

|  |
| --- |
| Immagine che contiene testo  Descrizione generata automaticamente  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](#_Toc57417549)

[Esercizio 1 5](#_Toc57417550)

[1.1 Traccia 5](#_Toc57417551)

[1.2 Soluzione 5](#_Toc57417552)

[1.3 Codice 6](#_Toc57417553)

[1.3.1 6](#_Toc57417554)

[1.4 Simulazione 9](#_Toc57417555)

[Esercizio 2 13](#_Toc57417556)

[2.1 Traccia 13](#_Toc57417557)

[2.2 Soluzione 13](#_Toc57417558)

[2.2.1 13](#_Toc57417559)

[2.2.2 15](#_Toc57417560)

[2.3 Codice 16](#_Toc57417561)

[2.3.1 16](#_Toc57417562)

[2.3.2 22](#_Toc57417563)

[2.4 Simulazione 26](#_Toc57417564)

[2.4.1 26](#_Toc57417565)

[2.4.2 29](#_Toc57417566)

[Esercizio 3 33](#_Toc57417567)

[3.1 Traccia 33](#_Toc57417568)

[3.2 Soluzione 33](#_Toc57417569)

[3.3 Codice 34](#_Toc57417570)

[3.4 Simulazione 39](#_Toc57417571)

[3.5 Sintesi su FPGA 41](#_Toc57417572)

[Esercizio 4 47](#_Toc57417573)

[4.1 Traccia 47](#_Toc57417574)

[4.2 Soluzione 47](#_Toc57417575)

[4.3 Codice 48](#_Toc57417576)

[4.3.1 48](#_Toc57417577)

[4.3.2 52](#_Toc57417578)

[4.4 Simulazione 53](#_Toc57417579)

[Esercizio 5 57](#_Toc57417580)

[5.1 Traccia 57](#_Toc57417581)

[5.2 Soluzione 57](#_Toc57417582)

[5.3 Codice 60](#_Toc57417583)

[5.3.1 Logica Cablata 60](#_Toc57417584)

[5.3.2 Logica Microprogrammata 68](#_Toc57417585)

[5.4 Simulazione 82](#_Toc57417586)

[5.4.1 Simulazione logica cablata 82](#_Toc57417587)

[5.4.2 Simulazione logica microprogrammata 85](#_Toc57417588)

[Esercizio 6 89](#_Toc57417589)

[6.1 Traccia 89](#_Toc57417590)

[6.2 Soluzione 89](#_Toc57417591)

[6.3 Codice 90](#_Toc57417592)

[6.4 Simulazione 94](#_Toc57417593)

[Esercizio 7 97](#_Toc57417594)

[7.1 Traccia 97](#_Toc57417595)

[7.2 Soluzione 97](#_Toc57417596)

[7.3 Codice 99](#_Toc57417597)

[7.4 Simulazione 107](#_Toc57417598)

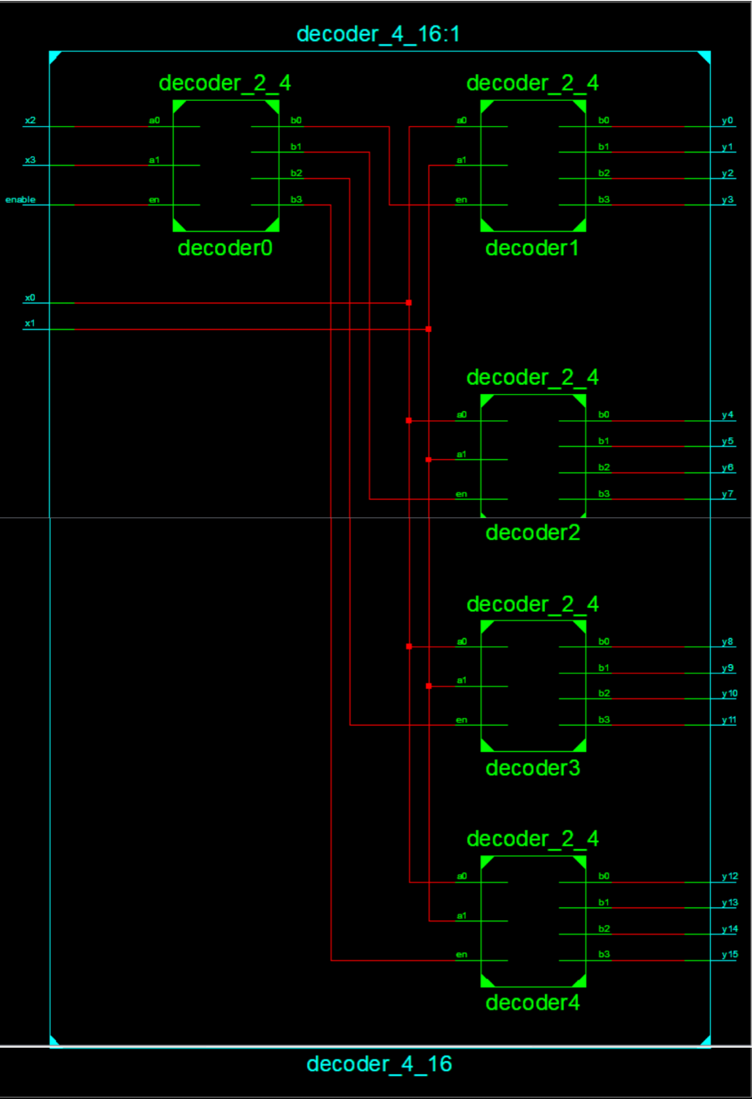
## Esercizio 1

### 1.1 Traccia

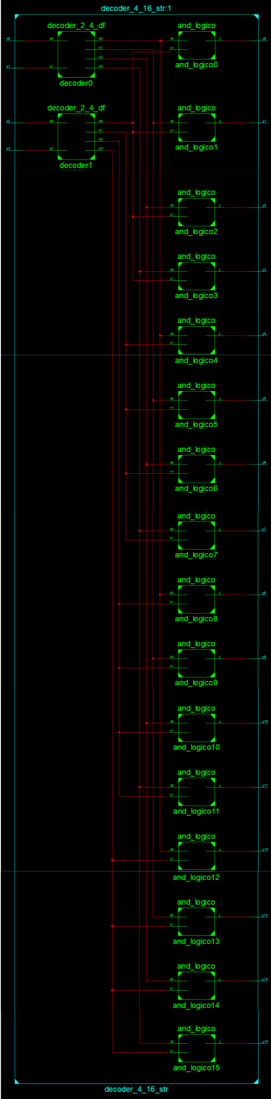
Si progetti un decoder 4:16 utilizzando componenti decoder 2:4 opportunamente interconnessi a) in una struttura ad albero e b) in una struttura a semiselezione.

