresso il nodo riceve il pacchetto da trasmettere ovvero $i0$ e $i1$ ed in uscita avremo $u0$ e $u1$ che propagano i pacchetti in ingresso nella rete o al destinatario stesso, ciò dipende se il nodo è nel layer terminale o nei restanti livelli. L'unità operativa è molto semplice e realizzata con due multiplexer 2:1 che ricevono ognuno i due pacchetti in ingresso $i0$ ed $i1$. Sono pilotati dall'unità di controllo per mezzo del segnale di selezione, un vettore di due bit: con il bit meno significativo si pilota il mux relativo ad $u0$ e con il bit più significativo pilota il mux relativo ad $u1$, come in figura.



In ingresso all'unità di controllo riceviamo quattro segnali da un bit ciascuno, in particolare il bit destinazione di $i0$ ed $i1$ e le rispettive abilitazioni (**dest_i0**, **dest_i1**, **en_i0**, **en_i1**). Le abilitazioni sono prese estraendo dal pacchetto in ingresso solo il bit più significativo, le destinazioni sono prese estraendo il secondo bit più significativo. Per rendere possibile tale schema in ogni nodo è implementata una politica di shift circolare verso sinistra del campo destinazione, di modo che dopo aver attraversato gli M layer della rete i bit della destinazione, che sono M , risultano nell'ordine iniziale e prelevando ad ogni passo il bit più significativo del campo destinazione avremo sempre il bit coerente al livello in cui ci troviamo. Il controllo è gestito da un process che è sensibile a tutti i segnali d'ingresso e valuta il valore delle abilitazioni settando le selezioni da inviare all'unità operativa secondo il seguente schema:

- entrambi gli enable alti e le destinazioni diverse allora setto le selezioni di entrambi i mux, impostando a 0 dove richiedo l'uscita di $i0$ e ad 1 dove richiedo l'uscita di $i1$, pilotando il primo o il secondo bit del vettore di selezione in base al valore del campo dest, se questo è 1 allora piloto la selezione di mux_1 , se è 0 pilota la selezione di mux_0 ;
- entrambi gli enable alti e destinazioni uguali, in questo caso avrò una collisione sul cavo d'uscita che sia questo mux_0 o mux_1 , per risolvere tale problema è adottata una politica di scarto del messaggio secondo uno schema prioritario ovvero ha la precedenza il messaggio ad indice minore in caso di collisione e l'altro messaggio è scartato. Dunque, avremo che in caso di collisione tra $i0$ ed $i1$ passerà sempre $i0$;
- solo un enable alto, setto la selezione del mux puntato dall'indirizzo dest, ovvero se avessi per esempio l' en_i0 alto e dest_i0 pari a 1 allora setto la selezione del mux_1 pari a 0 dato che dovrò far uscire il primo filo, l'altra selezione invece sarà undefined;
- entrambi gli enable bassi allora le selezioni sono poste undefined;

Utilizzando un controllo di questo tipo garantisco tolleranza alle collisioni e consento il parallelismo della trasmissione nel caso in cui due messaggi attraversano un nodo e puntano a due destinatari diversi. Inoltre, settando ad undefined le selezioni quando gli enable sono bassi si manda un segnale non definito al mux dell'unità operativa chiamata *switch* che propagherà un messaggio undefined consentendo la possibilità di resettare la linea facendo risultare in uscita un valore undefined. Nel caso in cui non si abbassino gli enable la rete continua a trasmettere secondo un modello pipelined.

Compreso come è costruito il nodo vediamo come sia possibile unire i nodi in modo da formare una rete generica senza aver bisogno di definire in modo fisso i nodi della rete. Innanzitutto, come anticipato, realizziamo una *omega network* basando l'accoppiamento degli host ai nodi di ogni layer sul concetto di *perfect shuffling*. La rete prevede in ingresso un unico vettore che abbia tutti i pacchetti in ingresso concatenati, dall'host con indice minimo all'host con indice massimo, ovvero i primi N bit dove N è la dimensione del pacchetto sono il messaggio dell'host_0, i secondi N bit rappresentano il messaggio dell'host_1 e così via. In uscita alla rete avremo lo stesso schema dell'ingresso. Di seguito vediamo la tabella che schematizza l'accoppiamento basato sulla tecnica del *perfect shuffling* su 8 nodi.

Ordinamento (Stato del mazzo di carte)	Schematizzazione (2 metà del mazzo di carte)	Accoppiamento per stadio
0 1 2 3 4 5 6 7	I metà del mazzo di carte	I stadio 0 con 4 1 " 5 2 " 6 3 " 7
	0 1 2 3	
	II metà del mazzo di carte	
	4 5 6 7	
0 4 1 5 2 6 3 7	I metà del mazzo di carte	II stadio 0 con 2 4 " 6 1 " 3 5 " 7
	0 4 1 5	
	II metà del mazzo di carte	
	2 6 3 7	
0 2 4 6 1 3 5 7	I metà del mazzo di carte	III stadio 0 con 1 2 " 3 4 " 5 6 " 7
	0 2 4 6	
	II metà del mazzo di carte	
	1 3 5 7	
0 1 2 3 4 5 6 7	I metà del mazzo di carte	Si è ripristinato l'ordinamento iniziale
	0 1 2 3	
	II metà del mazzo di carte	
	4 5 6 7	

Si è risaliti ad una regola generale ricorsiva che definisce come i vettori sono accoppiati notando dunque che ogni nodo ha in ingresso l'i-esimo host della prima metà e l'i-esimo della seconda metà, ovvero l'i-esimo dato più un certo offset pari alla metà del numero di host. Sfruttando questa regola si è realizzata l'*omega network generic* per mezzo di quattro costrutti *for...generate* gestendo opportunamente gli indirizzi ad ogni ciclo. Per aiutarci nella gestione degli indici dei cicli si è utilizzato un programma in Python ad hoc.

```
if __name__ == "__main__":
    m, n = 3, 9
    h = 2**m
    print("i : std_logic_vector({} downto 0)".format(n*h - 1))

    print("u : std_logic_vector({} downto 0)".format(n*h - 1))
    print("u_internal : std_logic_vector({} downto 0)".format(n*h*(m-1) - 1))
    print("Indici del primo stadio (stadio 0)")
```

```

for j in range(0, int(h/2)):
    print("      Switch {}".format(j))
    print("          i0 => i({} downto {}).format((j+1)*n - 1, j*n)")
    print("          i1 => i({} downto {}).format((int(h/2)+j+1)*n - 1,
        (int(h/2)+j)*n)")
    print("          u0 => u_internal({} downto {}).format((2*j+1)*n-1,
        2*j*n)")
    print("          u1 => u_internal({} downto {}).format((2*j+2)*n-1,
        (2*j+1)*n)")
for i in range(1, m-1):
    print("Indici dello stadio {}".format(i))
    for j in range(0, int(h/2)):
        print("      Switch {}".format(j))
        print("          i0 => u_internal({} downto {}).format((j+1)*n - 1 +
            (i-1)*n*h, j*n + (i-1)*n*h)")
        print("          i1 => u_internal({} downto
            {}).format((int(h/2)+j+1)*n - 1 + (i-1)*n*h, (int(h/2)+j)*n +
            (i-1)*n*h)")
        print("          u0 => u_internal({} downto {}).format((2*j+1)*n - 1
            + i*n*h, (2*j)*n + i*n*h)")
        print("          u1 => u_internal({} downto {}).format((2*j+2)*n - 1
            + i*n*h, (2*j+1)*n + i*n*h))")
print("Indici dell'ultimo stadio (stadio m-1)")
for j in range(0, int(h/2)):
    print("      Switch {}".format(j))
    print("          i0 => u_internal({} downto {}).format((j+1)*n - 1 + (m-
        2)*n*h, j*n + (m-2)*n*h)")
    print("          i1 => u_internal({} downto {}).format((int(h/2)+j+1)*n
        - 1 + (m-2)*n*h, (int(h/2)+j)*n + (m-2)*n*h))")
    print("          u0 => u({} downto {}).format((2*j+1)*n-1, 2*j*n)")
    print("          u1 => u({} downto {}).format((2*j+2)*n-1, (2*j+1)*n))")

```

Per proseguire la spiegazione della gestione degli indici faremo riferimento alle variabili in uso in questo codice che sono le stesse dei costrutti vhdl. Si definiscono con n il numero di bit del pacchetto, con m il numero di bit dell'indirizzo destinazione, h indica il numero di host che sono pari a 2^m .

Si passa ora a costruire con un primo *for...generate* i nodi del primo livello, cicliamo su j che va da 0 ad $h/2-1$ dato che dovremo costruire $h/2$ nodi aventi 2 ingressi ciascuno e collegabili ad h host. Nel for avremo gli ingressi del nodo gestiti con la tecnica del *perfect shuffling*, dunque, associamo all'host i-esimo della prima metà l'ingresso *i0* del nodo ed all'host i-esimo della seconda metà l'ingresso *i1*. Avendo i messaggi concatenati ed a lunghezza fissa risulta di semplice gestione il prelievo di un messaggio definendo la distanza n nello sliding dell'assegnazione dal vettore in ingresso al nodo e per differenziare le due entrate basta dare un ulteriore offset $h/2$ all'ingresso del nodo meno prioritario ovvero *i1*. Gli estremi usati nelle assegnazioni degli ingressi ai nodi del primo livello risultano:

- “*i0 => i({} downto {}).format((j+1)*n - 1, j*n)*”, dove j è l'indice del ciclo ed incrementa di volta in volta l'offset relativo del primo ingresso e per distanziare di n l'estremo superiore e inferiore si aggiunge un + 1 all'offset j del limite superiore;
- “*i1 => i({} downto {}).format((int(h/2)+j+1)*n - 1, (int(h/2)+j)*n)*”, la gestione degli indici è la stessa del punto precedente con la differenza che si incrementa di $h/2$ il valore di j ad ogni iterazione in modo da prendere il j-esimo elemento della seconda metà.

Per la gestione delle uscite si è anzitutto usato un signal interno che ha dimensione dipesa dal numero di messaggi trasmessi e dal numero di layer interni: di fatti avremo un vettore di dimensione pari a $((n*2^m)*(m-1))$ dove il primo membro del prodotto indica quanti bit ho ad ogni livello e $m-1$ indica il numero di livelli escluso l'ingresso del primo. Questa osservazione sul segnale interno è di fondamentale importanza per capire come sono gestite le assegnazioni interne. Prima di passare ai layer interni concludiamo la trattazione sul primo livello che assegna le uscite concatenando semplicemente $u0$ ed $u1$ con le seguenti regole:

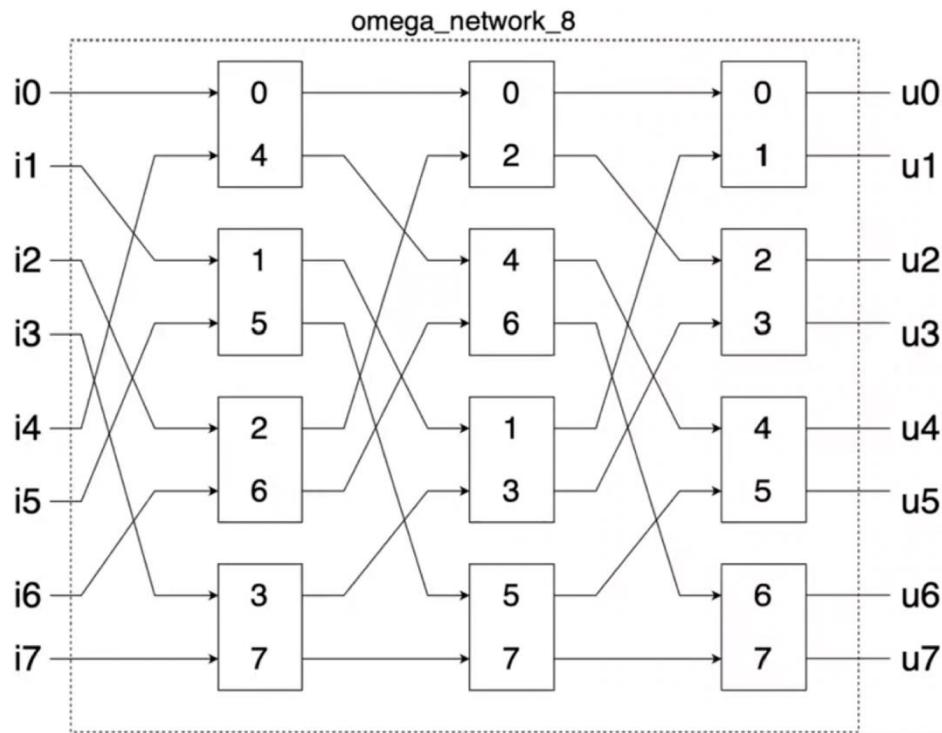
- “ $u0 \Rightarrow u_internal(\{\} \text{ downto } \{\})$ ”.format($(2*j+1)*n-1, 2*j*n$)), anche qui la gestione degli indici è dettata dalla gestione degli offset da moltiplicare a j , per consentire il salto di un messaggio tra un ciclo ed un altro: dato che ad ogni ciclo assegno $2n$ bit basta moltiplicare per due il fattore j ; per garantire la distanza tra i due estremi di n sommo $+1$ all'offset moltiplicativo del limite superiore;
- “ $u1 \Rightarrow u_internal(\{\} \text{ downto } \{\})$ ”.format($(2*j+2)*n-1, (2*j+1)*n$)), stesso discorso vale anche per l'uscita $u1$ con la differenza che gli offset moltiplicativi hanno un $+2$ per il limite superiore dato che scrive sui bit che vanno dall'indice $2n$ ad n e il limite inferiore ha un offset di $+1$.

