

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica



ELABORATO DI ARCHITETTURA DEI SISTEMI DIGITALI

Prof.ssa Alessandra De Benedictis

a.a. 2024-25

Studenti:

Giovanni Falco M63001713

Felice Palmiero M63001744

Mauro Lauritano M63001749

Vincenzo Laudiero M63001791

Sommario

Capitolo 1: reti combinatorie elementari.....	6
Esercizio 1: Multiplexer 16:1.....	6
Progetto e architettura.....	6
Multiplexer 4:1: aspetti teorici.....	7
Multiplexer 4:1: implementazione VHDL.....	7
Multiplexer 16:1: aspetti teorici.....	8
Multiplexer 16:1: implementazione VHDL.....	8
Simulazione.....	12
Esercizio 1.2: Rete di interconnessione a 16 sorgenti e 4 destinazioni.....	16
Progetto e architettura.....	16
Implementazione.....	18
Simulazione.....	20
Esercizio 1.3: Implementazione della rete di interconnessione su board.....	29
Sintesi su board di sviluppo.....	29
Esercizio 2 : Sistema ROM+M.....	37
Esercizio 2.1.....	37
Progetto e architettura.....	37
Implementazione.....	37
Simulazione.....	40
Esercizio 2.2: Implementazione sistema ROM+M su board.....	42
Sintesi su board di sviluppo.....	43
Capitolo 2: Reti sequenziali elementari.....	44
Esercizio 3 : Riconoscitore di sequenze.....	44
Progetto e architettura.....	45
Implementazione.....	47
Simulazione.....	52
Timing analysis.....	56
Esercizio 3.2: Implementazione riconoscitore di sequenza su board.....	57
Sintesi su board di sviluppo.....	57
Esercizio 4: Shift Register.....	62
Progetto e architettura.....	62
Implementazione (Behavioral).....	63
Simulazione (Behavioral).....	65
Implementazione (Structural).....	68
Simulazione (structural).....	73
Esercizio 5: Cronometro.....	80
Progetto e architettura.....	80
Implementazione.....	81
Simulazione.....	83
Esercizio 5.2: implementazione su board.....	87
Progetto e architettura.....	87
Implementazione.....	88

Sintesi su board di sviluppo.....	97
Esercizio 6: Sistema di lettura-elaborazione-scrittura PO_PC.....	100
Progetto e architettura.....	100
Implementazione.....	101
Simulazione.....	107
Esercizio 6.2: implementazione su board.....	109
Sintesi su board di sviluppo.....	117
Capitolo 3: Macchine aritmetiche.....	119
Esercizio 7: Moltiplicatore di Booth su 8 bit.....	119
Progetto e architettura.....	119
Implementazione.....	122
Implementazione: Adder Subber.....	122
Implementazione: Shift Register 17 bit.....	123
Implementazione: Unità Operativa.....	125
Implementazione: Unità di Controllo.....	129
Implementazione: Entità Finale.....	133
Simulazione.....	135
Esercizio 7.2: Implementazione su board moltiplicatore di Booth.....	139
Implementazione.....	140
Sintesi su board di sviluppo.....	145
Capitolo 4: Comunicazione con handshaking.....	146
Esercizio 8 – Comunicazione con handshaking.....	146
Progetto e architettura.....	147
Sistema A.....	147
Componenti Sistema A.....	148
Counter.....	148
MemA.....	150
Entità A.....	151
Sistema B.....	154
Componenti Sistema B.....	155
Memoria B.....	155
Unità Operativa.....	157
Entity B.....	160
Simulazione.....	164
Capitolo 5: Processore.....	167
Esercizio 9.1.....	167
Processore MIC-1.....	167
Le microistruzioni:.....	171
Unità di Controllo.....	172
Esercizio IADD e POP (Analisi e simulazione).....	173
IADD.....	173
Simulazione IADD.....	174
POP.....	175
Simulazione POP.....	176
IADD modificata.....	176