### 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:



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\*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.



### 1.3 Codice

#### 1.3.1

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\_df 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\_df;

architecture Dataflow of decoder\_2\_4\_df 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;

s0 : 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),

s0 => s,

yv => u

);

dec0: decoder\_2\_4

Port map(

x1 => x(1),

x0 => x(0),

s0 => u(0),

yv => y(3 downto 0)

);

dec1: decoder\_2\_4

Port map(

x1 => x(1),

x0 => x(0),

s0 => u(1),

yv => y(7 downto 4)

);

dec2: decoder\_2\_4

Port map(

x1 => x(1),

x0 => x(0),

s0 => u(2),

yv => y(11 downto 8)

);

dec3: decoder\_2\_4

Port map(

x1 => x(1),

x0 => x(0),

s0 => u(3),

yv => y(15 downto 12)

);

end Tree;

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;

s0 : 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), s0 => s, yv => u1

);

dec2 : decoder\_2\_4 port map(

x1 => x(1), x0 => x(0), s0 => 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";

x <= "0001";

wait for 10 ns;

assert y = "0000000000000010";

x <= "0010";

wait for 10 ns;

assert y = "0000000000000100";

x <= "0011";

wait for 10 ns;

assert y = "0000000000001000";

x <= "0100";

wait for 10 ns;

assert y = "0000000000010000";

x <= "0101";

wait for 10 ns;

assert y = "0000000000100000";

x <= "0110";

wait for 10 ns;

assert y = "0000000001000000";

x <= "0111";

wait for 10 ns;

assert y = "0000000010000000";

x <= "1000";

wait for 10 ns;

assert y = "0000000100000000";

x <= "1001";

wait for 10 ns;

assert y = "0000001000000000";

x <= "1010";

wait for 10 ns;

assert y = "0000010000000000";

x <= "1011";

wait for 10 ns;

assert y = "0000100000000000";

x <= "1100";

wait for 10 ns;

assert y = "0001000000000000";

x <= "1101";

wait for 10 ns;

assert y = "0010000000000000";

x <= "1110";

wait for 10 ns;

assert y = "0100000000000000";

x <= "1111";

wait for 10 ns;

assert y = "1000000000000000";

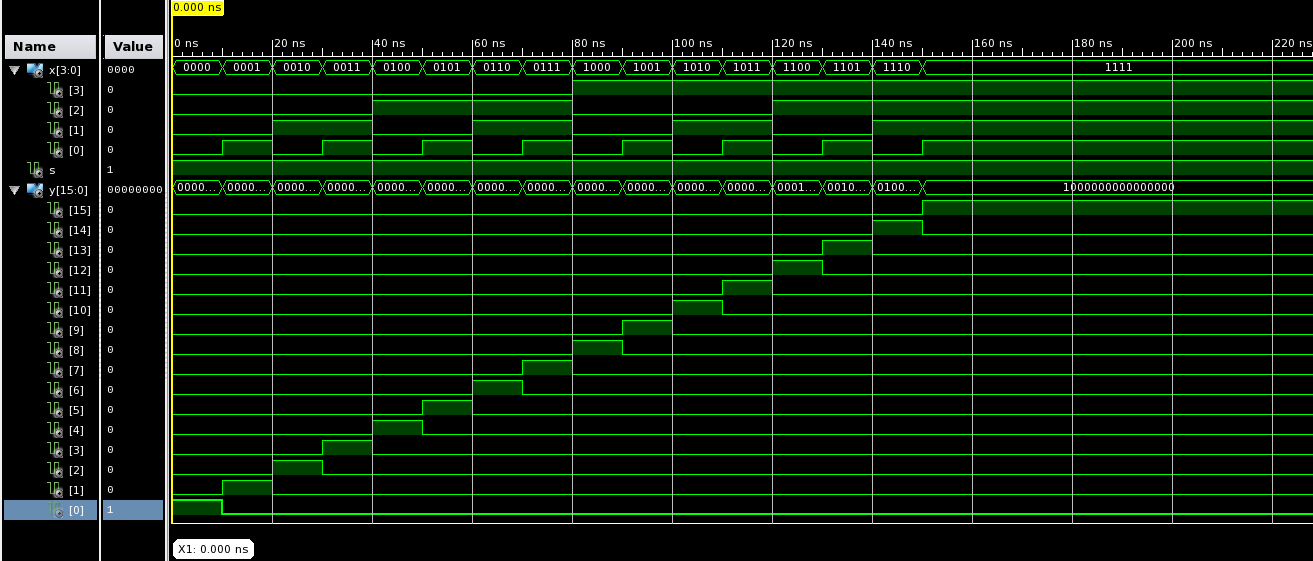
wait;