Al termine del ciclo avremo generato con gli indici giusti il primo livello della rete. Passiamo ora alla gestione degli indici dei layer interni; si sono usati due cicli, uno esterno che indicizza il livello, indicando in quale ci troviamo, uno interno che invece indicizza il nodo indicando quale nodo stiamo creando su quale riga della colonna i -esima. Si capisce dunque che j , indice interno, cicla sugli stessi valori del for precedente e i , indice esterno, invece cicla sui valori da 1 ad $m-2$ dato che il primo e l'ultimo livello sono collegati con l'interfaccia della rete e necessitano un trattamento dedicato. La gestione degli indici è molto simile al primo for per cui capiamo come sono stati sfruttati i segnali interni gestendo in maniera opportuna gli offset:

- “ $i0 \Rightarrow u_internal(\{\} \text{ downto } \{\})$ ”.format($((j+1)*n - 1 + (i-1)*n*h, j*n + (i-1)*n*h)$), notiamo che la prima parte degli indici, sia dell'estremo superiore che inferiore, sono le stesse dell'ingresso $i0$ del primo livello. Notiamo che è presente un ulteriore offset additivo relativo al livello, di fatti essendo all' i -esimo layer interno gli ingressi saranno le uscite del livello i meno uno e dunque sommo ad entrambi gli estremi un offset pari alla dimensione dei messaggi del layer per $i - 1$;
- “ $i1 \Rightarrow u_internal(\{\} \text{ downto } \{\})$ ”.format($((int(h/2)+j+1)*n - 1 + (i-1)*n*h, (int(h/2)+j)*n + (i-1)*n*h)$), per l'ingresso $i1$ di un layer interno valgono le stesse considerazioni del primo layer e le considerazioni sull'offset additivo del punto precedente;
- “ $u0 \Rightarrow u_internal(\{\} \text{ downto } \{\})$ ”.format($((2*j+1)*n - 1 + i*n*h, (2*j)*n + i*n*h)$), l'uscita è trattata allo stesso modo del primo livello e l'offset additivo, dato che assegniamo le uscite del livello i -esimo, deve avere un offset additivo di $i*n*h$ e non $i - 1$;
- “ $u1 \Rightarrow u_internal(\{\} \text{ downto } \{\})$ ”.format($((2*j+2)*n - 1 + i*n*h, (2*j+1)*n + i*n*h)$), valgono le considerazioni sugli estremi fatte per il primo livello e per l'offset additivo valgono le considerazioni del punto precedente.

La gestione degli indici dell'ultimo livello è la stessa dei livelli intermedi notando che l'offset additivo in ingresso, provenendo dal layer $m-2$, sarà $m-2*n*h$, mentre in uscita avremo il livello ultimo ovvero $m-1$ per cui l'offset additivo è $m-1*n*h$.

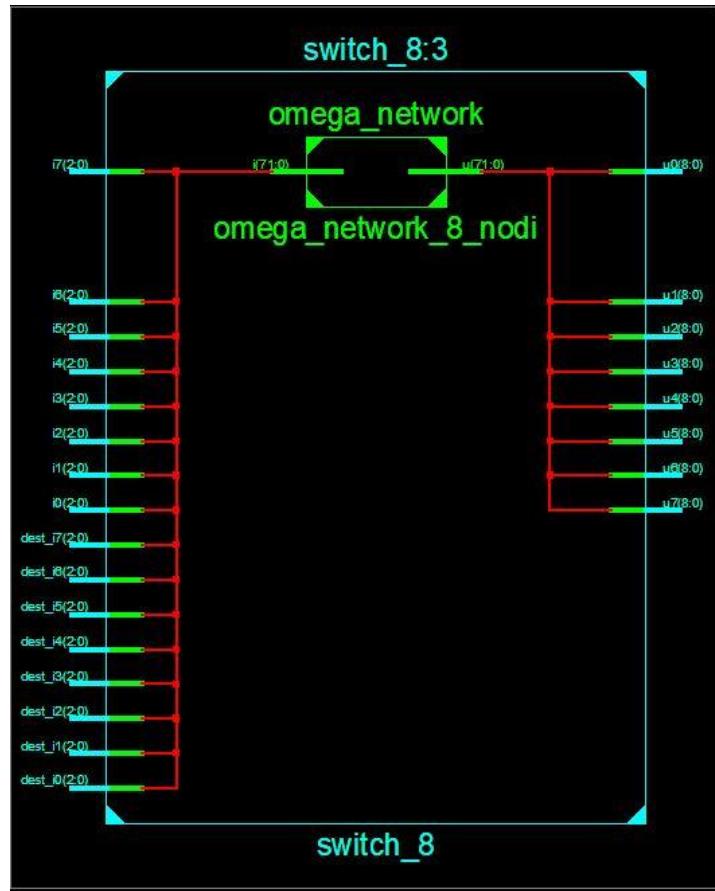
Con questo approccio generico possiamo gestire, come anticipato, una qualunque rete con un numero di host arbitrario e pacchetti non strettamente vincolati al protocollo definito come esempio. Vediamo come si presenta dunque la rete d'esempio ad 8 host.



Per facilitare la gestione dei casi di test abbiamo creato un ulteriore modulo chiamato *switch_8* che ingloba la rete omega e riceve in ingresso in modo parallelo i messaggi con i relativi destinatari e rende in uscita i pacchetti divisi per host e non concatenati come nella rete interna. Il modulo, dunque, prevede:

- di concatenare i dati in ingresso assegnando in modo statico gli indirizzi sorgente ad ogni pacchetto dati, secondo il protocollo definito precedentemente;
- di smistare i bit in uscita alla rete ai rispettivi host che leggeranno l'intero pacchetto comprensivo di header e payload.

L’interfaccia generale si presenta come in figura.



10.5 Codice

10.5.1 Codice rete a priorità

Di seguito vi è il codice relativo al singolo componente della unità operativa, lo **switch**.

```

entity SWITCH_2_2 is
Generic ( N : integer := 2 );
Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
      i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
      sel_src : in STD_LOGIC;
      sel_dest : in STD_LOGIC;
      u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
      u1 : out STD_LOGIC_VECTOR(N-1 downto 0));
end SWITCH_2_2;

architecture Structural of SWITCH_2_2 is

component MUX_2_1 is
Generic ( N : integer := 2 );
Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
      i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
      s : in STD_LOGIC;
      u : out STD_LOGIC_VECTOR(N-1 downto 0));
end component;

```

```

component DEMUX_1_2 is
Generic ( N : integer := 2 );
  Port ( i : in STD_LOGIC_VECTOR(N-1 downto 0);
         s : in STD_LOGIC;
         u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
         u1 : out STD_LOGIC_VECTOR(N-1 downto 0));
end component;

signal i : STD_LOGIC_VECTOR(N-1 downto 0) := (others => '0');

begin
mux: MUX_2_1
generic map(2)
port map(i0 => i0, i1 => i1, s => sel_src, u => i);
demux: DEMUX_1_2
generic map(2)
port map(i => i, s => sel_dest, u0 => u0, u1 => u1);
end Structural;

```

Di seguito vi sono i codici relativi ai componenti elementari dello switch, ovvero il multiplexer 2:1 e il demultiplexer 1:2, implementati con un approccio *dataflow*.

```

entity MUX_2_1 is
Generic ( N : integer := 2 );
  Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
         i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
         s : in STD_LOGIC;
         u : out STD_LOGIC_VECTOR(N-1 downto 0));
end MUX_2_1;

architecture Dataflow of MUX_2_1 is

begin
with s select
u <= i0 when '0',
i1 when '1',
(others => '-') when others;

end Dataflow;

entity DEMUX_1_2 is
Generic ( N : integer := 2 );
  Port ( i : in STD_LOGIC_VECTOR(N-1 downto 0);
         s : in STD_LOGIC;
         u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
         u1 : out STD_LOGIC_VECTOR(N-1 downto 0));
end DEMUX_1_2;

architecture Dataflow of DEMUX_1_2 is

begin
u0 <= i when s = '0' else (others => '-');
u1 <= i when s = '1' else (others => '-');

```

```
end Dataflow;
```

Di seguito vi è il codice relativo all'unità operativa del sistema, rappresentante la rete di switch.

```
entity U0_omega_network_4 is
Generic (N : integer := 2);
Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
      i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
      i2 : in STD_LOGIC_VECTOR(N-1 downto 0);
      i3 : in STD_LOGIC_VECTOR(N-1 downto 0);
      sel_src : in STD_LOGIC_VECTOR(1 downto 0);
      sel_dest : in STD_LOGIC_VECTOR(1 downto 0);
      u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
      u1 : out STD_LOGIC_VECTOR(N-1 downto 0);
      u2 : out STD_LOGIC_VECTOR(N-1 downto 0);
      u3 : out STD_LOGIC_VECTOR(N-1 downto 0));
end U0_omega_network_4;

architecture Structural of U0_omega_network_4 is

component SWITCH_2_2 is
Generic ( N : integer := 2 );
Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
       i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
       sel_src : in STD_LOGIC;
       sel_dest : in STD_LOGIC;
       u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
       u1 : out STD_LOGIC_VECTOR(N-1 downto 0));
end component;

signal i0_1, i1_1, i0_3, i1_3 : STD_LOGIC_VECTOR(N-1 downto 0) := (others => '0');

begin
switch_0: SWITCH_2_2 generic map(N)
port map(i0 => i0, i1 => i2, sel_src => sel_src(1), sel_dest => sel_dest(1), u0
=> i0_1, u1 => i0_3);
switch_1: SWITCH_2_2 generic map(N)
port map(i0 => i0_1, i1 => i1_1, sel_src => sel_src(0), sel_dest => sel_dest(0),
u0 => u0 ,u1 => u1);
switch_2: SWITCH_2_2 generic map(N)
port map(i0 => i1, i1 => i3, sel_src => sel_src(1), sel_dest => sel_dest(1),u0
=> i1_1 ,u1 => i1_3 );
switch_3: SWITCH_2_2 generic map(N)
port map(i0 => i0_3, i1 => i1_3, sel_src => sel_src(0), sel_dest => sel_dest(0),
u0 => u2, u1 => u3);
end Structural;
```

Di seguito vi è il codice relativo all'unità di controllo del sistema.

```
entity UC_omega_network_4 is
  Port ( en_i0 : in STD_LOGIC;
         en_i1 : in STD_LOGIC;
         en_i2 : in STD_LOGIC;
```

```

        en_i3 : in STD_LOGIC;
        dest_i0 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
        dest_i1 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
        dest_i2 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
        dest_i3 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
        sel_src : out STD_LOGIC_VECTOR(1 DOWNTO 0);
        sel_dest : out STD_LOGIC_VECTOR(1 DOWNTO 0));
end UC_omega_network_4;

architecture Structural of UC_omega_network_4 is

component retepriorita is
    Port ( en0:in STD_LOGIC;
           en1:in STD_LOGIC;
           en2:in STD_LOGIC;
           en3:in STD_LOGIC;
           u_prio:out STD_LOGIC_VECTOR(1 downto 0));
end component;

component MUX_4_1 is
    Port ( i0 :in STD_LOGIC_VECTOR(1 downto 0);
           i1 :in STD_LOGIC_VECTOR(1 downto 0);
           i2 :in STD_LOGIC_VECTOR(1 downto 0);
           i3:in STD_LOGIC_VECTOR(1 downto 0);
           s :in STD_LOGIC_VECTOR(1 downto 0);
           u :out STD_LOGIC_VECTOR(1 downto 0));
end component;

signal internal_sel_src :STD_LOGIC_VECTOR(1 DOWNTO 0) := (others => '0');

begin

src: retepriorita
Port map(en0 => en_i0, en1 => en_i1, en2 => en_i2, en3 => en_i3, u_prio => internal_sel_src);

dest: MUX_4_1
Port map(i0 => dest_i0, i1 => dest_i1, i2 => dest_i2, i3 => dest_i3, s => internal_sel_src, u => sel_dest);

sel_src <= internal_sel_src;

end Structural;

```

Di seguito vi è il codice relativo ai componenti dell'unità di controllo, la rete a priorità, implementata con un approccio *dataflow*.

```

entity retepriorita is
    Port ( en0 : in STD_LOGIC;
           en1 : in STD_LOGIC;
           en2 : in STD_LOGIC;
           en3 : in STD_LOGIC;
           u_prio : out STD_LOGIC_VECTOR(1 downto 0));
end retepriorita;

```

```

architecture Dataflow of retepriorita is

begin
u_prio <= "00" when en0 = '1' else
"01" when en1 = '1' else
"10" when en2 = '1' else
"11" when en3 = '1' else
"--";
end Dataflow;

```

Il codice relativo al multiplexer4:1, anch'esso implementato con un approccio *dataflow*.

```

entity MUX_4_1 is
  Port ( i0 : in STD_LOGIC_VECTOR(1 downto 0);
         i1 : in STD_LOGIC_VECTOR(1 downto 0);
         i2 : in STD_LOGIC_VECTOR(1 downto 0);
         i3 : in STD_LOGIC_VECTOR(1 downto 0);
         s : in STD_LOGIC_VECTOR(1 downto 0);
         u : out STD_LOGIC_VECTOR(1 downto 0));
end MUX_4_1;

```

```

architecture Dataflow of MUX_4_1 is

begin
with s select
u <= i0 when "00",
  i1 when "01",
  i2 when "10",
  i3 when "11",
  (others => '-') when others;
end Dataflow;