Capitolo 6: Interfaccia Seriale.....	179
Esercizio 10.1.....	179
Progetto e architettura.....	179
Implementazione.....	180
Sistema complessivo.....	180
Sistema A.....	182
Sistema B.....	193
Simulazione.....	203
Capitolo 7: Switch Multistadio.....	206
Esercizio 11.1.....	206
Progetto e architettura.....	206
Implementazione.....	207
Simulazione.....	216
Capitolo 8: Esercizio d'esame.....	219
Esercizio 12 (prova di esame del 19 dicembre 2024).....	219
MACCHINA CARRY LOOK AHEAD.....	220
implementazione.....	221
SISTEMA A.....	222
Progetto e architettura.....	222
SISTEMA B.....	223
Progetto e architettura.....	224
SISTEMA COMPLESSIVO.....	225
Implementazione.....	225
Sistema complessivo:.....	225
Sistema A:.....	227
Unità di controllo A:.....	230
Sistema B:.....	234
Unità di controllo B:.....	238
Registro:.....	241
Simulazione.....	242
Clock A più veloce:.....	242
Clock B più veloce:.....	245
Appendice.....	249
MUX2_1.....	249
Progetto e architettura.....	249
Implementazione.....	249
MULTIPLEXER4_1.....	250
Progetto e architettura.....	250
Implementazione.....	250
DEMUX1_2.....	251
Progetto e architettura.....	251
Implementazione.....	252
DEMUX1_4.....	253
Progetto e architettura.....	253

Implementazione.....	253
ROM.....	254
Progetto e architettura.....	254
Implementazione.....	255
Full Adder.....	256
Progetto e architettura.....	256
Implementazione.....	256
Adder Carry Ripple.....	257
Progetto e architettura.....	257
Implementazione.....	257
Registro PIPO.....	259
Progetto e architettura.....	259
Implementazione.....	259
Contatore Modulo 8.....	261
Progetto e architettura.....	261
Implementazione.....	261
Contatore generico.....	262
Progetto e architettura.....	262
Implementazione.....	262
Memoria.....	264
Progetto e architettura.....	264
Implementazione.....	264
Button Debouncer.....	266
Progetto e architettura.....	266
Implementazione.....	266
Divisore di frequenza.....	269
Progetto e architettura.....	269
Implementazione.....	269

Capitolo 1: reti combinatorie elementari

Esercizio 1: Multiplexer 16:1

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

Progetto e architettura

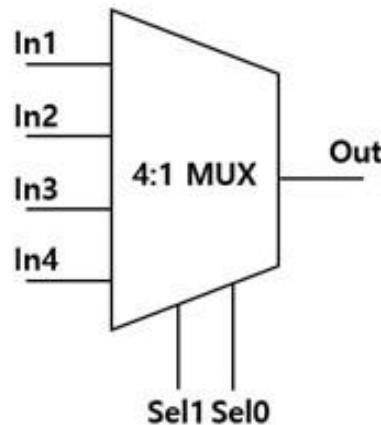
Il nostro obiettivo è la realizzazione di un multiplexer indirizzabile 16:1 - una macchina combinatoria notevole caratterizzata da 16 ingressi, 4 linee di selezione ed 1 uscita - avendo a disposizione dei multiplexer 4:1. Pertanto, verranno utilizzati 5 multiplexer 4:1 opportunamente connessi tra di loro. Riportiamo, di seguito, i vari componenti utilizzati coerentemente alla notazione usata in VHDL:

- MUX_4_1_1:
 - Input: [A0, ..., A3] a cui corrispondono gli ingressi [I(0), ..., I(3)] del mux 16:1;
 - Output: mux1;
 - Selezioni: S0, S1 a cui corrispondono i bit meno significativi (S0, S1) delle linee di selezione del mux 16:1;
- MUX_4_1_2:
 - Input: [A0, ..., A3] a cui corrispondono gli ingressi [I(4), ..., I(7)] del mux 16:1;
 - Output: mux2;
 - Selezioni: S0, S1 a cui corrispondono i bit meno significativi (S0, S1) delle linee di selezione del mux 16:1;
- MUX_4_1_3:
 - Input: [A0, ..., A3] a cui corrispondono gli ingressi [I(8), ..., I(11)] del mux 16:1;
 - Output: mux3;
 - Selezioni: S0, S1 a cui corrispondono i bit meno significativi (S0, S1) delle linee di selezione del mux 16:1;
- MUX_4_1_4:
 - Input: [A0, ..., A3] a cui corrispondono gli ingressi [I(12), ..., I(15)] del mux 16:1;
 - Output: mux4;
 - Selezioni: S0, S1 a cui corrispondono i bit meno significativi (S0, S1) delle linee di selezione del mux 16:1;
- MUX_4_1_5:
 - Input: [A0, ..., A3] a cui corrispondono le uscite [mux1, ..., mux4] dei precedenti mux 4:1;
 - Output: y, corrispondente all'unica uscita del multiplexer 16:1;
 - Selezioni: S0, S1 a cui corrispondono i bit più significativi (S2, S3) delle linee di selezione del mux 16:1;

E' doveroso precisare che i multiplexer 4:1 utilizzati possono essere – a loro volta – decomposti in multiplexer 2:1 e, quindi, possono essere implementati in maniera strutturale piuttosto che comportamentale.

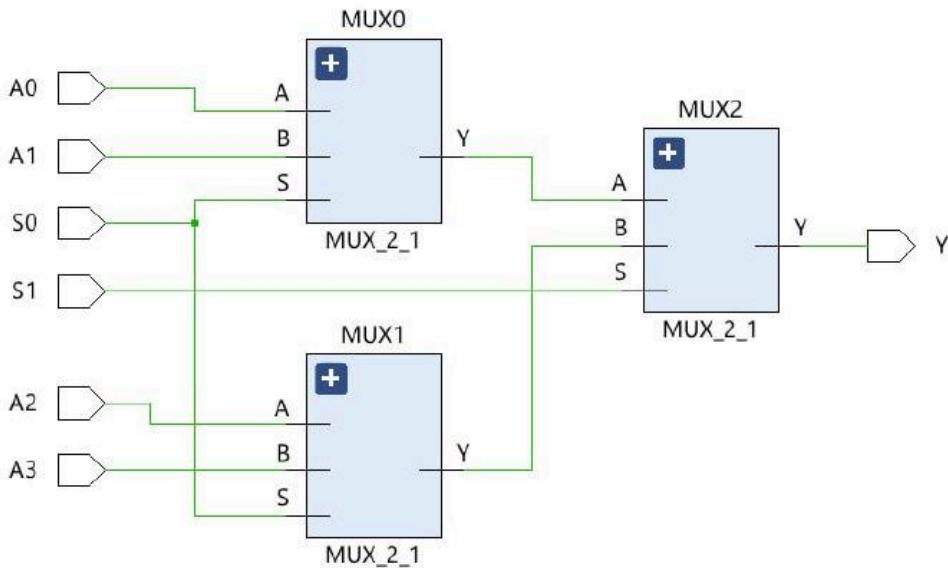
Multiplexer 4:1: aspetti teorici

Il multiplexer 4:1 è una macchina combinatoria notevole caratterizzata da 4 ingressi, 2 linee di selezione e un'unica uscita, come mostrato nella figura riportata di seguito.



Multiplexer 4:1: implementazione VHDL

Per l'implementazione VHDL del multiplexer 4:1 si è optato per un approccio di tipo strutturale. Sono stati usati 3 multiplexer 2:1 opportunamente collegati tra di loro. Riportiamo il codice VHDL e lo schematic prodotto per la suddetta realizzazione.