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

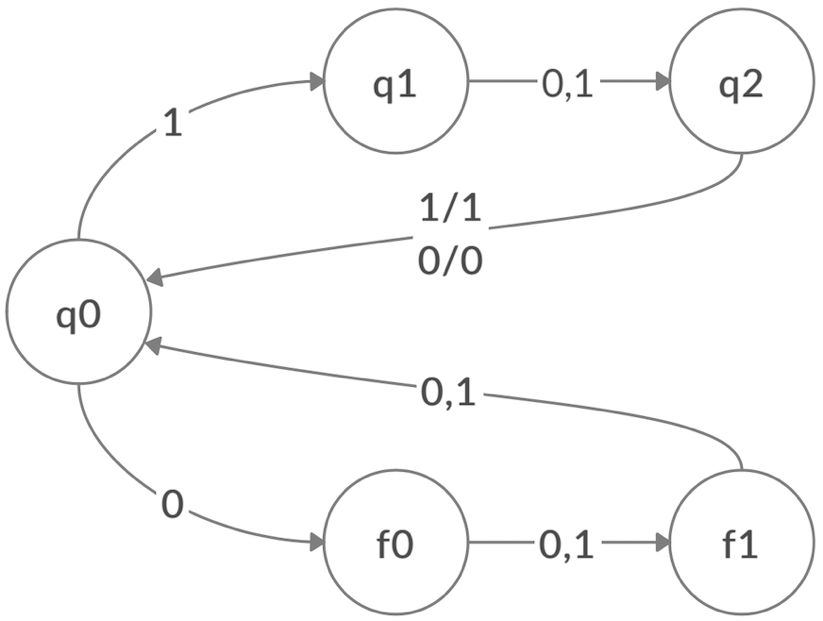
Si vuole progettare un riconoscitore di sequenza come macchina sincrona a sincronizzazione esterna: la macchina fornisce uscita alta quando viene riconosciuta la sequenza 1-10, e le sequenze possono sovrapporsi (esempio: la sequenza 11101010 produrrebbe un’uscita alta in corrispondenza del quarto, sesto 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

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 q0 in corrispondenza della ricezione di un numero di bit pari alla lunghezza della sequenza che si vuole riconoscere. Dal punto di vista della tempificazione, 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.

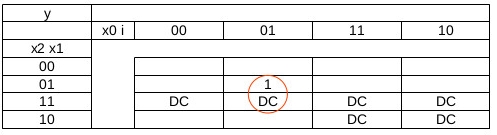
A questo punto, visto che è richiesta una descrizione strutturale con componenti AND e OR, è stato necessario effettuare la sintesi mediante mappe di Karnaugh.

Nelle seguenti mappe di Karnaugh è stato indicato con DC l’eventuale presenza di punti di non specificazione *“don’t care”.*

Codifica degli stati:

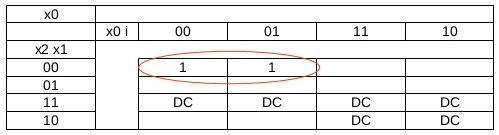
*q0 = 000, q1 = 001, q2 = 010, f0 = 011, f1 = 100, con variabili di stato rispettivamente x2, x1 e x0.*

Uscita:



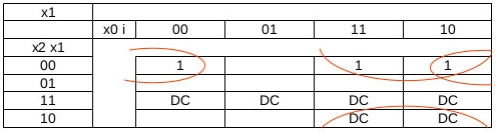
*y = x1 and (not x0) and i*

Variabile di stato x0:



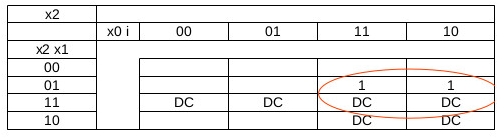
*x0 = (not x2) and (not x1) and (not x0)*

Variabile di stato x1:



*x1 = ((not x1) and x0) or ((not x2) and (not x1) and (not i))*

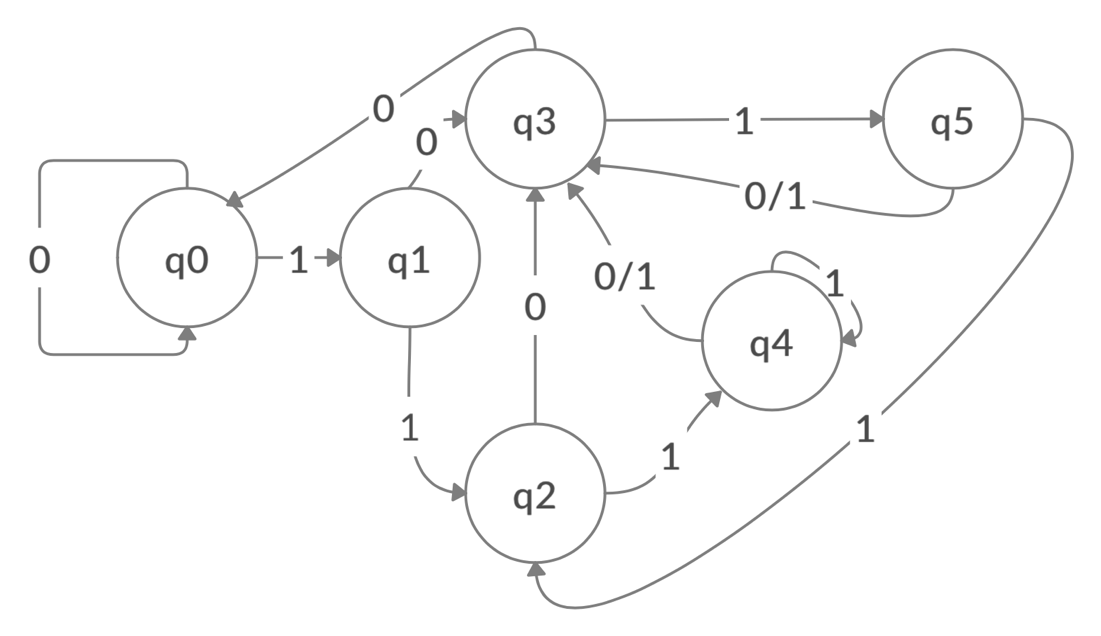
Variabile di stato x2:



*x2 = x1 and x0*

#### 2.2.2

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, f0 = 011, f1 = 100, con variabili di stato rispettivamente x2, x1 e x0.*

Uscita:

**

*y = x2 and (not i)*

Variabile di stato x0:



*x0 = (x1 and (not x0) and (not i)) or (x2 and (not i)) or (not(x1) and x0 and not(i)) or*

*(x1 and x0 and i) or (not(x2) and not(x1) and not(x0) and i)*

Variabile di stato x1:



*x1 = (x1 and not(x0) and not(i)) or (x2 and not(i)) or ((not x1) and x0)*

Variabile di stato x2:



*x2 = (x1 and i) or (x2 and not(x0) and i)*

### 2.3 Codice

#### 2.3.1

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;