```

Infine, vi è il codice relativo al sistema complessivo, composto dall'unità operativa e dall'unità di controllo.

```

entity omega_network_4 is
Generic (N : integer := 2);
  Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
         i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
         i2 : in STD_LOGIC_VECTOR(N-1 downto 0);
         i3 : in STD_LOGIC_VECTOR(N-1 downto 0);
         en_i0 : in STD_LOGIC;
         en_i1 : in STD_LOGIC;
         en_i2 : in STD_LOGIC;
         en_i3 : in STD_LOGIC;
         dest_i0 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
         dest_i1 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
         dest_i2 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
         dest_i3 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
         u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
         u1 : out STD_LOGIC_VECTOR(N-1 downto 0);
         u2 : out STD_LOGIC_VECTOR(N-1 downto 0);
         u3 : out STD_LOGIC_VECTOR(N-1 downto 0));

```

```

end omega_network_4;

architecture Structural of omega_network_4 is
component U0_omega_network_4 is
Generic (N : integer := 2);
  Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
         i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
         i2 : in STD_LOGIC_VECTOR(N-1 downto 0);
         i3 : in STD_LOGIC_VECTOR(N-1 downto 0);
         sel_src : in STD_LOGIC_VECTOR(1 downto 0);
         sel_dest : in STD_LOGIC_VECTOR(1 downto 0);
         u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
         u1 : out STD_LOGIC_VECTOR(N-1 downto 0);
         u2 : out STD_LOGIC_VECTOR(N-1 downto 0);
         u3 : out STD_LOGIC_VECTOR(N-1 downto 0));
end component;

component UC_omega_network_4 is
  Port ( en_i0 : in STD_LOGIC;
         en_i1 : in STD_LOGIC;
         en_i2 : in STD_LOGIC;
         en_i3 : in STD_LOGIC;
         dest_i0 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
         dest_i1 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
         dest_i2 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
         dest_i3 : in STD_LOGIC_VECTOR(1 DOWNTO 0);
         sel_src : out STD_LOGIC_VECTOR(1 DOWNTO 0);
         sel_dest : out STD_LOGIC_VECTOR(1 DOWNTO 0));
end component;

signal sel_src : STD_LOGIC_VECTOR(1 DOWNTO 0) := (others => '0');
signal sel_dest : STD_LOGIC_VECTOR(1 DOWNTO 0) := (others => '0');

begin
U0: U0_omega_network_4
Generic map(N)
Port map(i0 => i0, i1 => i1, i2 => i2, i3 => i3, sel_src => sel_src, sel_dest => sel_dest, u0 => u0, u1 => u1, u2 => u2, u3 => u3);

UC: UC_omega_network_4
Port map(en_i0 => en_i0, en_i1 => en_i1, en_i2 => en_i2, en_i3 => en_i3, dest_i0 => dest_i0, dest_i1 => dest_i1, dest_i2 => dest_i2, dest_i3 => dest_i3, sel_src => sel_src, sel_dest => sel_dest);

end Structural;

```

Per la simulazione e la verifica del corretto funzionamento del sistema è stato implementato un *testbench*, il cui codice è mostrato di seguito.

```

ENTITY omega_network_4_tb IS
END omega_network_4_tb;

ARCHITECTURE behavior OF omega_network_4_tb IS

```

```

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT omega_network_4
PORT(
    i0 : IN std_logic_vector(1 downto 0);
    i1 : IN std_logic_vector(1 downto 0);
    i2 : IN std_logic_vector(1 downto 0);
    i3 : IN std_logic_vector(1 downto 0);
    en_i0 : IN std_logic;
    en_i1 : IN std_logic;
    en_i2 : IN std_logic;
    en_i3 : IN std_logic;
    dest_i0 : IN std_logic_vector(1 downto 0);
    dest_i1 : IN std_logic_vector(1 downto 0);
    dest_i2 : IN std_logic_vector(1 downto 0);
    dest_i3 : IN std_logic_vector(1 downto 0);
    u0 : OUT std_logic_vector(1 downto 0);
    u1 : OUT std_logic_vector(1 downto 0);
    u2 : OUT std_logic_vector(1 downto 0);
    u3 : OUT std_logic_vector(1 downto 0)
);
END COMPONENT;

--Inputs
signal i0 : std_logic_vector(1 downto 0) := (others => '0');
signal i1 : std_logic_vector(1 downto 0) := (others => '0');
signal i2 : std_logic_vector(1 downto 0) := (others => '0');
signal i3 : std_logic_vector(1 downto 0) := (others => '0');
signal en_i0 : std_logic := '0';
signal en_i1 : std_logic := '0';
signal en_i2 : std_logic := '0';
signal en_i3 : std_logic := '0';
signal dest_i0 : std_logic_vector(1 downto 0) := (others => '0');
signal dest_i1 : std_logic_vector(1 downto 0) := (others => '0');
signal dest_i2 : std_logic_vector(1 downto 0) := (others => '0');
signal dest_i3 : std_logic_vector(1 downto 0) := (others => '0');

--Outputs
signal u0 : std_logic_vector(1 downto 0);
signal u1 : std_logic_vector(1 downto 0);
signal u2 : std_logic_vector(1 downto 0);
signal u3 : std_logic_vector(1 downto 0);
-- No clocks detected in port list. Replace <clock> below with
-- appropriate port name

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: omega_network_4 PORT MAP (
    i0 => i0,
    i1 => i1,
    i2 => i2,
    i3 => i3,

```

```

    en_i0 => en_i0,
    en_i1 => en_i1,
    en_i2 => en_i2,
    en_i3 => en_i3,
    dest_i0 => dest_i0,
    dest_i1 => dest_i1,
    dest_i2 => dest_i2,
    dest_i3 => dest_i3,
    u0 => u0,
    u1 => u1,
    u2 => u2,
    u3 => u3
);

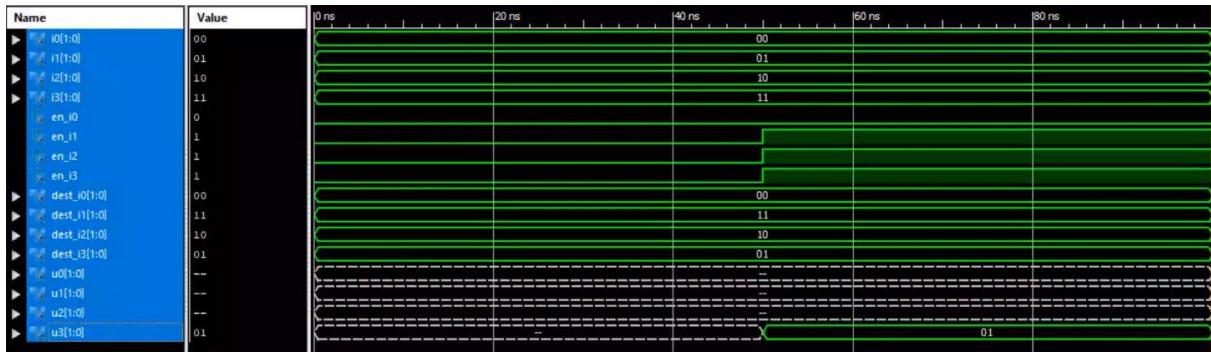
-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100 ns.
i0 <= "00";
i1 <= "01";
i2 <= "10";
i3 <= "11";
dest_i0 <= "00";
dest_i1 <= "11";
dest_i2 <= "10";
dest_i3 <= "01";
    wait for 50 ns;
en_i1 <= '1';
en_i2 <= '1';
en_i3 <= '1';
wait for 50 ns;
assert u3 = "01";
report "errore0"
severity failure;

    -- insert stimulus here
wait;
    wait;
end process;

END;

```

I risultati della simulazione sono riportati nella seguente figura.



È possibile osservare come i nodi ‘1’, ‘2’ e ‘3’ vogliono trasmettere il proprio messaggio ma la rete a priorità schedula il nodo ‘1’, essendo a maggiore priorità, e quindi solo il proprio messaggio arriva a destinazione, infatti, al termine della simulazione, il nodo destinazione, nodo ‘3’, riceve il messaggio “01” dal nodo ‘1’.

10.5.2 Codice rete generic

Di seguito è riportato il codice della rete implementata con approccio generic.

Il primo modulo è switch_8 che è realizzato con approccio structural.

```
entity switch_8 is
Generic(N: integer :=3);
  Port ( i0 : in STD_LOGIC_VECTOR ( N-1 DOWNTO 0 );
         i1 : in STD_LOGIC_VECTOR ( N-1 DOWNTO 0 );
         i2 : in STD_LOGIC_VECTOR ( N-1 DOWNTO 0 );
         i3 : in STD_LOGIC_VECTOR ( N-1 DOWNTO 0 );
         i4 : in STD_LOGIC_VECTOR ( N-1 DOWNTO 0 );
         i5 : in STD_LOGIC_VECTOR ( N-1 DOWNTO 0 );
         i6 : in STD_LOGIC_VECTOR ( N-1 DOWNTO 0 );
         i7 : in STD_LOGIC_VECTOR ( N-1 DOWNTO 0 );
dest_i0 : in STD_LOGIC_VECTOR ( 2 DOWNTO 0 );
         dest_i1 : in STD_LOGIC_VECTOR ( 2 DOWNTO 0 );
         dest_i2 : in STD_LOGIC_VECTOR ( 2 DOWNTO 0 );
         dest_i3 : in STD_LOGIC_VECTOR ( 2 DOWNTO 0 );
         dest_i4 : in STD_LOGIC_VECTOR ( 2 DOWNTO 0 );
         dest_i5 : in STD_LOGIC_VECTOR ( 2 DOWNTO 0 );
         dest_i6 : in STD_LOGIC_VECTOR ( 2 DOWNTO 0 );
         dest_i7 : in STD_LOGIC_VECTOR ( 2 DOWNTO 0 );
u0 : out STD_LOGIC_VECTOR ( N+6-1 DOWNTO 0 );
         u1 : out STD_LOGIC_VECTOR ( N+6-1 DOWNTO 0 );
         u2 : out STD_LOGIC_VECTOR ( N+6-1 DOWNTO 0 );
         u3 : out STD_LOGIC_VECTOR ( N+6-1 DOWNTO 0 );
         u4 : out STD_LOGIC_VECTOR ( N+6-1 DOWNTO 0 );
         u5 : out STD_LOGIC_VECTOR ( N+6-1 DOWNTO 0 );
         u6 : out STD_LOGIC_VECTOR ( N+6-1 DOWNTO 0 );
```

```

        u7 : out STD_LOGIC_VECTOR ( N+6-1 DOWNTO 0 ));
end switch_8;

architecture structural of switch_8 is

component omega_network is
Generic(N: integer :=6; M: integer :=3);
    Port ( i : in STD_LOGIC_VECTOR(N*2**M - 1 downto 0);
           u : out STD_LOGIC_VECTOR(N*2**M - 1 downto 0));
end component;
signal i,exit_u : std_logic_vector ((N+6)*2**3 - 1 downto 0):= (others =>
'0');

begin

i <= i7(N-1)& dest_i7 & "111" & i7(N-2 downto 0) & i6(N-1) & dest_i6 &
"110" & i6(N-2 downto 0) & i5(N-1) & dest_i5 & "101" & i5(N-2 downto 0) &
i4(N-1) & dest_i4 & "100" & i4(N-2 downto 0) & i3(N-1) & dest_i3 & "011"
& i3(N-2 downto 0) & i2(N-1) & dest_i2 & "010" & i2(N-2 downto 0) & i1(N-
1) & dest_i1 & "001" & i1(N-2 downto 0) & i0(N-1) & dest_i0 & "000" &
i0(N-2 downto 0);
omega_network_8_nodi : omega_network generic map (N+6,3)
Port map(
i => i,
u => exit_u);

u7 <= exit_u((N+6)*8-1 downto (N+6)*7);
u6 <= exit_u((N+6)*7-1 downto (N+6)*6);
u5 <= exit_u((N+6)*6-1 downto (N+6)*5);
u4 <= exit_u((N+6)*5-1 downto (N+6)*4);
u3 <= exit_u((N+6)*4-1 downto (N+6)*3);
u2 <= exit_u((N+6)*3-1 downto (N+6)*2);
u1 <= exit_u((N+6)*2-1 downto (N+6)*1);
u0 <= exit_u((N+6)-1 downto 0);

end structural;

```

Il modulo di seguito è la *omega_network* implementata con approccio structural.

```

entity omega_network is
Generic(N: integer :=6; M: integer :=3);
    Port ( i : in STD_LOGIC_VECTOR(N*2**M - 1 downto 0);
           u : out STD_LOGIC_VECTOR(N*2**M - 1 downto 0));
end omega_network;

architecture Structural of omega_network is
component node is
Generic(N : integer := 6; M : integer := 3);
    Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
           i1 : in STD_LOGIC_VECTOR(N-1 downto 0);

```

```

        u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
        u1 : out STD_LOGIC_VECTOR(N-1 downto 0));
end component;

constant h : integer := 2**M;
signal u_internal: STD_LOGIC_VECTOR(((N*2**M)*(M-1) - 1) downto 0);
begin
nodi_c1: for j in 0 to (h/2-1) generate
    nodo_j_0: node Generic map(N,M)
        Port map(i0 => i(((j+1)*N-1) downto (j*N)),
                  i1 => i(((h/2+j+1)*N-1) downto ((h/2+j)*N)),
                  u0 => u_internal(((2*j+1)*N-1) downto (2*j*N)),
                  u1 => u_internal(((2*j+2)*N-1) downto ((2*j+1)*N)));
end generate;
nodi_r: for i in 1 to M - 2 generate
    nodi_c:for j in 0 to (h/2-1) generate
        nodo_j_i: node Generic map(N,M)
            Port map(i0 => u_internal(((j+1)*N-1 + (i-1)*(N*2**M)) downto
                (j*N) + (i-1)*(N*2**M)),
                      i1 => u_internal(((h/2+j+1)*N-1 + (i-1)*(N*2**M)) downto
                ((h/2+j)*N) + (i-1)*(N*2**M)),
                      u0 => u_internal(((2*j+1)*N-1 + (i)*(N*2**M)) downto (2*j*N) +
                (i)*(N*2**M)),
                      u1 => u_internal(((2*j+2)*N-1 + (i)*(N*2**M)) downto
                ((2*j+1)*N) + (i)*(N*2**M)));
            end generate;
    end generate;
nodi_cM_1: for j in 0 to (h/2-1) generate
    nodo_j_M_1: node Generic map(N,M)
        Port map(i0 => u_internal(((j+1)*N-1 + (M-2)*(N*2**M)) downto (j*N) +
                (M-2)*(N*2**M)),
                  i1 => u_internal(((h/2+j+1)*N-1 + (M-2)*(N*2**M)) downto
                ((h/2+j)*N) + (M-2)*(N*2**M)),
                  u0 => u(((2*j+1)*N-1) downto (2*j*N)),
                  u1 => u(((2*j+2)*N-1) downto ((2*j+1)*N)));
    end generate;
end Structural;