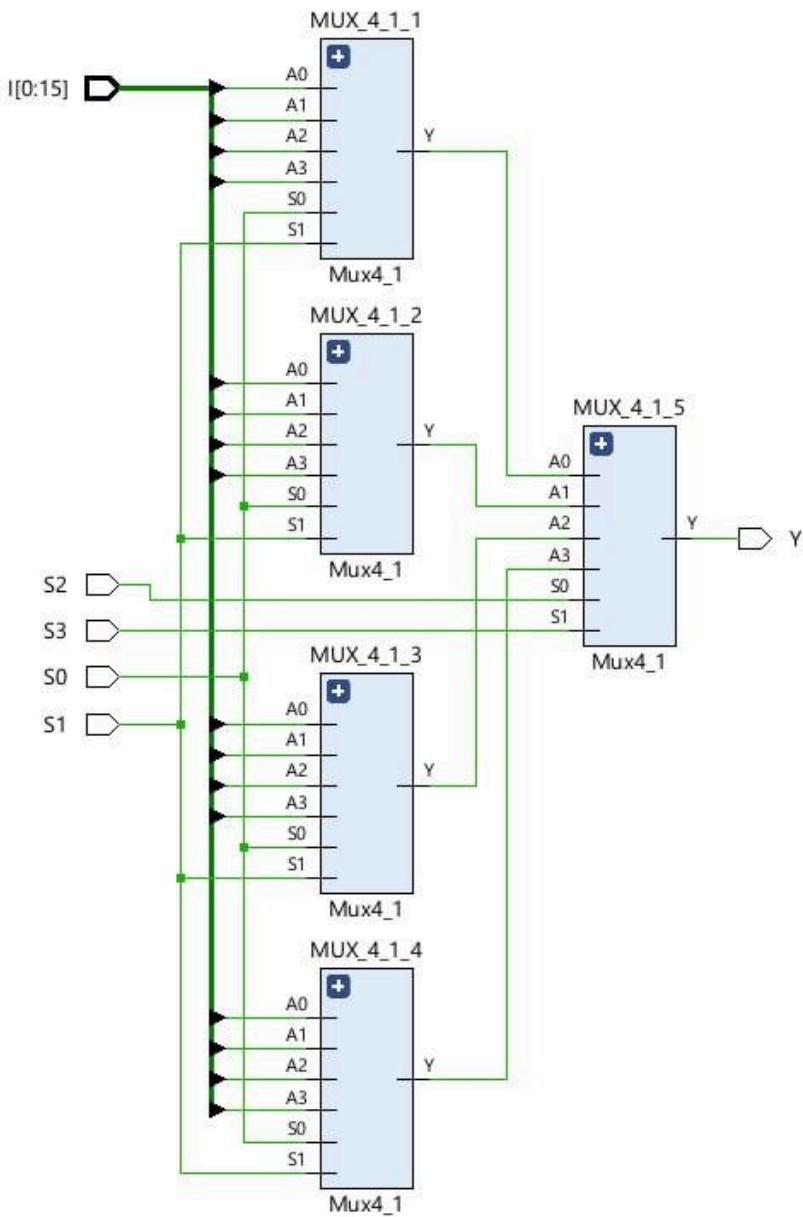


Multiplexer 16:1: aspetti teorici

Il multiplexer 16:1 è una macchina combinatoria notevole caratterizzata da 16 ingressi, 4 linee di selezione e un'unica uscita. Sfruttando la proprietà di modularità, il seguente multiplexer è stato realizzato a partire dall'interconnessione di 5 multiplexer 4:1.

Multiplexer 16:1: implementazione VHDL

Per l'implementazione VHDL del multiplexer 16:1, si è scelto di utilizzare una descrizione strutturale. L'entità Mux16_1 include 16 variabili d'ingresso, rappresentate dal vettore I (che va da I(0) a I(15)) , dichiarato come STD_LOGIC_VECTOR, e 4 variabili di selezione (S0, S1, S2, S3). L'uscita del multiplexer è una variabile Y di tipo STD_LOGIC. Rispetto all'implementazione del multiplexer 4:1, questa versione arricchisce l'architettura con l'istanziazione di 4 multiplexer 4:1, i cui risultati vengono combinati successivamente. In particolare, ogni multiplexer 4:1 è configurato per selezionare tra 4 ingressi, utilizzando i segnali di selezione S0 e S1. Le uscite di questi multiplexer 4:1 sono poi combinate tramite un secondo livello di multiplexing, utilizzando altri segnali di selezione (S2 e S3). Nel codice VHDL, i segnali mux1, mux2, mux3 e mux4 rappresentano le uscite intermedie dei 4 multiplexer 4:1, che vengono poi inviati a un quinto multiplexer 4:1 per produrre l'uscita finale Y dell'intera macchina. Riportiamo, di seguito lo schematico prodotto.



```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity declaration for the 16:1 MUX
entity Mux16_1 is
    Port ( I : in STD_LOGIC_VECTOR (0 to 15); -- 16 inputs
           S0 : in STD_LOGIC; -- Selection inputs
           Y : out STD_LOGIC);
end Mux16_1;