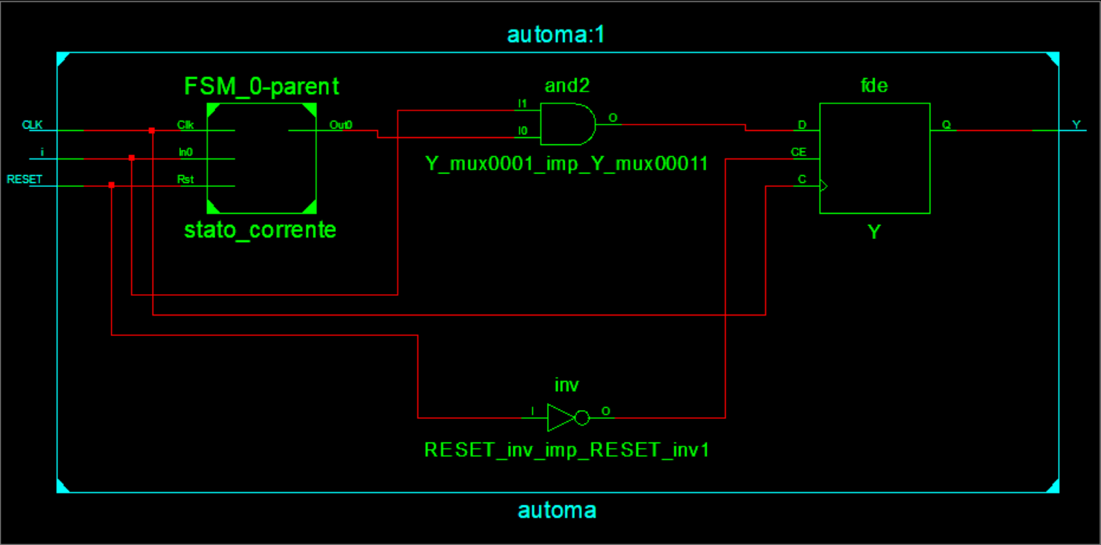
end if;

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,

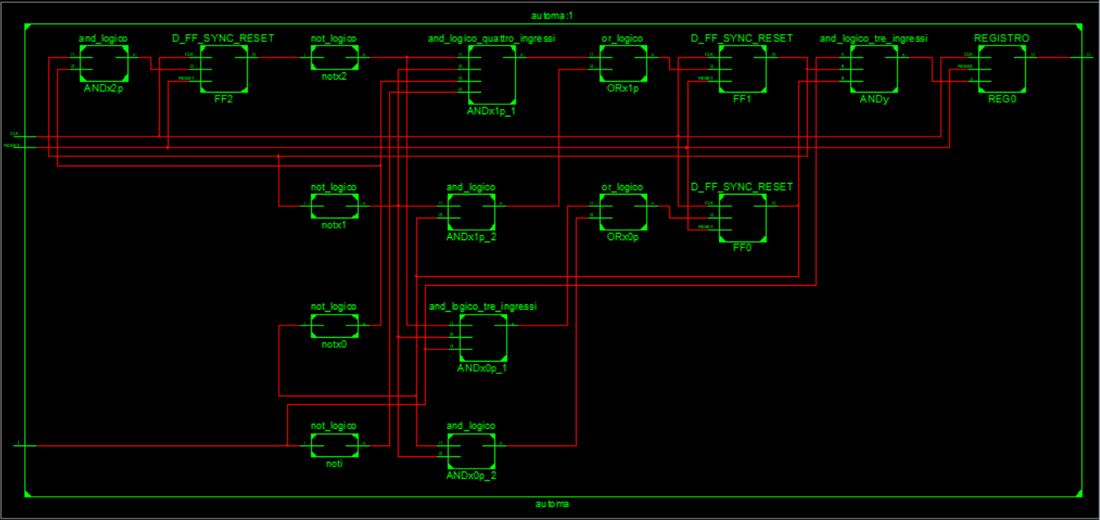
rst => rst,

q => y

);

end Structural;

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



#### 2.3.2

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;

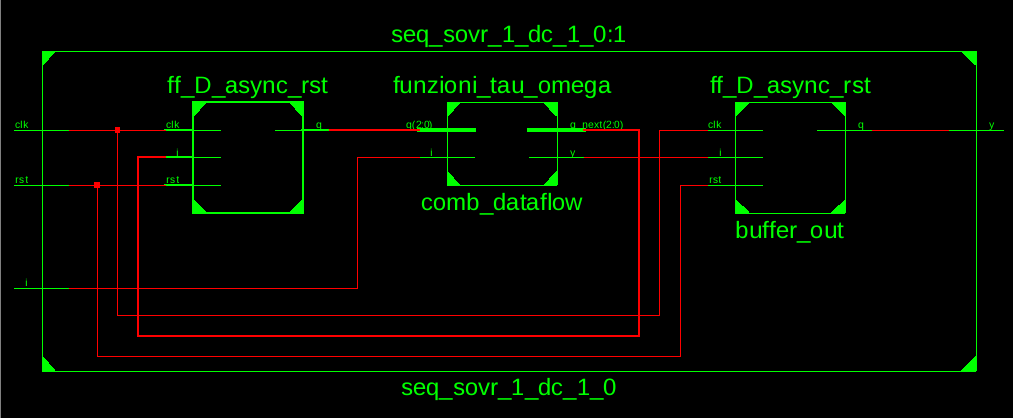
buffer\_out : ff\_D\_async\_rst port map(

i => y\_tmp, clk => clk, rst => rst, q => y

);

end Hybrid;

L’architettura RTL ottenuta è la seguente:



### 2.4 Simulazione

#### 2.4.1

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"