```

Di seguito abbiamo il codice del modulo *node* realizzato con un approccio strutturale.

```

entity node is
Generic(N : integer := 6; M : integer := 3);
    Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
           i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
           u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
           u1 : out STD_LOGIC_VECTOR(N-1 downto 0));
end node;

architecture Structural of node is
component Switch_2_2 is

```

```

Generic(N : integer := 6; M : integer := 3);
  Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
          i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
          sel_u : in STD_LOGIC_VECTOR(1 downto 0);
          u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
          u1 : out STD_LOGIC_VECTOR(N-1 downto 0));
end component;
component switch_control is
  Port ( en_i0 : in STD_LOGIC;
          en_i1 : in STD_LOGIC;
          dest_i0 : in STD_LOGIC;
          dest_i1 : in STD_LOGIC;
          sel_u : out STD_LOGIC_VECTOR(1 downto 0));
end component;

signal sel_u : STD_LOGIC_VECTOR(1 downto 0) := (others => '0');
begin

switch: Switch_2_2 Generic map(N,M)
Port map(i0 => i0, i1 => i1, sel_u => sel_u, u0 => u0, u1 => u1);

control: switch_control
Port map(en_i0 => i0(N-1), en_i1 => i1(N-1), dest_i0 => i0(N-2), dest_i1
=> i1(N-2), sel_u => sel_u);

end Structural;

```

Di seguito abbiamo il modulo switch_control realizzato con un approccio behavioral.

```

entity switch_control is
  Port ( en_i0 : in STD_LOGIC;
          en_i1 : in STD_LOGIC;
          dest_i0 : in STD_LOGIC;
          dest_i1 : in STD_LOGIC;
          sel_u : out STD_LOGIC_VECTOR(1 downto 0));
end switch_control;

architecture Behavioral of switch_control is

begin
UC: process (en_i0, en_i1, dest_i0, dest_i1)
  variable d_i0, d_i1 : integer range 0 to 1 := 0;
  begin
    if dest_i0 = '1' then
      d_i0 := 1;
    else
      d_i0 := 0;
    end if;
    sel_u <= "--";
    if dest_i1 = '1' then

```

```

        d_i1 := 1;
    else
        d_i1 := 0;
    end if;
    if en_i0='1' and en_i1='1' then
        if d_i0 /= d_i1 then -- /= operatore "not equal"
            sel_u(d_i1) <= '1';
        end if;
        sel_u(d_i0) <= '0';
    elsif en_i0 = '1' then
        sel_u(d_i0) <= '0';
    elsif en_i1 = '1' then
        sel_u(d_i1) <= '1';
    end if;
end process;

end Behavioral;

```

Di seguito abbiamo il codice del modulo *switch* implementato con approccio ibrido, uso delle componenti e gestione dataflow dello shift circolare del campo destinazione.

```

entity Switch_2_2 is
Generic(N : integer := 6; M : integer := 3);
    Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
           i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
           sel_u : in STD_LOGIC_VECTOR(1 downto 0);
           u0 : out STD_LOGIC_VECTOR(N-1 downto 0);
           u1 : out STD_LOGIC_VECTOR(N-1 downto 0));
end Switch_2_2;

architecture ibrido of Switch_2_2 is
component MUX_2_1 is
Generic ( N : integer := 2 );
    Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
           i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
           s : in STD_LOGIC;
           u : out STD_LOGIC_VECTOR(N-1 downto 0));
end component;

signal shifted_0, shifted_1 : STD_LOGIC_VECTOR(N-1 downto 0);

begin

shifted_0 <= i0(N-1) & i0(N-3 downto N-M-1) & i0(N-2) & i0(N-M-2 downto 0);
shifted_1 <= i1(N-1) & i1(N-3 downto N-M-1) & i1(N-2) & i1(N-M-2 downto 0);

mux_0: MUX_2_1 Generic map(N)
Port map(i0 => shifted_0, i1 => shifted_1, s => sel_u(0), u => u0);

```

```

mux_1: MUX_2_1 Generic map(N)
Port map(i0 => shifted_0, i1 => shifted_1, s => sel_u(1), u => u1);

end ibrido;

```

Di seguito troviamo il modulo del multiplexer 2:1 realizzato con approccio dataflow.

```

entity MUX_2_1 is
Generic ( N : integer := 2 );
  Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
         i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
         s : in STD_LOGIC;
         u : out STD_LOGIC_VECTOR(N-1 downto 0));
end MUX_2_1;

architecture Dataflow of MUX_2_1 is

begin
with s select
  u <= i0 when '0',
  i1 when '1',
  (others => '-') when others;

end Dataflow;

```

Di seguito è riportato il test bench che mira a testare:

- il caso di invio pipelined;
- il caso di reset della linea;
- la collisione sul filo d'uscita di un nodo;
- il passaggio parallelo nel nodo di due messaggi aventi destinatari distinti
- l'invio simultaneo dei messaggi su tutte le linee.

```

ENTITY switch_8_tb IS
END switch_8_tb;

ARCHITECTURE behavior OF switch_8_tb IS

  -- Component Declaration for the Unit Under Test (UUT)

  COMPONENT switch_8
  PORT(
    i0 : IN std_logic_vector(2 downto 0);
    i1 : IN std_logic_vector(2 downto 0);
    i2 : IN std_logic_vector(2 downto 0);
    i3 : IN std_logic_vector(2 downto 0);
    i4 : IN std_logic_vector(2 downto 0);
    i5 : IN std_logic_vector(2 downto 0);

```

```

    i6 : IN std_logic_vector(2 downto 0);
    i7 : IN std_logic_vector(2 downto 0);
    dest_i0 : IN std_logic_vector(2 downto 0);
    dest_i1 : IN std_logic_vector(2 downto 0);
    dest_i2 : IN std_logic_vector(2 downto 0);
    dest_i3 : IN std_logic_vector(2 downto 0);
    dest_i4 : IN std_logic_vector(2 downto 0);
    dest_i5 : IN std_logic_vector(2 downto 0);
    dest_i6 : IN std_logic_vector(2 downto 0);
    dest_i7 : IN std_logic_vector(2 downto 0);
    u0 : OUT std_logic_vector(8 downto 0);
    u1 : OUT std_logic_vector(8 downto 0);
    u2 : OUT std_logic_vector(8 downto 0);
    u3 : OUT std_logic_vector(8 downto 0);
    u4 : OUT std_logic_vector(8 downto 0);
    u5 : OUT std_logic_vector(8 downto 0);
    u6 : OUT std_logic_vector(8 downto 0);
    u7 : OUT std_logic_vector(8 downto 0)
);
END COMPONENT;

```

--Inputs

```

signal i0 : std_logic_vector(2 downto 0) := (others => '0');
signal i1 : std_logic_vector(2 downto 0) := (others => '0');
signal i2 : std_logic_vector(2 downto 0) := (others => '0');
signal i3 : std_logic_vector(2 downto 0) := (others => '0');
signal i4 : std_logic_vector(2 downto 0) := (others => '0');
signal i5 : std_logic_vector(2 downto 0) := (others => '0');
signal i6 : std_logic_vector(2 downto 0) := (others => '0');
signal i7 : std_logic_vector(2 downto 0) := (others => '0');
signal dest_i0 : std_logic_vector(2 downto 0) := (others => '0');
signal dest_i1 : std_logic_vector(2 downto 0) := (others => '0');
signal dest_i2 : std_logic_vector(2 downto 0) := (others => '0');
signal dest_i3 : std_logic_vector(2 downto 0) := (others => '0');
signal dest_i4 : std_logic_vector(2 downto 0) := (others => '0');
signal dest_i5 : std_logic_vector(2 downto 0) := (others => '0');
signal dest_i6 : std_logic_vector(2 downto 0) := (others => '0');
signal dest_i7 : std_logic_vector(2 downto 0) := (others => '0');

```

--Outputs

```

signal u0 : std_logic_vector(8 downto 0);
signal u1 : std_logic_vector(8 downto 0);
signal u2 : std_logic_vector(8 downto 0);
signal u3 : std_logic_vector(8 downto 0);
signal u4 : std_logic_vector(8 downto 0);
signal u5 : std_logic_vector(8 downto 0);
signal u6 : std_logic_vector(8 downto 0);
signal u7 : std_logic_vector(8 downto 0);

```

-- No clocks detected in port list. Replace <clock> below with

```

-- appropriate port name

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: switch_8 PORT MAP (
    i0 => i0,
    i1 => i1,
    i2 => i2,
    i3 => i3,
    i4 => i4,
    i5 => i5,
    i6 => i6,
    i7 => i7,
    dest_i0 => dest_i0,
    dest_i1 => dest_i1,
    dest_i2 => dest_i2,
    dest_i3 => dest_i3,
    dest_i4 => dest_i4,
    dest_i5 => dest_i5,
    dest_i6 => dest_i6,
    dest_i7 => dest_i7,
    u0 => u0,
    u1 => u1,
    u2 => u2,
    u3 => u3,
    u4 => u4,
    u5 => u5,
    u6 => u6,
    u7 => u7
);

```

```

-- Stimulus process
stim_proc: process
begin

i0 <= "100";
i2 <= "111";
dest_i0 <= "101";
dest_i2 <= "110";

wait for 50 ns;

assert u5 = "110100000";
report "errore nel passaggio di i0"
severity failure;

```

```

assert u6 = "111001011";
report "errore nel passaggio di i2"
severity failure;

wait for 100 ns;

i0 <= "100";
i2 <= "111";
dest_i0 <= "110";
dest_i2 <= "101";

wait for 10 ns;
assert u4 = "110000010";
report "errore collisione"
severity failure;

wait for 100 ns;

i0 <= "100";
i1 <= "111";
i2 <= "110";
i3 <= "101";
i4 <= "100";
i5 <= "111";
i6 <= "110";
i7 <= "101";
dest_i0 <= "001";
dest_i1 <= "010";
dest_i2 <= "011";
dest_i3 <= "100";
dest_i4 <= "101";
dest_i5 <= "110";
dest_i6 <= "111";
dest_i7 <= "000";

wait for 100 ns;

i0 <= "100";
i1 <= "111";
i2 <= "110";
i3 <= "101";
i4 <= "100";
i5 <= "111";
i6 <= "110";
i7 <= "101";
dest_i0 <= "000";
dest_i1 <= "001";
dest_i2 <= "010";
dest_i3 <= "011";
dest_i4 <= "100";

```

```
dest_i5 <= "101";
dest_i6 <= "110";
dest_i7 <= "111";
```

```
wait for 100 ns;
```

```
i0(2) <= '0';
i1(2) <= '0';
i2(2) <= '0';
i3(2) <= '0';
i4(2) <= '0';
i5(2) <= '0';
i6(2) <= '0';
i7(2) <= '0';
```

```
wait for 100 ns;
```

```
i0 <= "100";
i1 <= "111";
i2 <= "110";
i3 <= "101";
i4 <= "100";
i5 <= "111";
i6 <= "110";
i7 <= "101";
dest_i0 <= "001";
dest_i1 <= "010";
dest_i2 <= "011";
dest_i3 <= "100";
dest_i4 <= "101";
dest_i5 <= "110";
dest_i6 <= "111";
dest_i7 <= "000";
      wait;
end process;
```

```
END;
```

I risultati della simulazione sono.

Name	Value	0 ns	100 ns	200 ns	300 ns	400 ns	500 ns	600 ns	700 ns	800 ns	900 ns
i0[2:0]	100			100						100	
i1[2:0]	111		000		111		000			111	
i2[2:0]	110		111		110		010			110	
i3[2:0]	101		000		101		001			101	
i4[2:0]	100		000		100		000			100	
i5[2:0]	111		000		111		011			111	
i6[2:0]	110		000		110		010			110	
i7[2:0]	101		000		101		001			101	
dest_i0[2:0]	001		101		110		001			001	
dest_i1[2:0]	010		000		010		001			010	
dest_i2[2:0]	011		110		101		011			011	
dest_i3[2:0]	100		000		100		011			100	
dest_i4[2:0]	101		000		101		100			101	
dest_i5[2:0]	110		000		110		101			110	
dest_i6[2:0]	111		000		111		110			111	
dest_i7[2:0]	000			000			111			000	
u0[8:0]	100011101				100011101		100000000			100011101	
u1[8:0]	100100000				100100000		100100111			100100000	
u2[8:0]	101000111				101000111		101001010			101000111	
u3[8:0]	101101010				101101010		101101101			101101010	
u4[8:0]	110001101				110001101		110010000			110001101	
u5[8:0]	110110000		110100000		110101011		110110111			110110000	
u6[8:0]	111010111		111000000		111010111		111011010			111010111	
u7[8:0]	111111010				11111010		11111101			111111010	

Notiamo che:

- l'invio parallelo di due messaggi aventi destinatari diversi ha successo (prima transizione);
- In caso di collisioni non è trasmesso il messaggio proveniente dall'host con indice minore (seconda transizione)
- Si è trasmessi su tutte le linee in modo pipelined (terza e quarta transizione)
- Si è resettato con successo la linea abbassando le abilitazioni di tutti gli host prima di procedere ad un nuovo invio (quinta e sesta transizione).
- Si evince in ogni messaggio in modo distinto il protocollo usato verificando che l'abilitazione sia alta, che il destinatario e la sorgente siano corretti, che il payload ricevuto corrisponda a quello inviato.