```

```
S1 : in STD_LOGIC;  
S2 : in STD_LOGIC;  
S3 : in STD_LOGIC;  
Y : out STD_LOGIC);  
end Mux16_1;
```

```
architecture Behavioral of Mux16_1 is
```

```
-- Component declaration for 4:1 MUX
```

```
component MUX4_1 is
```

```
port (  
    A0 : in STD_LOGIC;  
    A1 : in STD_LOGIC;  
    A2 : in STD_LOGIC;  
    A3 : in STD_LOGIC;  
    S0 : in STD_LOGIC;  
    S1 : in STD_LOGIC;  
    Y : out STD_LOGIC);
```

```
end component;
```

```
-- Signals to connect the outputs of the 4:1 MUXes
```

```
signal mux1, mux2, mux3, mux4 : STD_LOGIC;
```

```
begin
```

```
-- First level of 4:1 MUXes
```

```
MUX_4_1_1 : MUX4_1 port map (
```

```
    A0 => I(0),
```

```
    A1 => I(1),
```

```
    A2 => I(2),
```

```
    A3 => I(3),
```

```
    S0 => S0,
```

```
    S1 => S1,
```

```
Y => mux1
);

MUX_4_1_2 : MUX4_1 port map (
    A0 => I(4),
    A1 => I(5),
    A2 => I(6),
    A3 => I(7),
    S0 => S0,
    S1 => S1,
    Y => mux2
);

MUX_4_1_3 : MUX4_1 port map (
    A0 => I(8),
    A1 => I(9),
    A2 => I(10),
    A3 => I(11),
    S0 => S0,
    S1 => S1,
    Y => mux3
);

MUX_4_1_4 : MUX4_1 port map (
    A0 => I(12),
    A1 => I(13),
    A2 => I(14),
    A3 => I(15),
    S0 => S0,
    S1 => S1,
    Y => mux4
);
```

```
-- Second level of 4:1 MUX to combine the results
```

```
MUX_4_1_5 : MUX4_1 port map (
```

```
    A0 => mux1,
```

```
    A1 => mux2,
```

```
    A2 => mux3,
```

```
    A3 => mux4,
```

```
    S0 => S2,
```

```
    S1 => S3,
```

```
    Y => Y
```

```
);
```

```
end Behavioral;
```

Simulazione

Sono stati effettuati **6 test bench** con l'obiettivo di verificare il corretto funzionamento del **multiplexer indirizzabile 16:1**, composto internamente da **quattro multiplexer 4:1**. L'obiettivo dei test è assicurarsi che, tramite le opportune combinazioni delle linee di selezione (**S0, S1, S2, S3**), sia possibile attivare correttamente ciascuno dei quattro multiplexer interni e instradare l'ingresso selezionato sull'uscita **Y**.