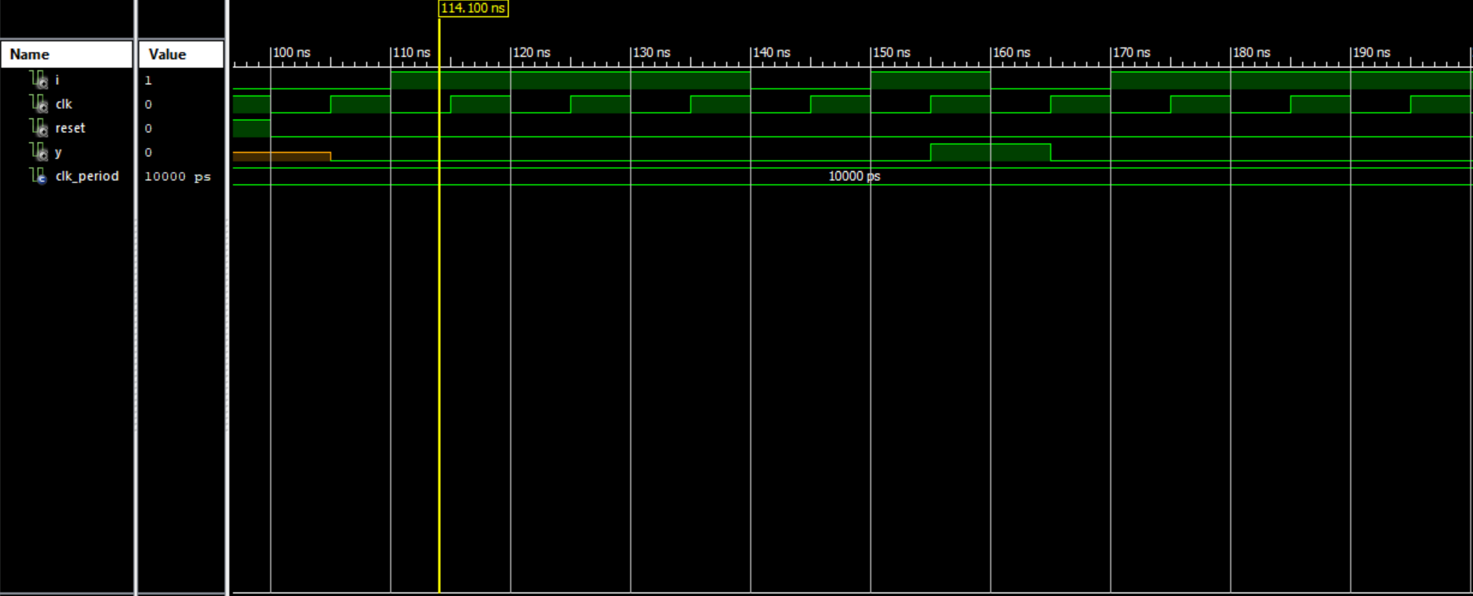
severity failure;

wait;

end process;

END;

I risultati della simulazione sono riportati nella seguente figura:



#### 2.4.2

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"

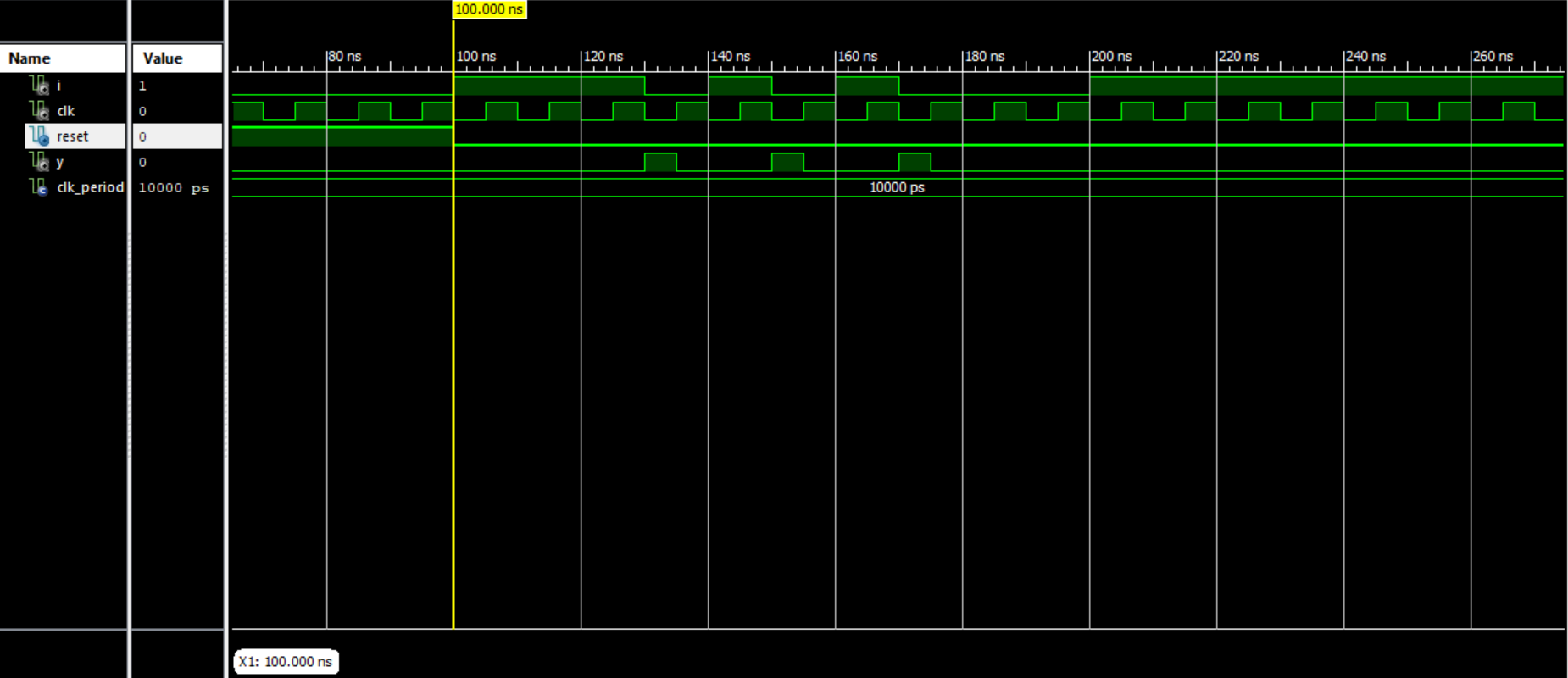
severity failure;