Esercizio 11

11.1 Traccia

Esercizio 11

Progettare e implementare in VHDL una macchina aritmetica sequenziale a scelta fra le seguenti:
moltiplicatore di Robertson, per effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna;
moltiplicatore di Booth, per effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna;
divisore non-restoring, per effettuare la divisione intera fra due stringhe A e B di 4 bit ciascuna;
divisore restoring, per effettuare la divisione intera fra due stringhe A e B di 4 bit ciascuna;
Opzionalmente, la macchina implementata può essere sintetizzata su FPGA e testata mediante l'utilizzo dei dispositivi di input/output (switch, bottoni, led, display) presenti sulla board di sviluppo in dotazione al gruppo. La modalità di utilizzo degli stessi è a completa discrezione degli studenti.

11.2 Accenni all'algoritmo del moltiplicatore di Robertson

Il moltiplicatore di Robertson è un moltiplicatore sequenziale che esegue prodotti in modo molto simile all'operazione di moltiplicazione in algebra. L'idea è quella di sommare colonna per colonna sistemandone di volta in volta in modo opportuno gli operandi di modo che ad ogni passo della moltiplicazione la somma avvenga sempre tra cifre dello stesso peso. Consideriamo l'algebra binaria per fare un esempio e capirne il ragionamento. Consideriamo due numeri X e Y e proviamo a farne il prodotto supponendo Y = 1010 (10 in decimale) e X = 1101 (13 in decimale).

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

Eseguendo i passaggi come appreso dall'operazione algebrica della moltiplicazione al primo passo scriveremo il valore del moltiplicando in prodotto col primo bit del moltiplicatore che somma tutti zero essendo che alcuna somma parziale è stata eseguita. Come sappiamo, al passo successivo dovremo sommare le cifre di peso maggiore, quindi, è come se andassimo a fare uno shift a destra della somma parziale e sommiamo sempre gli stessi valori nelle stesse posizioni. Dunque, al secondo passo sommiamo al prodotto parziale shiftato il valore del moltiplicando per il secondo bit del moltiplicatore. Rieseguiamo uno shift a destra del prodotto parziale per rendere coerente la somma al terzo passo. Iterando tale processo saremo in grado di ricavare il risultato del prodotto che in questo caso è $X \cdot Y = 10000010$ (130 in decimale), notando che i bit del prodotto sono il doppio dei bit degli operandi in uso. Inoltre, il numero di passi da eseguire dipende direttamente

dal numero di cifre del moltiplicatore: in questo caso essendo a 4 bit impieghiamo 4 passi. L'operazione così definita non è eseguibile su numeri negativi per cui dovremo effettuare delle correzioni all'algoritmo visto. In primo luogo, notiamo che con la rappresentazione in complementi a due i numeri positivi sono resi con il bit più significativo basso e calcolati in decimale con la seguente formula :

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

I numeri negativi in complementi a due invece sono resi con il bit più significativo ad uno e sono calcolati con la stessa sommatoria con la differenza che il bit a valenza maggiore ha segno meno ovvero:

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

Notiamo inoltre che nell'algoritmo visto conosciamo a priori solo il segno del moltiplicando e non quello del moltiplicatore che è scoperto solo all'ultimo passo. Questa considerazione ci porta a valutare quattro situazioni possibili nelle quali dovremo all'occorrenza eseguire un ultimo passo detto di correzione, ovvero se notassimo gli operandi discordi l'ultima operazione non sarà una somma bensì una differenza. In particolare, abbiamo 4 possibili casi da dover gestire:

- Se X e Y sono entrambi positivi e facciamo una moltiplicazione, le somme sono sempre positive, non dobbiamo correggere niente;
- Se X è positivo ma Y (moltiplicando) è negativo, essendo trasparente al controllo sapremo subito il segno del prodotto parziale per cui ogni volta che prendiamo la Y e la moltiplichiamo per una cifra non nulla il prodotto parziale risulta negativo, e salviamo il segno, un bit pari ad 1, in un flip flop, tale valore resta invariato fintanto che non è scoperto il segno del moltiplicatore (fare questa cosa serve per effettuare l'estensione con segno del prodotto parziale nel momento in cui si fa lo shift a destra, ovvero il flip flop è posto in ingresso allo shift register);
- Se Y è positivo e X è negativo ci accorgeremo del segno meno solo all'ultimo passo per cui dovremo correggere il risultato effettuando una differenza invece che una somma essendo coerenti alla rappresentazione in complementi (passo di correzione);
- Se X e Y sono entrambi negativi, inizialmente supponiamo il prodotto parziale negativo, alla fine scoprendo il segno del moltiplicatore sapremo che il prodotto parziale è positivo, per cui correggiamo sottraendo invece che sommando all'ultimo passo (passo di correzione).

11.3 Soluzione

Abbiamo scelto di implementare il moltiplicatore di Robertson. Il sistema è suddiviso in unità operativa e unità di controllo. Esso presenta come ingressi il moltiplicando **A**, ad **N bit**, il moltiplicatore **B**, ad **M bit**, un segnale di *start* per iniziare con l'algoritmo, il *clock* e un segnale di *reset* e presenta come unica uscita il prodotto finale, **P**.

L'unità operativa è costituita da due registri a scorrimento, due multiplexer 2:1, un flip flop D, due registri parallelo-parallelo, un contatore e un addizionatore/sottrattore.

Il primo registro a scorrimento è il registro A (**shift register A**), esso è costituito da N celle di memoria, 8 nel caso testato in simulazione. È inizializzato a 0 ($P_0 = 0$) e viene caricato in modo parallelo con i prodotti parziali calcolati durante l'esecuzione. È stato scelto uno shift register per memorizzare i prodotti parziali perché questi devono essere opportunamente shiftati di un bit a destra ad ogni passo dell'esecuzione

$$(P_{i+1} = 2^{-i} (P_i + B_i A))$$

Al termine dell'algoritmo l'uscita parallela del registro A costituisce la metà più significativa del prodotto finale (**P15:P8**, nel caso testato). L'uscita seriale dello shift register A entra nell'ingresso seriale dello shift register Q in modo tale da conservare permanentemente il bit poiché non deve essere più considerato nelle somme dei prodotti parziali.

Lo **shift register Q**, invece, è inizializzato con il **moltiplicatore** attraverso un caricamento parallelo pilotato dall'unità di controllo. Quindi, è costituito da M celle di memoria. È stato utilizzato uno shift register in modo tale che ad ogni passo dell'algoritmo, dopo aver elaborato il bit meno significativo

del moltiplicatore, questo viene shiftato a destra, perdendo la cifra meno significativa del moltiplicatore ma facendo spazio per un bit del prodotto parziale che non deve essere più considerato nelle somme dei prodotti parziali, perché costituente già parte del prodotto finale. L'uscita seriale dello shift register Q può essere "scartata" poiché è costituita dai bit del moltiplicatore che sono stati elaborati e non devono essere più considerati. Al termine dell'esecuzione l'uscita parallela dello shift register Q costituisce la metà meno significativa del prodotto finale (**P7:P0**, nel caso testato).

In pratica, i due shift register, insieme, costituiscono un singolo shift register più grande ad $M + N$ celle di memoria, però sono stati suddivisi in due per semplicità di gestione dei caricamenti paralleli e perché, almeno inizialmente, hanno utilizzzi concettualmente differenti.

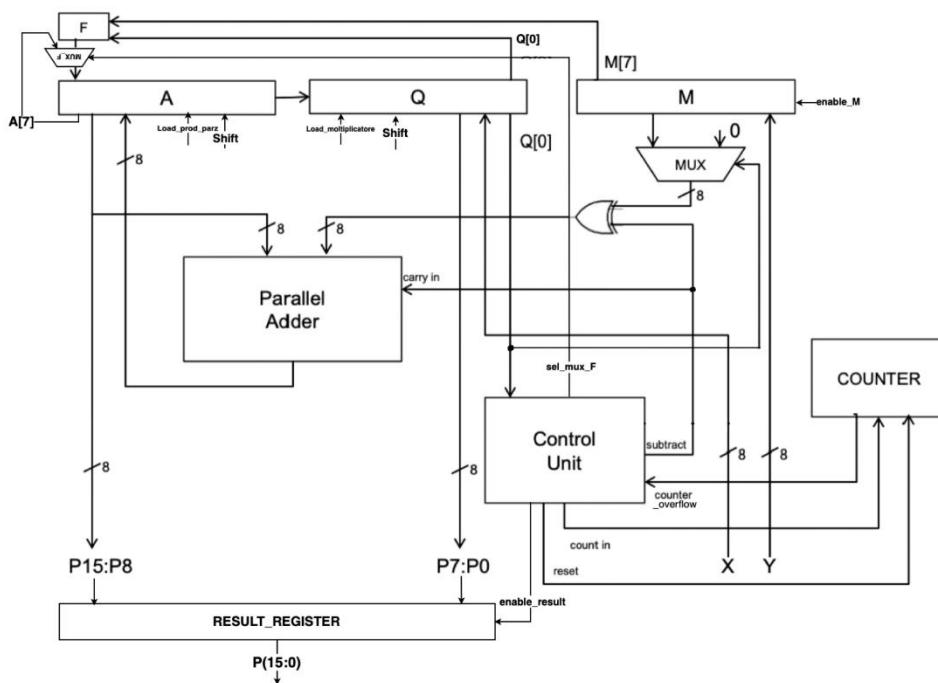
La concatenazione delle uscite dei due shift register sono l'ingresso del registro parallelo-parallelo contenente il prodotto finale (**Result_register**). Questo, a tale scopo, campiona l'uscita dei due registri solo quando si è certi che l'esecuzione è terminata, quindi presenta un'abilitazione pilotata dall'unità di controllo.

Il secondo registro parallelo-parallelo è il **registro M**, esso è inizializzato con il **moltiplicando** e il suo contenuto non varia per tutto il tempo di esecuzione dell'algoritmo. Anche il registro M presenta un'abilitazione pilotata dall'unità di controllo. L'uscita del registro M entra in ingresso al primo multiplexer 2:1, questo è utilizzato durante l'esecuzione per calcolare i prodotti parziali $P_{i+1} = 2^{-1} (P_i + B_i A)$. Mentre, il secondo ingresso è 0. L'ingresso di selezione del multiplexer è il bit meno significativo del registro Q, ovvero è la cifra del moltiplicatore che viene elaborata in quel passo, ($Q(0)$, cioè B_i nell'algoritmo). A seconda dell'ingresso di selezione: se è 1, l'uscita del multiplexer coincide con il moltiplicando, poiché si effettua teoricamente il prodotto $1 \times A$; se è 0, l'uscita del multiplexer è 0, poiché si effettua teoricamente il prodotto $0 \times A$. L'uscita del multiplexer, in entrambi i casi, entra come ingresso al Parallel Adder, rappresentando l'operando 2. L'ingresso rappresentante l'operando 1 del Parallel Adder è, invece, l'uscita parallela dello shift register A, ovvero il prodotto parziale calcolato al passo precedente. Presi in ingresso il prodotto parziale del passo precedente, P_i e il prodotto tra il bit analizzato del moltiplicatore e il moltiplicando, $B_i A$, il parallel adder calcola il prossimo prodotto parziale P_{i+1} , il quale viene memorizzato, come detto in precedenza, nello shift register A, attraverso un caricamento parallelo pilotato dall'unità di controllo.

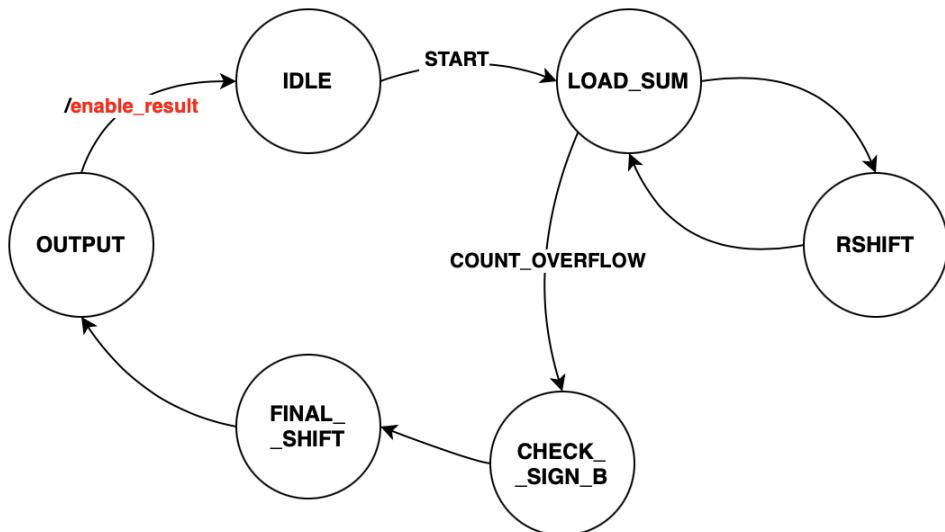
Durante tutti i passi dell'algoritmo il Parallel adder calcola la somma $P_i + B_i A$, fatta eccezione per l'ultimo passo, ovvero quando viene analizzato il bit più significativo del moltiplicatore, nel caso in cui esso valga 0 si procede con la somma, mentre, nel caso in cui esso valga 1 bisogna effettuare un passo di correzione e bisogna sostituire l'operazione di somma con l'operazione di sottrazione $P_i - B_i A$. A tale scopo, nasce la necessità di avere il Parallel Adder in grado di effettuare sia addizioni che sottrazioni a comando. Quindi, è stato implementato in modo tale che l'operando 1 passi invariato all'interno dell'addizionatore, mentre l'operando 2 entri come ingresso di una porta XOR con un segnale trasmesso dall'unità di controllo, *subtract*. In tal modo a seconda del valore assunto da *subtract* l'operando può passare invariato (caso *subtract* = 0) oppure gli si può effettuare il complemento a 1 (caso *subtract* = 1). Nel caso *subtract* = 0, il *carry_in* dell'addizionatore è settato pari a 0 (*carry_in* = 0) e si procede con la somma, mentre, nel caso *subtract* = 1, il *carry_in* è settato pari 1 (quindi *carry_in* = 1) in modo tale che si completa l'operazione di negazione dell'operando, effettuando così il complemento a due e procedendo, quindi, con l'operazione di sottrazione. È possibile osservare che, in entrambi i casi, *carry_in* coincide con il segnale *subtract*, infatti essi sono stati considerati come un unico segnale pilotato dall'unità di controllo. Il registro F è implementato come un Flip-Flop D con abilitazione, esso è importante nel caso in cui il moltiplicando è negativo e durante gli shift a destra dello shift register A, bisogna mantenere il segno caricando, come ingresso seriale del registro a scorrimento, il bit 1. Negli altri casi il registro F restituisce in uscita il bit 0.