- Nel primo test bench è stata scelta la seguente configurazione $I(0) \leq '1'; S0 \leq '0'; S1 \leq '0'; S2 \leq '0'; S3 \leq '0'$; Viene correttamente selezionato il **MUX_4_1_1**. L'ingresso 0 è attivo e selezionato.
- Nel secondo è stata scelta la seguente configurazione: $I(5) \leq '1'; S0 \leq '1'; S1 \leq '0'; S2 \leq '0'; S3 \leq '1'$; selezionato il **MUX_4_1_2**, ma con una **combinazione errata** delle linee di selezione. L'ingresso attivo non è correttamente selezionato.
- Nel terzo è stata scelta la seguente configurazione: $I(5) \leq '1'; S0 \leq '0'; S1 \leq '1'; S2 \leq '0'; S3 \leq '1'$; Stesso ingresso del test precedente, ma **selezione corretta**. L'ingresso 5 è instradato correttamente.
- Nel quarto è stata scelta la seguente configurazione: $I(10) \leq '1'; S0 \leq '0'; S1 \leq '1'; S2 \leq '0'; S3 \leq '1'$; L'ingresso attivo è il 10, ma la selezione corrisponde all'ingresso 5. L'ingresso non è quindi selezionato.
- Nel quinto è stata scelta la seguente configurazione: $I(10) \leq '1'; S0 \leq '1'; S1 \leq '0'; S2 \leq '1'; S3 \leq '0'$; Ora la selezione corrisponde correttamente all'ingresso attivo.
- Nel sesto è stata scelta la seguente configurazione: $I(15) \leq '1'; S0 \leq '1'; S1 \leq '1'; S2 \leq '1'; S3 \leq '1'$; È selezionato correttamente il **MUX_4_1_4** e l'ingresso 15. L'uscita risulta **alta**.

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.ALL;
```

```
USE ieee.std_logic_arith.ALL;
```

```
USE ieee.std_logic_unsigned.ALL;
```

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

```

ENTITY mux16_1_tb IS
END mux16_1_tb;

ARCHITECTURE behavioral OF mux16_1_tb IS
COMPONENT Mux16_1
PORT (
  I : IN std_logic_vector(0 TO 15);
  S0 : IN std_logic;
  S1 : IN std_logic;
  S2 : IN std_logic;
  S3 : IN std_logic;
  Y : OUT std_logic
);
END COMPONENT;

SIGNAL I : std_logic_vector(0 TO 15) := (OTHERS => '0');
SIGNAL S0, S1, S2, S3 : std_logic := '0';
SIGNAL Y : std_logic;

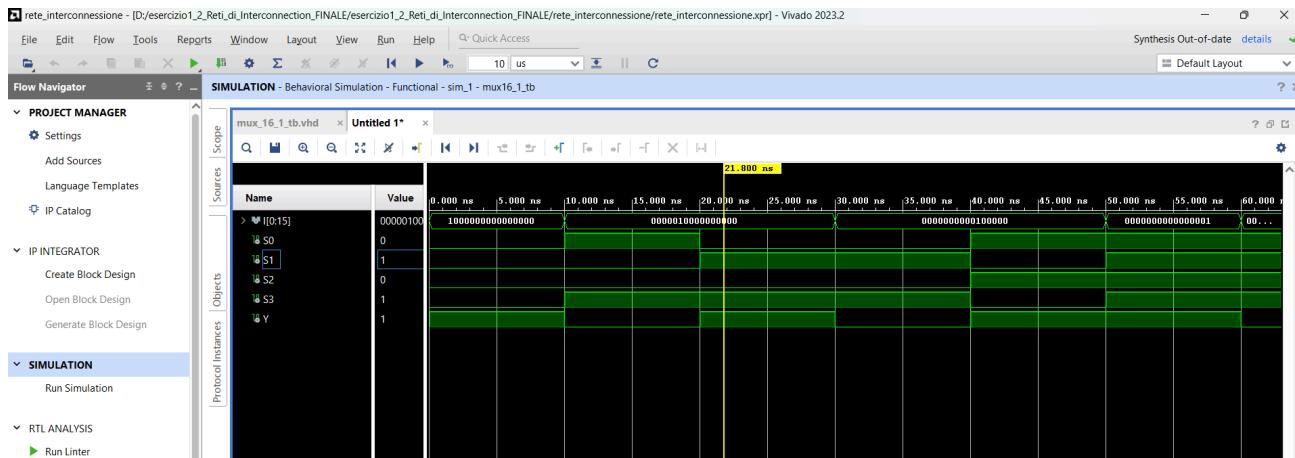
BEGIN
  UUT: Mux16_1 PORT MAP (I, S0, S1, S2, S3, Y);