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 2N 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 = '0' and clk'event) 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 = '0' and clk'event) 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

);

-- 0111100 = 60 in decimale

cont\_sec : contatore\_mod\_M generic map(n\_bit => 6, M => "111100") 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 => "111100") port map(

clk => clk\_min, rst => rst, par => set\_m, set => set, overflow => clk\_h,

y => m

);

-- 11000 = 24 in decimale

cont\_h : contatore\_mod\_M generic map(n\_bit => 5, M => "11000") 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;

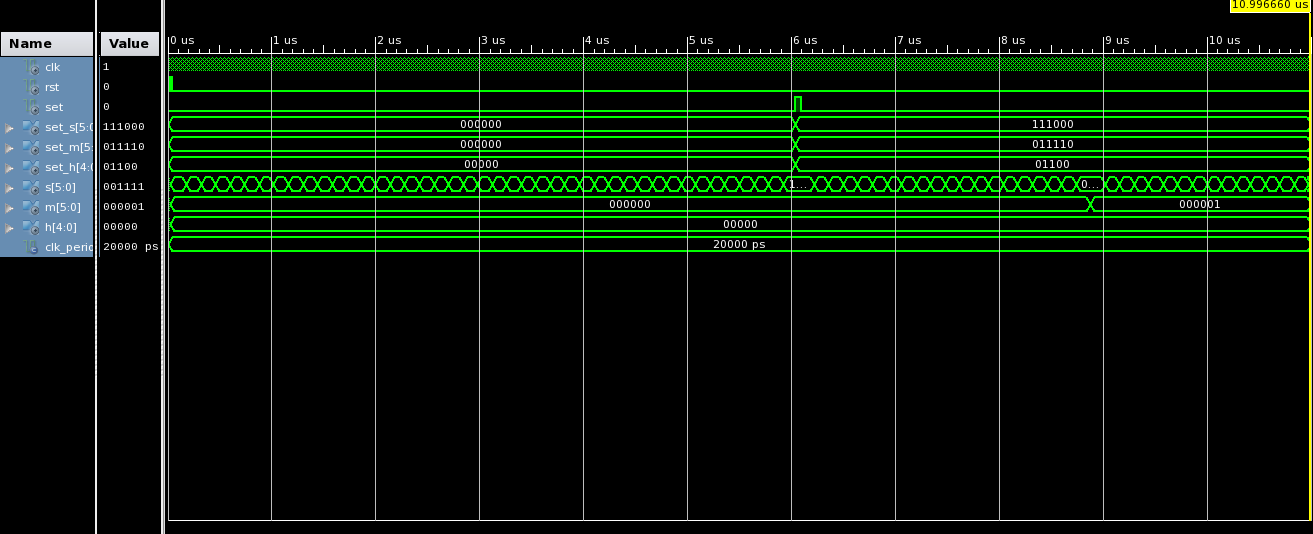
set <= '0';

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

);

Un’altra modifica che è stata fatta riguarda i valori di M dei contatori: per quello della simulazione sono stati 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).

## 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

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');

signal T : STD\_LOGIC\_VECTOR(0 to N-1) := (others => '0');

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;

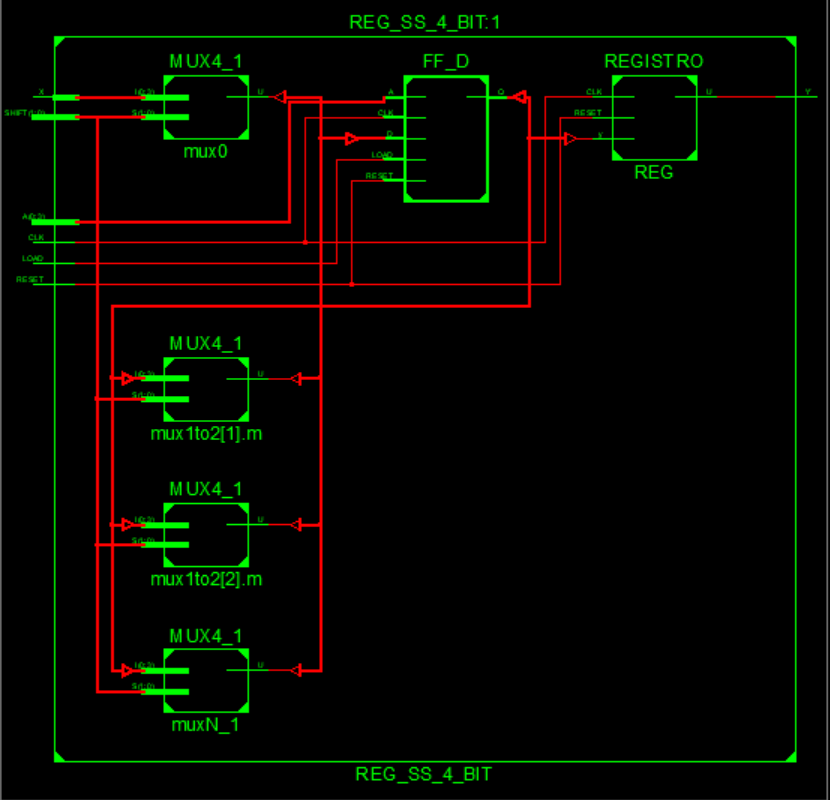
architecture Dataflow of MUX2\_1 is

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

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 microprogrammata**.