In generale, esso alimenta sempre il bit più significativo del registro A e la sua uscita è calcolata secondo la *seguente* $F = (M(7) \text{ AND } Q(0)) \text{ OR } F$. Ovvero, se il bit più significativo del moltiplicando, $M(7)$ vale 1 e il moltiplicando è stato caricato nel registro A in uno dei passi dell'algoritmo, poiché moltiplicato con il bit, analizzato in quel passo, del moltiplicatore, il quale valeva 1, allora F diventa alto. Si osservi che il registro F è inizializzato a 0 come lo shift register A, quindi, quando viene analizzato il primo bit che vale 1 del moltiplicatore, nel caso in cui ce ne sia almeno uno, viene effettuata la somma tra il moltiplicando e lo shift register che fino a quel momento

non era variato e successivamente sarà caricato con il valore del moltiplicando. Solo in quel momento, nel caso in cui il moltiplicando sia negativo, F deve dare in uscita il valore 1, mentre in tutti gli altri casi deve valere 0. Però una volta memorizzato 1, si instaura il segno negativo e non bisogna più rimuoverlo, ciò spiega la *OR F*. Un'ultima osservazione riguardante il registro F, esso, durante l'ultimo shift, quello relativo a bit più significativo del moltiplicatore, non deve alimentare il registro A, infatti l'ultimo bit del registro a scorrimento A deve rimanere invariato ($A(7) = A(7)$). Questo serve a gestire il caso in cui sia il moltiplicatore che il moltiplicando sono negativi e all'ultimo passo il si ottiene un valore positivo, e quindi bisogna conservarne il segno. A tale scopo è stato inserito nell'architettura un altro multiplexer 2:1 (**mux_F**), esso presenta come ingressi dato, l'uscita del registro F (*F_out*) e il bit più significativo dello shift register A (*u_par_A(7)*) e come ingresso di selezione un segnale pilotato dall'unità di controllo la quale fa passare in uscita al multiplexer sempre l'uscita del registro F, con l'unica eccezione rappresentata, appunto dall'ultimo passo, nel quale la selezione dirige in uscita A(7). Infine, vi è un contatore il quale ha lo scopo di scandire, con il proprio conteggio, i bit del moltiplicatore e arrivato all'ultimo bit genera un segnale di stato, *count_overflow*, verso l'unità di controllo. L'architettura dell'unità operativa è rappresentata in figura.



L'unità di controllo è implementata in logica cablata secondo l'automa in figura.



Essa presenta 6 stati:

- **IDLE**, è lo stato in cui il moltiplicatore è "spento" e non effettua operazioni rilevanti. L'unità di controllo, in questo stato, si limita a resettare i registri e si pone in attesa del segnale di start per avviare l'esecuzione vera e propria. Ricevuto il segnale di start, campionato sul fronte di salita del clock, l'unità di controllo abilita il caricamento del moltiplicando nel registro M, con il segnale *enable_M*, e il caricamento del moltiplicatore nello shift register Q, con il segnale *load_moltiplicatore*, abbassa tutti i segnali di reset dei vari registri e del contatore e passa da questo stato allo stato *load_sum*;
- **LOAD_SUM**, in questo stato si abilita il caricamento in parallelo del prodotto parziale in uscita dal sommatore, nello shift register A, con il segnale *load_prod_parz*, e da l'abilitazione al registro F in modo tale che calcoli il proprio valore. Ricevuto un colpo di clock passa da questo stato allo stato *rshift*. Nel caso in cui è stato scandito completamente il moltiplicatore, arrivati all'ultimo bit, il contatore comunica lo stato dell'unità operativa con il segnale *count_overflow*, e l'unità di controllo passa dallo stato *load_sum* allo stato *check_sign_B*;
- **RSHIFT**, in questo stato l'unità di controllo abilita lo shift dei registri a scorrimento, con il segnale *shift*. La somma e lo shift sono stati suddivisi in due stati differenti, in modo tale da essere sicuri che venga effettuata prima la somma e successivamente lo shift, per garantire il corretto funzionamento. Dopo un colpo di clock si ritorna allo stato *load_sum*, eseguendo un loop tra i due stati, allo scopo di scandire tutti i bit del moltiplicatore. Quindi in questo stato bisogna fare anche un controllo sullo stato dell'unità operativa.
- **CHECK_SIGN_B**, in questo stato si controlla il bit meno significativo dello shift register Q (Q(0)) ricevuto in ingresso all'unità di controllo. Arrivati a tale stato, tale bit rappresenta il bit più significativo del moltiplicatore. Se esso vale 0, in teoria si procede con una somma, ma il suo valore 0 seleziona un operando pari a 0 per la somma, mediante il mux, per cui coerentemente con l'algoritmo di Robertson non si fa niente; nel caso in cui esso vale 1 si effettua il passo di correzione, abilitando l'operazione di sottrazione con il segnale *subtract* = 1. Dopo un colpo di clock si passa da questo stato allo stato *final_shift*.
- **FINAL SHIFT**, in questo stato si effettua uno shift differente da quello effettuato nello stato *shift*, poiché, come detto in precedenza, bisogna lasciare invariato il bit più significativo del registro A e shiftare il resto del due registri a scorrimento, questo viene garantito dall'unità di controllo abilitando ancora una volta lo shift dei registri ma cambiando l'ingresso di selezione del multiplexer (mux_F) con il segnale *sel_mux_F*. Al colpo di clock successivo si passa da questo stato allo stato finale *output*.
- **OUTPUT**, in questo stato si è sicuri che il prodotto finale è stato calcolato e si trova nei registri a scorrimento, quindi l'unità di controllo abilita l'acquisizione del risultato da parte del registro *Result_register*, con il segnale *enable_result*. In questo modo il registro memorizza solo il risultato e non memorizza i vari prodotti parziali calcolati durante l'esecuzione dell'algoritmo. Al successivo colpo di clock l'unità di controllo si porta nuovamente nello stato iniziale, *idle*, per procedere con un'ipotetica nuova elaborazione.

11.4 Codice

Di seguito vi è il codice relativo al moltiplicatore, rappresentante i vari componenti illustrati precedentemente.

Il primo componente è lo shift register, esso è implementato con un approccio Structural e permette, oltre all'inserimento e all'uscita seriali, anche il caricamento e l'uscita parallela.

```
entity shift_register is
Generic(N : integer :=16);
  Port ( i_ser : in STD_LOGIC;--ingresso seriale
         u_ser: out STD_LOGIC;--uscita seriale
         load : in STD_LOGIC;--abilitazione per caricamento parallelo dei flip
flop
         i_par : in STD_LOGIC_VECTOR(N-1 downto 0);--ingresso parallelo
         CLK : in STD_LOGIC;
         RST : in STD_LOGIC;
         SHIFT : in STD_LOGIC;--abilitazione dei flip flop
         u_par : out STD_LOGIC_VECTOR(N-1 downto 0)
       );
end shift_register;

architecture Structural of shift_register is
component FLIP_FLOP_D_LOAD is
  Port ( CLK : in STD_LOGIC;
         EN : in STD_LOGIC;
         RESET: in STD_LOGIC;
         D : in STD_LOGIC;
         Q : out STD_LOGIC;
         I : in STD_LOGIC;--ingresso parallelo
         SET : in STD_LOGIC
       );
end component;

signal T : STD_LOGIC_VECTOR (N-2 downto 0) := (others => '0');
signal u_ser_internal : STD_LOGIC := '0';

begin
  --flip flop con bit più significativo
  ff_0: FLIP_FLOP_D_LOAD
    Port map(CLK => CLK, EN => SHIFT, RESET => RST, D => i_ser, Q => T(N-2), I =>
i_par(N-1), SET => LOAD);

  ff_1toN_2: for k in 1 to N-2 generate
    ff_i: FLIP_FLOP_D_LOAD
      Port map(CLK => CLK, EN => SHIFT, RESET => RST, D => T(N-1-k), Q => T(N-2-k), I
=> i_par(N-1-k), SET => LOAD);
    end generate;
  ff_N_1: FLIP_FLOP_D_LOAD
    Port map(CLK => CLK, EN => SHIFT, RESET => RST, D => T(0), Q => u_ser_internal,
I => i_par(0), SET => LOAD);

  u_par <= T&u_ser_internal;
  u_ser <= u_ser_internal;
```

```
end Structural;
```

Di seguito vi è il codice dei Flip-Flop D con i quali sono stati implementate le celle di memoria dello shift register. Essi permettono anche il set di un valore. È implementato con un approccio Behavioral

```
entity FLIP_FLOP_D_LOAD is
  Port ( CLK : in STD_LOGIC;
         EN : in STD_LOGIC;
         RESET: in STD_LOGIC;
         D : in STD_LOGIC;
         Q : out STD_LOGIC;
         I : in STD_LOGIC;--ingresso parallelo
         SET : in STD_LOGIC
      );
end FLIP_FLOP_D_LOAD;

architecture rtl of FLIP_FLOP_D_LOAD is
signal q_internal : std_logic := '0';

begin
ff: process( CLK )
begin
  if( CLK'event and CLK = '1' ) then
    if( RESET = '1' ) then
      q_internal <= '0';
    elsif(SET = '1') then
      q_internal <= I;
    elsif(EN= '1') then
      q_internal <= D;
    else
      q_internal <= q_internal;
    end if;
  end if;
end process;
Q <= q_internal;
end rtl;
```

Di seguito vi è il codice del registro parallelo parallelo, anch'esso è implementato con un approccio Structural.

```
entity Registro is
Generic( N : integer := 8);
  Port ( CLK : in STD_LOGIC;
         EN: in STD_LOGIC;
         RESET : in STD_LOGIC;
         D : in STD_LOGIC_VECTOR(N-1 downto 0);
         Q : out STD_LOGIC_VECTOR(N-1 downto 0)
      );
end Registro;

architecture Structural of Registro is
```

```

component FLIP_FLOP_D is
port( CLK : in STD_LOGIC;
      EN: in STD_LOGIC;
      RESET: in STD_LOGIC;
      D : in STD_LOGIC;
      Q : out STD_LOGIC
);
end component;

begin

regN_1to0: for i in N-1 downto 0 generate
  FF: FLIP_FLOP_D
  Port map( CLK => CLK,
            EN => EN,
            RESET => RESET,
            D => D(i),
            Q => Q(i)
  );
end generate;
end Structural;

```

Di seguito vi è il codice del adder/subtractor, esso è implementato con approccio Structural ed è costituito da un ripple carry adder per effettuare la somma e da una porta XOR, nel caso in cui vi è l'esigenza di negare l'operando 2.

```

entity adder_subtractor is
Generic(N : integer := 8);
  Port ( op1 : in STD_LOGIC_VECTOR(N-1 downto 0);
         op2 : in STD_LOGIC_VECTOR(N-1 downto 0);
         subtract : in STD_LOGIC;
         c_out : out STD_LOGIC;
         res : out STD_LOGIC_VECTOR(N-1 downto 0));
end adder_subtractor;

architecture Structural of adder_subtractor is

component ripple_carry_adder is
Generic(N : integer := 8);
  Port ( op1 : in STD_LOGIC_VECTOR(N-1 downto 0);
         op2 : in STD_LOGIC_VECTOR(N-1 downto 0);
         c_in : in STD_LOGIC;
         c_out : out STD_LOGIC;
         res : out STD_LOGIC_VECTOR(N-1 downto 0));
end component;

signal op2_internal : STD_LOGIC_VECTOR(N-1 downto 0);

begin

operand_2: for i in 0 to N-1 generate
  op2_internal(i) <= op2(i) xor subtract;

```

```

end generate;

rca: ripple_carry_adder Generic map(N
Port map(op1 => op1, op2 => op2_internal, c_in => subtract, c_out => c_out, res
=> res);

end Structural;