PROCESS
BEGIN
  -- Test case 1: Select input 0
  I(0) <= '1'; S0 <= '0'; S1 <= '0'; S2 <= '0'; S3 <= '0';
  WAIT FOR 10 ns;
  I(0) <= '0';

  -- Test case 2: Select input 5
  I(5) <= '1'; S0 <= '1'; S1 <= '0'; S2 <= '0'; S3 <= '1';

```

```
WAIT FOR 10 ns;  
I(5) <= '0';  
  
-- Test case 3: Select input 5  
I(5) <= '1'; S0 <= '0'; S1 <= '1'; S2 <= '0'; S3 <= '1';  
WAIT FOR 10 ns;  
I(5) <= '0';  
  
-- Test case 4: Select input 10  
I(10) <= '1'; S0 <= '0'; S1 <= '1'; S2 <= '0'; S3 <= '1';  
WAIT FOR 10 ns;  
I(10) <= '0';  
-- Test case 5: Select input 10  
I(10) <= '1'; S0 <= '1'; S1 <= '0'; S2 <= '1'; S3 <= '0';  
WAIT FOR 10 ns;  
I(10) <= '0';  
  
-- Test case 6: Select input 15  
I(15) <= '1'; S0 <= '1'; S1 <= '1'; S2 <= '1'; S3 <= '1';  
WAIT FOR 10 ns;  
I(15) <= '0';  
  