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:

A1 = A0 – B

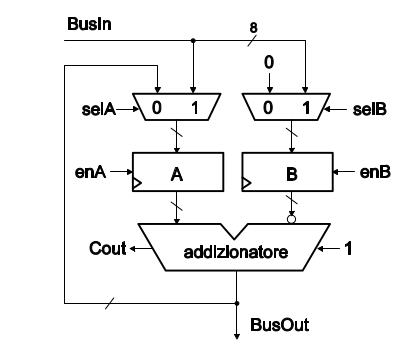
A2 = A1 – B

…

An = An-1 – B

Quand’è che si deve fermare? Risposta: quando risulta An < 0, e quindi risulterà che:

An-1 = A0 mod B, dove A0 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 predispone 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

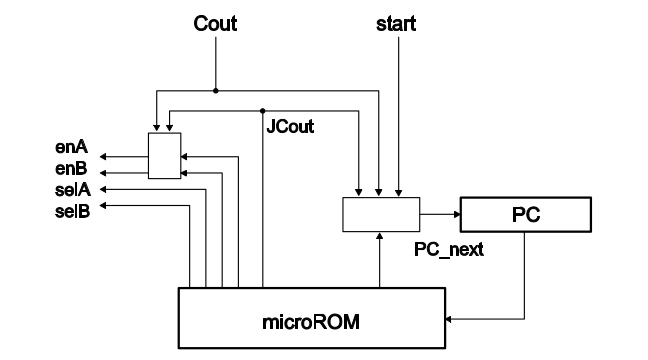
Il secondo modo prevede che il client invii una conferma della conferma, cioè un altro ack, dopo aver ricevuto l’ack del server, e quindi che il server attenda l’ack del client prima di procedere. Entrambi i modi sono validi, ma nel primo caso bisogna stare più attenti ai tempi relativi delle operazioni. 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 predispone 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;

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 ottenere 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 = '1' and clk'event) 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';

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\_parallelo\_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\_parallelo\_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\_parallelo\_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\_parallelo\_N;

architecture Behavioral of mux\_2\_1\_parallelo\_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 Behavioral;

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 Behavioral 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 Behavioral;

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 Behavioral of mux\_2\_1 is

begin

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

end Behavioral;

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\_O 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;

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 behevioral.

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 isA\_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';

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\_O 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\_O;

architecture structural of U\_O 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;

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 = '1' and clk'event) then

if (ready = '0') then

PC <= (others => '0');

elsif (jcout = '1' and cout = '0') then

PC <= "101";

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 tempificazione 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';

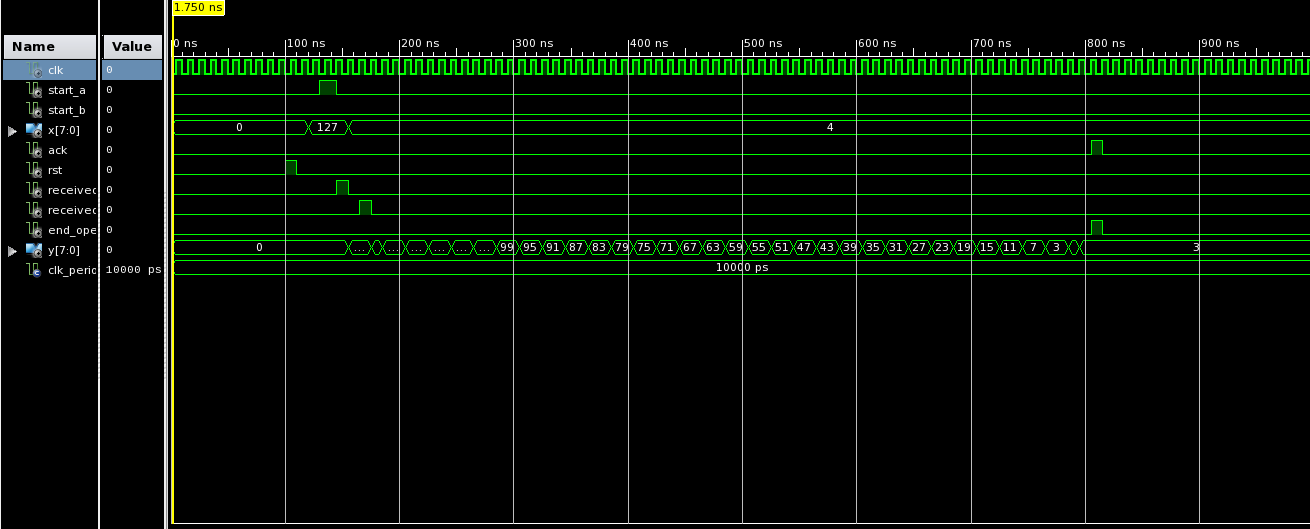
assert y = "00000011";

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 microprogrammata.

ENTITY system\_tb IS

END system\_tb;

architecture behavior OFsystem\_tb IS

COMPONENT systemù

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 EOOP = '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.

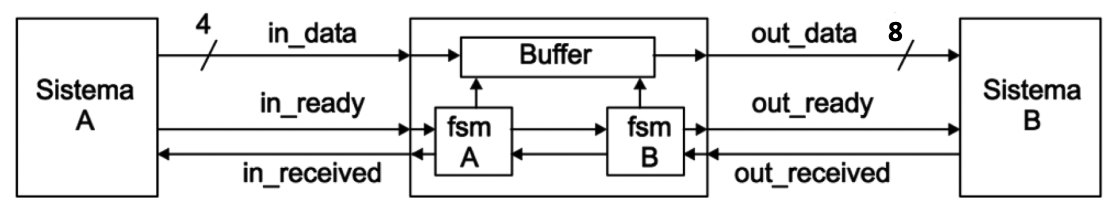
Le due unità non sono dotate di un collegamento diretto costituito da un bus di 16 bit, ma l’unità A possiede un bus di 4 bit in uscita e l’unità B possiede un bus di 8 bit in ingresso. Per questa ragione, il trasferimento deve avvenire in più passi facendo uso di un buffer.

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 progetto 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 grafico:



L’automa fsmA comunica con il sistema A mediante protocollo asincrono 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 per far partire il sistema A e fargli iniziare a produrre le parole che 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, anch’esso mediante un protocollo 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 consecutivamente 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 quiindi senza attendere che vengano riempiti tutti e 16 i 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 o 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 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 il 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'; current\_state <= FSMA\_ATTESA\_INRDY0;

When 3 => count:=0; buffer\_full <= '1'; current\_state <= FSMA\_ATTESA\_BUFFERPICKED;

end case;

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;

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\_ARDY1;

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;

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).