```

Il ripple carry adder è anch'esso implementato con approccio Structural, componendolo con Full Adder.

```

entity ripple_carry_adder is
Generic(N : integer := 8);
  Port ( op1 : in STD_LOGIC_VECTOR(N-1 downto 0);
         op2 : in STD_LOGIC_VECTOR(N-1 downto 0);
         c_in : in STD_LOGIC;
         c_out : out STD_LOGIC;
         res : out STD_LOGIC_VECTOR(N-1 downto 0));
end ripple_carry_adder;

architecture Structural of ripple_carry_adder is

component full_adder is
  Port ( op1 : in STD_LOGIC;
         op2 : in STD_LOGIC;
         c_in : in STD_LOGIC;
         c_out : out STD_LOGIC;
         res : out STD_LOGIC);
end component;

signal carry_in : STD_LOGIC_VECTOR(N-1 downto 0) := (others => '0');

begin
f_a_0: full_adder Port map (op1 => op1(0), op2 => op2(0), c_in => c_in, c_out
=> carry_in(1), res => res(0));

f_a_1toN_2: for i in 1 to N-2 generate
f_a_i: full_adder Port map (op1 => op1(i), op2 => op2(i), c_in => carry_in(i),
c_out => carry_in(i + 1), res => res(i));
end generate;

f_a_N_1: full_adder Port map (op1 => op1(N - 1), op2 => op2(N - 1), c_in =>
carry_in(N-1), c_out => c_out, res => res(N - 1));
end Structural;

```

Di seguito vi è il codice relativo ai Full Adder.

```

entity full_adder is
  Port ( op1 : in STD_LOGIC;
         op2 : in STD_LOGIC;
         c_in : in STD_LOGIC;
         c_out : out STD_LOGIC;
         res : out STD_LOGIC);

```

```

end full_adder;

architecture Dataflow of full_adder is

begin
    res <= op1 XOR op2 XOR c_in;
    c_out <= (op1 AND op2) OR (c_in AND (op1 XOR op2));
end Dataflow;

```

Di seguito vi è il codice del Flip-Flop D utilizzato per il registro F, il quale si differenzia da quello utilizzato per comporre le celle di memoria dello shift register, poiché non presenta l'ingresso di set. Esso è implementato con un approccio behavioral.

```

entity FLIP_FLOP_D is
    Port ( CLK : in STD_LOGIC;
            EN : in STD_LOGIC;
            RESET: in STD_LOGIC;
            D : in STD_LOGIC;
            Q : out STD_LOGIC
        );
end FLIP_FLOP_D;

architecture rtl of FLIP_FLOP_D is
begin
ff: process( CLK )
begin
    if( CLK'event and CLK = '1' ) then
        if( RESET = '1' ) then
            Q <= '0';
        elsif(EN= '1') then
            Q <= D;
        end if;
    end if;
end process;

end rtl;

```

Il multiplexer2:1 è implementato con un approccio Dataflow e presenta i due ingressi dato in parallelo.

```

entity mux_2_1 is
Generic ( N : integer := 2 );
Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
       i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
       s : in STD_LOGIC;
       u : out STD_LOGIC_VECTOR(N-1 downto 0));
end mux_2_1;

architecture Dataflow of mux_2_1 is

begin
    with s select
        u <= i0 when '0',

```

```

        i1 when '1',
(others => '-') when others;

end Dataflow;

```

Il contatore è implementato con un approccio Behavioral.

```

entity counter is
Generic(N : integer := 3; M : integer := 7);
--N rappresenta l'esponente di 2, es. N=3, il contatore si comporta come modulo8
--M rappresenta il valore di overflow
  Port ( clock : in STD_LOGIC;
         reset : in STD_LOGIC;
         enable : in STD_LOGIC;
         count : out STD_LOGIC_VECTOR (N-1 downto 0);
         overflow : out STD_LOGIC);
end counter;

architecture Behavioral of counter is

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

begin

count <= c;

counter_process: process(clock, reset)
begin

if reset = '1' then
  c <= (others => '0');
  overflow <= '0';
elsif clock'event AND clock = '1' AND enable = '1' then
  if to_integer(unsigned(c)) = (M - 1) then
    c <= (others => '0');
    overflow <= '1';
  else
    c <= std_logic_vector(unsigned(c) + 1);
    overflow <= '0';
  end if;
end if;
end process;

end Behavioral;

```

L'unità di controllo è implementata con un approccio Behavioral, attraverso un unico process, sensibile al solo segnale di clock. Questa è implementata in logica cablata, con un automa a stati finiti, gestendo gli stati con il costrutto *case...when*.

```

entity control_unit is
Generic(N : integer := 8);
  Port ( q_0 : in STD_LOGIC;
         clock : in STD_LOGIC;

```

```

        reset : in STD_LOGIC;
        count_overflow : in STD_LOGIC;
        start : in STD_LOGIC;
        subtract : out STD_LOGIC;
        count_in : out STD_LOGIC;
        enable_M : out STD_LOGIC;
        enable_F : out STD_LOGIC;
        load_A : out STD_LOGIC;
        load_Q : out STD_LOGIC;
        reset_regs : out STD_LOGIC;
        enable_result : out STD_LOGIC;
        shift : out STD_LOGIC;
        sel_mux_f : out STD_LOGIC);
end control_unit;

architecture Behavioral of control_unit is

type state is (idle, load_sum, rshift, check_sign_B, final_shift, output);

signal q : state := idle;
begin

control: process(clock)
begin
if reset = '1' then
    q <= idle;
elsif (clock'event and clock = '1') then
    case q is
    when idle =>
        if start = '1' then
            q <= load_sum; subtract <= '0'; count_in <= '0'; enable_M <= '1';
            enable_F <= '0'; load_A <= '0'; load_Q <= '1'; reset_regs <= '0';
            enable_result <= '0'; shift <= '0'; sel_mux_f <= '0';
        else
            q <= idle; subtract <= '0'; count_in <= '0'; enable_M <= '0';
            enable_F <= '0'; load_A <= '0'; load_Q <= '0'; reset_regs <= '1';
            enable_result <= '0'; shift <= '0'; sel_mux_f <= '0';
        end if;
    when load_sum =>
        if count_overflow = '1' then
            q <= check_sign_B; subtract <= '0'; count_in <= '0'; enable_M <= '0';
            enable_F <= '0'; load_A <= '0'; load_Q <= '0'; reset_regs <= '0';
            enable_result <= '0'; shift <= '0'; sel_mux_f <= '0';
        else
            q <= rshift; subtract <= '0'; count_in <= '1'; enable_M <= '0';
            enable_F <= '1'; load_A <= '1'; load_Q <= '0'; reset_regs <= '0';
            enable_result <= '0'; shift <= '0'; sel_mux_f <= '0';
        end if;
    when rshift =>
        q <= load_sum; subtract <= '0'; count_in <= '0'; enable_M <= '0';
        enable_F <= '0'; load_A <= '0'; load_Q <= '0'; reset_regs <= '0';
        enable_result <= '0'; shift <= '1'; sel_mux_f <= '0';
    when check_sign_B =>
        q <= final_shift; count_in <= '0'; enable_M <= '0';

```

```

enable_F <= '0'; load_A <= '1'; load_Q <= '0'; reset_regs <= '0';
enable_result <= '0'; shift <= '0'; sel_mux_f <= '1';
if q_0 = '1' then
    subtract <= '1';
else
    subtract <= '0';
end if;
when final_shift =>
    q <= output; subtract <= '0'; count_in <= '0'; enable_M <= '0';
    enable_F <= '0'; load_A <= '0'; load_Q <= '0'; reset_regs <= '0';
    enable_result <= '0'; shift <= '1'; sel_mux_f <= '1';
when output =>
    q <= idle; subtract <= '0'; count_in <= '0'; enable_M <= '0';
    enable_F <= '0'; load_A <= '0'; load_Q <= '0'; reset_regs <= '0';
    enable_result <= '1'; shift <= '0'; sel_mux_f <= '0';
when others =>
    q <= idle;
end case;
end if;
end process;

end Behavioral;

```

A questo punto, il top module è il moltiplicatore stesso, che fornisce l’interfaccia per porre in ingresso gli operandi da moltiplicatore, il segnale di start, clock e reset, e fornisce in uscita il risultato della moltiplicazione con segno. Gli operandi hanno entrambi numero di bit generic e si è utilizzato il binding dinamico per il sommatore, per prevedere la possibilità di utilizzare diverse implementazioni del sommatore, dato che è presente l’”*adder in the loop*”.

```

use work.all;

entity Robertson_multiplier is
Generic(N : integer := 8; M : integer :=8); --N numero bit moltiplicando, M numero
bit moltiplicatore
  Port ( A : in STD_LOGIC_VECTOR(N-1 downto 0); --moltiplicando
         B : in STD_LOGIC_VECTOR(M-1 downto 0); --moltiplicatore
         START : in STD_LOGIC;
         CLK : in STD_LOGIC;
         RST : in STD_LOGIC;
         P : out STD_LOGIC_VECTOR(N+M-1 downto 0));
end Robertson_multiplier;

architecture Structural of Robertson_multiplier is

component control_unit is
  Generic(N : integer := 8);
  Port ( q_0 : in STD_LOGIC;
          clock : in STD_LOGIC;
          reset : in STD_LOGIC;
          count_overflow : in STD_LOGIC;
          start : in STD_LOGIC;
          subtract : out STD_LOGIC;
          count_in : out STD_LOGIC;

```

```

enable_M : out STD_LOGIC;
enable_F : out STD_LOGIC;
load_A : out STD_LOGIC;
load_Q : out STD_LOGIC;
reset_regs : out STD_LOGIC;
enable_result : out STD_LOGIC;
    shift : out STD_LOGIC;
sel_mux_f : out STD_LOGIC);
end component;

component mux_2_1 is
  Generic ( N : integer := 2 );
  Port ( i0 : in STD_LOGIC_VECTOR(N-1 downto 0);
         i1 : in STD_LOGIC_VECTOR(N-1 downto 0);
         s : in STD_LOGIC;
         u : out STD_LOGIC_VECTOR(N-1 downto 0));
end component;

component shift_register is
  Generic(N : integer :=16);
  Port ( i_ser : in STD_LOGIC;--ingresso seriale
          u_ser : out STD_LOGIC;--uscita seriale
          load : in STD_LOGIC;--abilitazione per caricamento parallelo dei flip
flop
          i_par : in STD_LOGIC_VECTOR(N-1 downto 0);--ingresso parallelo
          CLK : in STD_LOGIC;
          RST : in STD_LOGIC;
          SHIFT : in STD_LOGIC;--abilitazione dei flip flop
          u_par : out STD_LOGIC_VECTOR(N-1 downto 0)
        );
end component;

component adder_subtractor is
  Generic(N : integer := 8);
  Port ( op1 : in STD_LOGIC_VECTOR(N-1 downto 0);
          op2 : in STD_LOGIC_VECTOR(N-1 downto 0);
          subtract : in STD_LOGIC;
          c_out : out STD_LOGIC;
          res : out STD_LOGIC_VECTOR(N-1 downto 0));
end component;

component Registro is
  Generic( N : integer := 8);
  Port ( CLK : in STD_LOGIC;
          EN: in STD_LOGIC;
          RESET : in STD_LOGIC;
          D : in STD_LOGIC_VECTOR(N-1 downto 0);
          Q : out STD_LOGIC_VECTOR(N-1 downto 0)
        );
end component;

component counter is
  Generic(N : integer := 3; M : integer := 7);

```

```

Port ( clock : in STD_LOGIC;
       reset : in STD_LOGIC;
enable : in STD_LOGIC;
       count : out STD_LOGIC_VECTOR (N-1 downto 0);
 overflow : out STD_LOGIC);
end component;

component FLIP_FLOP_D is
  Port ( CLK : in STD_LOGIC;
         EN : in STD_LOGIC;
         RESET: in STD_LOGIC;
         D : in STD_LOGIC;
         Q : out STD_LOGIC
      );
end component;

signal count_in, f_out, f_in, A_u_ser, load_prod_parz, subtract,
load_moltiplicatore,
en_F, enable_m, shift, count_overflow, enable_result, sel_mux_F, u_mux_F,
reset_F, reset_counter, reset_A, reset_regs : STD_LOGIC := '0';
signal M_out, prod_parz, A_u_par, mux_out, zero : STD_LOGIC_VECTOR(N-1 downto 0)
:= (others => '0');
signal Q_u_par : STD_LOGIC_VECTOR(M-1 downto 0) := (others => '0');
signal P_out : STD_LOGIC_VECTOR(N + M - 1 downto 0) := (others => '0');

begin

reset_F <= RST or reset_regs;
reset_A <= RST or reset_regs;
reset_counter <= RST or reset_regs;

cntrl_unit: control_unit
Generic map(N)
Port map(
q_0 => Q_u_par(0), clock => CLK, reset => RST, count_overflow => count_overflow,
start => START, subtract => subtract, count_in => count_in, enable_M => enable_m,
enable_F => en_F, load_A => load_prod_parz, load_Q => load_moltiplicatore,
reset_regs => reset_regs, enable_result => enable_result, shift => shift,
sel_mux_f => sel_mux_F);

-- registro contenente il moltiplicando
M_register: Registro
Generic map(N)
  Port map(CLK => CLK, EN => enable_m, RESET => RST, D => A, Q => M_out);

MUX_moltiplicando: mux_2_1
Generic map(N)
  Port map(i0 => zero, i1 => M_out, s => Q_u_par(0), u => mux_out);

MUX_F: mux_2_1
Generic map(1)
Port map(i0(0) => f_out, i1(0) => A_u_par(N-1), s => sel_mux_F, u(0) => u_mux_F);

A_shift_register: shift_register