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);

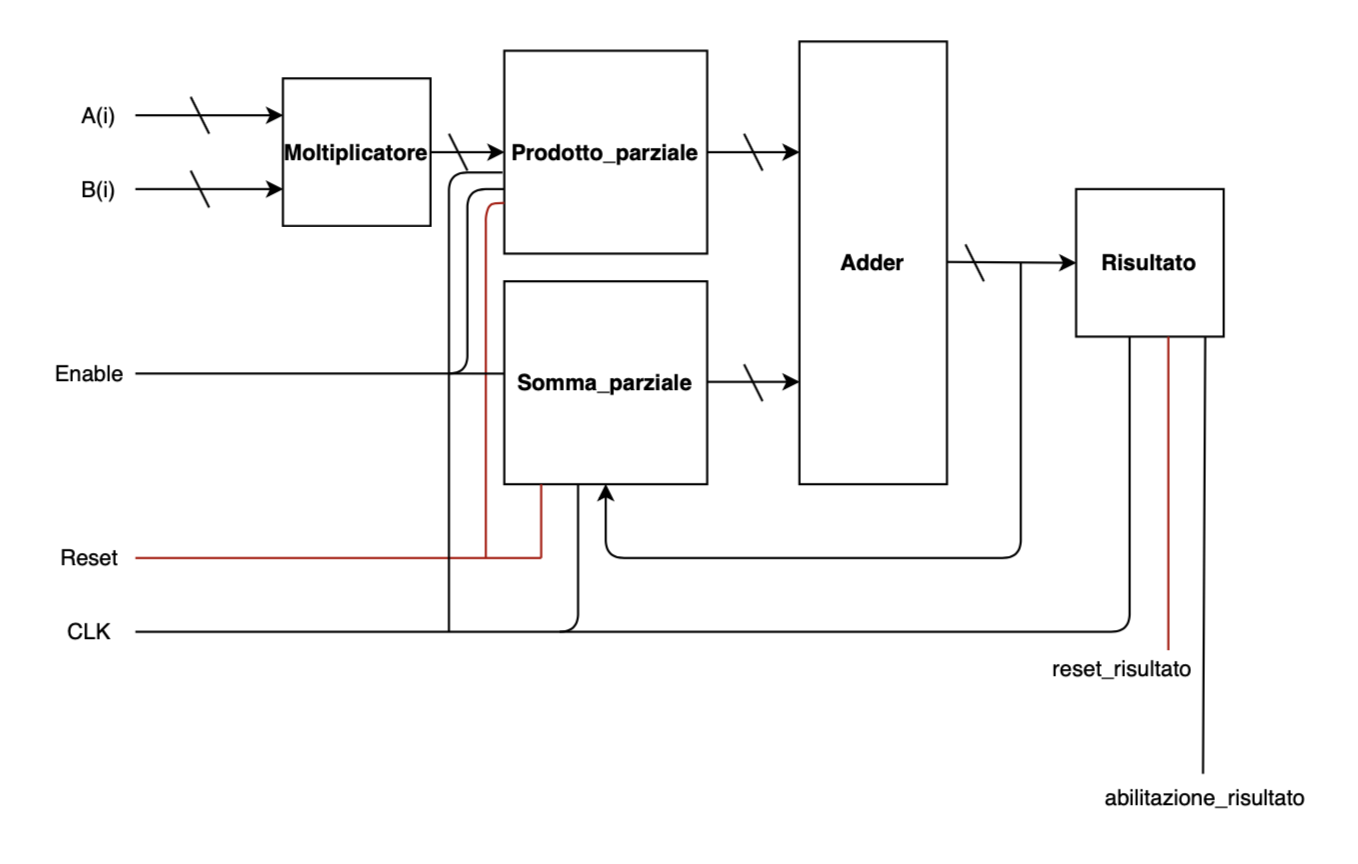
3. Ciascuna coppia di vettori viene ricevuta da un’entità produttore mediante handshaking (e gestita in modalità parallela o seriale a seconda dell’architettura scelta);

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

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 ciascuno 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 un 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.

### 7.3 Codice

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(7 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(7 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\_O: 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 riportati 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(7 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(7 downto 0) := (others=>'0');

signal AperB : STD\_LOGIC\_VECTOR(7 downto 0) := (others=>'0');

signal S\_Parziale : STD\_LOGIC\_VECTOR(7 downto 0) := (others=>'0');

signal P\_Parziale : STD\_LOGIC\_VECTOR(7 downto 0) := (others=>'0');

signal Accumulatore : STD\_LOGIC\_VECTOR(7 downto 0) := (others=>'0');

Component Adder is

Port ( A : in STD\_LOGIC\_VECTOR(7 downto 0);

B : in STD\_LOGIC\_VECTOR(7 downto 0);

ApiuB : out STD\_LOGIC\_VECTOR(7 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(7 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(7 downto 0);

Q : out STD\_LOGIC\_VECTOR(7 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(7 downto 0));

end Moltiplicatore;

architecture Behavioral of Moltiplicatore is

signal AperB\_tmp : STD\_LOGIC\_VECTOR(15 downto 0) := (others => '0');

begin

AperB\_tmp <= std\_logic\_vector(unsigned(A) \* unsigned(B));

AperB <= AperB\_tmp(7 downto 0);

end Behavioral;

entity Adder is

Port ( A : in STD\_LOGIC\_VECTOR(7 downto 0);

B : in STD\_LOGIC\_VECTOR(7 downto 0);

ApiuB : out STD\_LOGIC\_VECTOR(7 downto 0));

end Adder;

architecture Behavioral of Adder is

begin

ApiuB <= 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(7 downto 0);

Q : out STD\_LOGIC\_VECTOR(7 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

reg7to0: for i in 7 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;

### 7.4 Simulazione

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(7 downto 0)

);

END COMPONENT;

SIGNAL START : std\_logic := '0';

SIGNAL CLK : std\_logic := '0';

SIGNAL AB : std\_logic\_vector(7 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="00011110"

report "errore0"

severity failure;

wait for 4\*CLK\_period;

assert AB="00110101"

report "errore0"

severity failure;

wait for 4\*CLK\_period;

assert AB="00111010"

report "errore0"

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:

Immagine che contiene elettronico, circuito, computer

Descrizione generata automaticamente