```

```

Generic map(N)
  Port map( i_ser => u_mux_F, u_ser => A_u_ser, load => load_prod_parz, i_par
=> prod_parz, u_par => A_u_par,
             CLK => CLK, RST => reset_A, SHIFT => shift);
Q_shift_register: shift_register
Generic map(M)
  Port map( i_ser => A_u_ser, load => load_moltiplicatore, i_par => B, u_par
=> Q_u_par,
             CLK => CLK, RST => RST, SHIFT => shift);

Parallel_Adder: entity work.adder_subtractor(Structural)
Generic map(N)
  Port map(op1 => A_u_par, op2 => mux_out, subtract => subtract, res =>
prod_parz);

f_in <= ( (M_out(N-1) and Q_u_par(0)) or f_out );

F_register: FLIP_FLOP_D
  Port map( CLK => CLK, EN => en_F, RESET => reset_F, D => f_in, Q => f_out);

Contatore: counter
Generic map(M, M-1)
  Port map(clock => CLK, reset => reset_counter, enable => count_in, overflow
=> count_overflow);

Risultato: Registro
Generic map(N+M)
  Port map(CLK => CLK, EN => enable_result, RESET => RST, D => P_out, Q => P);
P_out <= A_u_par & Q_u_par;

end Structural;

```

11.5 Simulazione

Per la simulazione e la verifica del corretto funzionamento del sistema è stato implementato un *testbench*, il cui codice è mostrato di seguito.

```

ENTITY multiplier_testbench IS
END multiplier_testbench;

ARCHITECTURE behavior OF multiplier_testbench IS

  -- Component Declaration for the Unit Under Test (UUT)

  COMPONENT Robertson_multiplier
  PORT(
    A : IN  std_logic_vector(7 downto 0);
    B : IN  std_logic_vector(7 downto 0);
    START : IN  std_logic;
    CLK : IN  std_logic;
    RST : IN  std_logic;
    P : OUT  std_logic_vector(15 downto 0)
  
```

```

    );
END COMPONENT;

--Inputs
signal A : std_logic_vector(7 downto 0) := (others => '0');
signal B : std_logic_vector(7 downto 0) := (others => '0');
signal START : std_logic := '0';
signal CLK : std_logic := '0';
signal RST : std_logic := '0';

--Outputs
signal P : std_logic_vector(15 downto 0);

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

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: Robertson_multiplier PORT MAP (
    A => A,
    B => B,
    START => START,
    CLK => CLK,
    RST => RST,
    P => P
);

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

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100 ns.
    RST <= '1';
    wait for 100 ns;
    RST <= '0';
    wait for CLK_period;

    A <= "11010101"; -- (-43)
    B <= "10110011"; -- (-77)
    START <= '1';
    wait for CLK_period;
    START <= '0';
    wait for CLK_period*20;

```

```

A <= "00110010"; -- 50
B <= "10110011"; -- (-77)
START <= '1';
wait for CLK_period;
START <= '0';
wait for CLK_period*20;

A <= "11010101"; -- (-43)
B <= "01000111"; -- 71
START <= '1';
wait for CLK_period;
START <= '0';
wait for CLK_period*20;

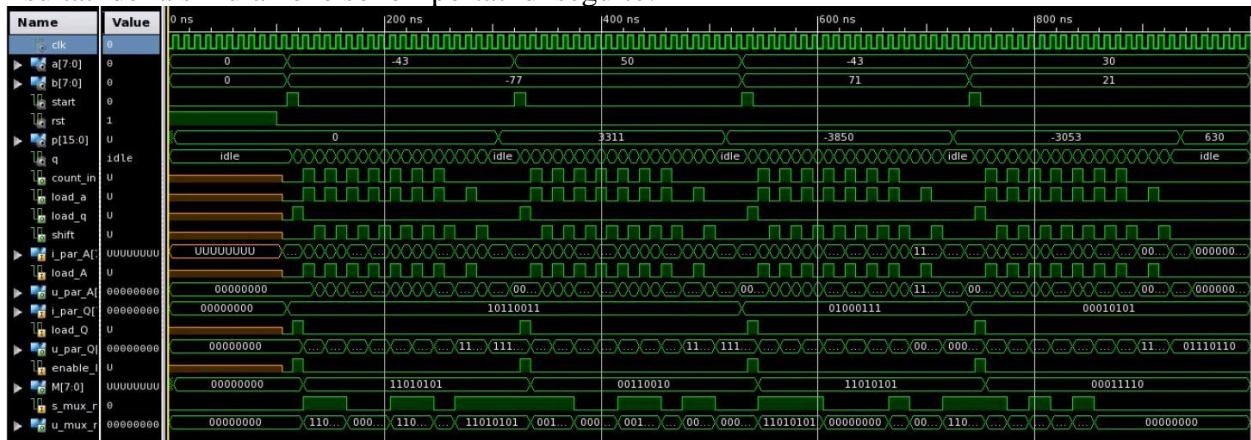
A <= "00011110"; -- 30
B <= "00010101"; -- 21
START <= '1';
wait for CLK_period;
START <= '0';
wait for CLK_period*20;

wait;
end process;

END;

```

I risultati della simulazione sono riportati di seguito:



11.6 Sintesi su FPGA

Per la sintesi su FPGA, abbiamo usato operandi ad 8 bit; avendo la Nexys A7 50T a disposizione, avevamo 16 switch, per cui abbiamo usato i primi 8 per il primo operando e gli altri per il secondo operando. Inoltre, abbiamo usato un bottone per lo start ed uno per il reset. All'interno del top module, abbiamo gestito lo start mediante un process ed un signal *actual_start*, in modo simile a come abbiamo fatto con il segnale di write nell'esercizio della seriale (anche se quello dipendeva dal segnale RDA in uscita dalla seriale): quando *actual_start* è alto, il process lo schedula basso per il colpo di clock successivo, mentre quando è basso, il process lo schedula pari al segnale di start campionato mediante il bottone. In questo modo si evita di mandare start in continuazione alla macchina. La visualizzazione è stata fatta con 4 cifre del display a 7 segmenti, dato che il risultato è a 16 bit ed ogni cifra del display corrisponde ad una cifra esadecimale, cioè a 4 bit. Come al solito, i

moduli `clock_filter` e `display_seven_segments` non li riportiamo perché ci sono stati forniti dai Docenti, e riportiamo invece quelli che abbiamo modificato, cioè il top module (`display_on_board`) e la control unit che stabilisce quali cifre abilitare (valori degli anodi) e quali valori scrivere (valori dei catodi); è chiaro che nella control unit è presente il moltiplicatore. Abbiamo inizialmente usato il moltiplicatore in maniera conservativa, con un clock di 1 MHz, poi ci siamo spinti fino a 10 MHz.

Il modulo `display_on_board`:

```

entity display_on_board is
Port(
  clock : in STD_LOGIC;
  reset : in STD_LOGIC;
  start : in STD_LOGIC;
  A : in STD_LOGIC_VECTOR(7 downto 0);
  B : in STD_LOGIC_VECTOR(7 downto 0);
  anodes : out STD_LOGIC_VECTOR (7 downto 0); --anodi e catodi delle cifre
  cathodes : out STD_LOGIC_VECTOR (7 downto 0)
);
end display_on_board;

architecture Structural of display_on_board is

COMPONENT display_seven_segments
  GENERIC(
    clock_frequency_in : integer := 50000000; --questi parametri servono a
    configurare
    clock_frequency_out : integer := 5000000 --il clock filter
  );
  PORT(
    clock : IN std_logic;
    reset_n : IN std_logic;
    value : IN std_logic_vector(31 downto 0);--6 nibble da mostrare
    enable : IN std_logic_vector(7 downto 0);--abilitazione delle 4 cifre
    dots : IN std_logic_vector(7 downto 0); --punti
    anodes : OUT std_logic_vector(7 downto 0);
    cathodes : OUT std_logic_vector(7 downto 0)
  );
END COMPONENT;

component ctrl_unit is
  Port (
    clock : in STD_LOGIC;
    reset_n : in STD_LOGIC;
    start : in STD_LOGIC;
    A : in STD_LOGIC_VECTOR(7 downto 0);
    B : in STD_LOGIC_VECTOR(7 downto 0);
    value : out STD_LOGIC_VECTOR(31 downto 0);
    enable : out STD_LOGIC_VECTOR(7 downto 0)
  );
end component;

component clock_filter is
  generic(
    clock_frequency_in : integer := 50000000;

```

```

clock_frequency_out : integer := 5000000
);
  Port ( clock_in : in STD_LOGIC;
reset_n : in STD_LOGIC;
          clock_out : out STD_LOGIC);
end component;

signal reset_n : std_logic;
signal cu_value : std_logic_vector(31 downto 0);
signal cu_enable : std_logic_vector(7 downto 0);
signal cu_dots : std_logic_vector(7 downto 0) := (others => '0');
signal clk_moltiplicatore : std_logic;
signal actual_start : std_logic := '0';

begin
reset_n <= not reset;

client_protocol : process(clk_moltiplicatore)
begin
  if (clk_moltiplicatore'event and clk_moltiplicatore = '1') then
    if actual_start = '1' then
      actual_start <= '0';
    else
      actual_start <= start;
    end if;
  end if;
end process;

clk_filter : clock_filter GENERIC MAP(clock_frequency_in => 100000000,
clock_frequency_out => 10000000)
  PORT MAP( clock_in => clock, reset_n => reset_n, clock_out =>
clk_moltiplicatore);

seven_segment_array: display_seven_segments GENERIC MAP(
clock_frequency_in => 10000000, --qui inserisco i parametri effettivi (clock
della board e clock in uscita desiderato)
clock_frequency_out => 50000
)
PORT MAP(
clock => clock,
reset_n => reset_n,
value => cu_value,
enable => cu_enable,
dots => cu_dots,
anodes => anodes,
cathodes => cathodes
);

cu: ctrl_unit PORT MAP(
clock => clk_moltiplicatore,
reset_n => reset_n,
start => actual_start,
A => A,

```

```

B => B,
value => cu_value,
enable => cu_enable
);

```

end Structural;

A questo punto, la control unit:

```

entity ctrl_unit is
  Port (
    clock : in STD_LOGIC;
    reset_n : in STD_LOGIC;
    start : in STD_LOGIC;
    A : in STD_LOGIC_VECTOR(7 downto 0);
    B : in STD_LOGIC_VECTOR(7 downto 0);
    value : out STD_LOGIC_VECTOR(31 downto 0);
    enable : out STD_LOGIC_VECTOR(7 downto 0)
  );
end ctrl_unit;

architecture Structural of ctrl_unit is

signal reset : std_logic;
signal prod : std_logic_vector(15 downto 0);

component Robertson_multiplier is
Generic(N : integer := 8; M : integer :=8); --N numero bit moltiplicando, M numero
bit moltiplicatore
  Port ( A : in STD_LOGIC_VECTOR(N-1 downto 0); --moltiplicando
         B : in STD_LOGIC_VECTOR(M-1 downto 0); --moltiplicatore
         START : in STD_LOGIC;
         CLK : in STD_LOGIC;
         RST : in STD_LOGIC;
         P : out STD_LOGIC_VECTOR(N+M-1 downto 0));
end component;

begin

reset <= not reset_n;
enable <= "00001111";
value(31 downto 16) <= (others => '0');
value(15 downto 0) <= prod;

moltiplicatore : Robertson_multiplier port map(
  CLK => clock, A => A, B => B, RST => reset, START => start, P => prod
);

end Structural;

```

Riportiamo infine i **constraints** utilizzati:

```

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

##Switches
set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports {
A[0] }];
#IO_L24N_T3_RS0_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 {
B[0] }];
#IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8       IOSTANDARD LVCMOS18 } [get_ports {
B[1] }];
#IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16      IOSTANDARD LVCMOS33 } [get_ports {
B[2] }];
#IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13      IOSTANDARD LVCMOS33 } [get_ports {
B[3] }];
#IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6       IOSTANDARD LVCMOS33 } [get_ports {
B[4] }];
#IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12      IOSTANDARD LVCMOS33 } [get_ports {
B[5] }];
#IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11      IOSTANDARD LVCMOS33 } [get_ports {
B[6] }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10      IOSTANDARD LVCMOS33 } [get_ports {
B[7] }];
#IO_L21P_T3_DQS_14 Sch=sw[15]

##7 segment display
set_property -dict { PACKAGE_PIN T10      IOSTANDARD LVCMOS33 } [get_ports {
cathodes[0] }];
#IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10      IOSTANDARD LVCMOS33 } [get_ports {
cathodes[1] }];
#IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16      IOSTANDARD LVCMOS33 } [get_ports {
cathodes[2] }];
#IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13      IOSTANDARD LVCMOS33 } [get_ports {
cathodes[3] }];
#IO_L17P_T2_A26_15 Sch=cd

```

```

set_property -dict { PACKAGE_PIN P15    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[6] }]; #IO_L4P_T0_D04_14 Sch=cg
set_property -dict { PACKAGE_PIN H15    IO_STANDARD LVCMOS33 } [get_ports {
cathodes[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
set_property -dict { PACKAGE_PIN J17    IO_STANDARD LVCMOS33 } [get_ports {
anodes[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18    IO_STANDARD LVCMOS33 } [get_ports {
anodes[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9     IO_STANDARD LVCMOS33 } [get_ports {
anodes[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14    IO_STANDARD LVCMOS33 } [get_ports {
anodes[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
set_property -dict { PACKAGE_PIN P14    IO_STANDARD LVCMOS33 } [get_ports {
anodes[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14    IO_STANDARD LVCMOS33 } [get_ports {
anodes[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2     IO_STANDARD LVCMOS33 } [get_ports {
anodes[6] }]; #IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13    IO_STANDARD LVCMOS33 } [get_ports {
anodes[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons
set_property -dict { PACKAGE_PIN N17    IO_STANDARD LVCMOS33 } [get_ports {
reset }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18    IO_STANDARD LVCMOS33 } [get_ports {
start }]; #IO_L4N_T0_D05_14 Sch=btnc

```