WAIT;  
END PROCESS;  
END behavioral;
```



Esercizio 1.2: Rete di interconnessione a 16 sorgenti e 4 destinazioni

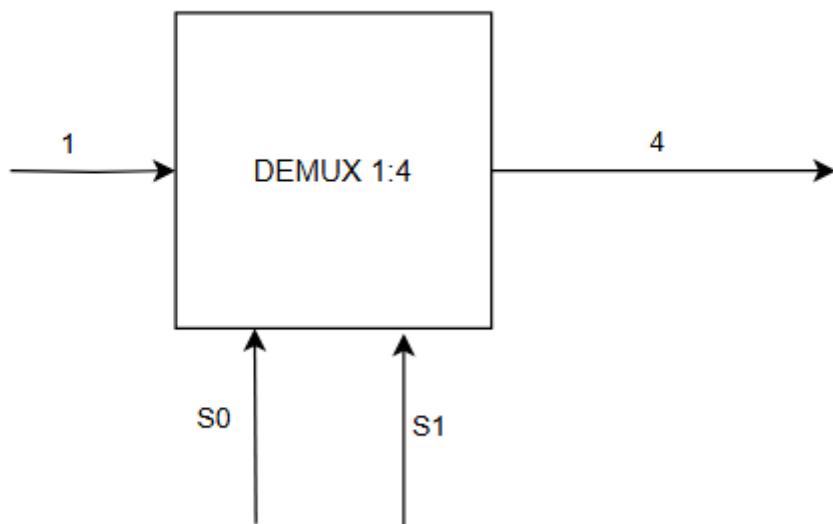
Utilizzando il componente sviluppato al punto precedente, progettare, implementare in VHDL e testare mediante simulazione una **rete di interconnessione a 16 sorgenti e 4 destinazioni**.

Progetto e architettura

Il progetto è stato realizzato partendo dall'esercizio precedente, in cui è stato sviluppato un MUX 16:1, con l'aggiunta di un demultiplexer 1:4.

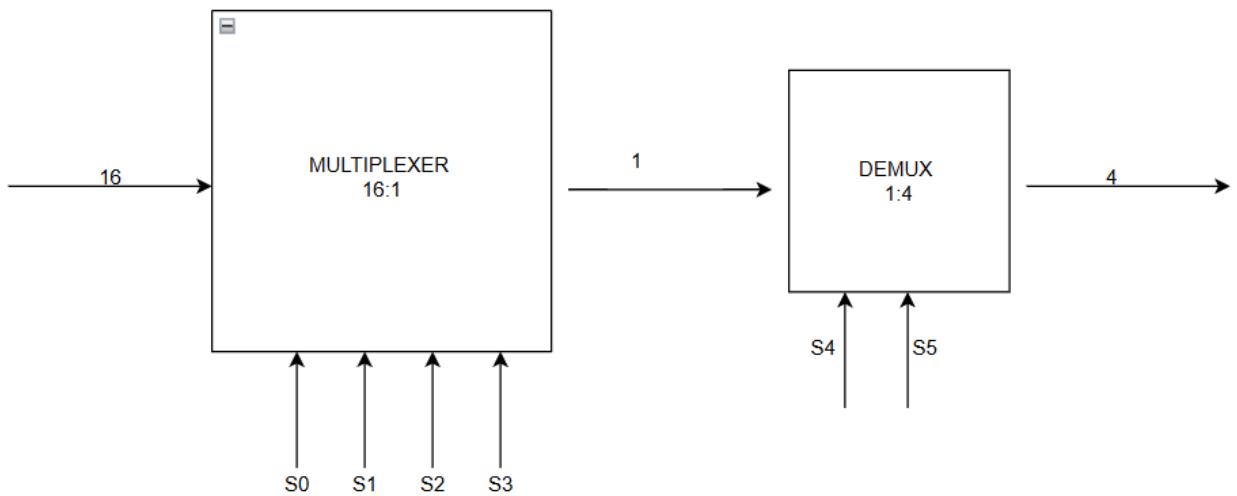
Il compito del DEMUX è quello di instradare un singolo segnale di ingresso su una delle 2^N linee di uscita, tramite N linee di selezione.

Il DEMUX è stato realizzato con un approccio Behavioral.



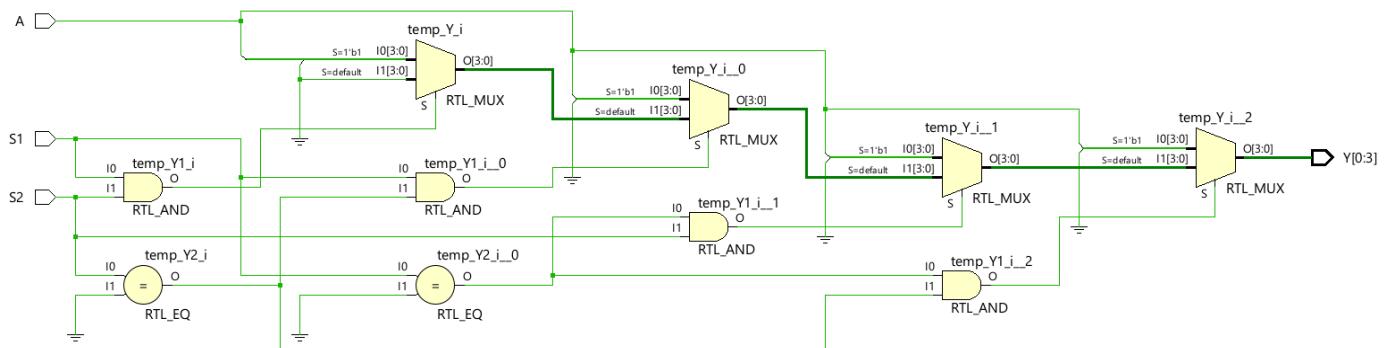
Grazie alla sua combinazione con il **multiplexer 16:1**, è stato possibile creare una rete di interconnessione capace di trasmettere i bit selezionati dai multiplexer agli ingressi del demultiplexer.

Unendo questi due componenti, il sistema è in grado di **instradare 16 sorgenti su 4 destinazioni**. Il funzionamento prevede che il **multiplexer selezioni uno degli ingressi**, mentre il demultiplexer smisti l'uscita del multiplexer verso una delle 4 destinazioni in base alle proprie linee di selezione.

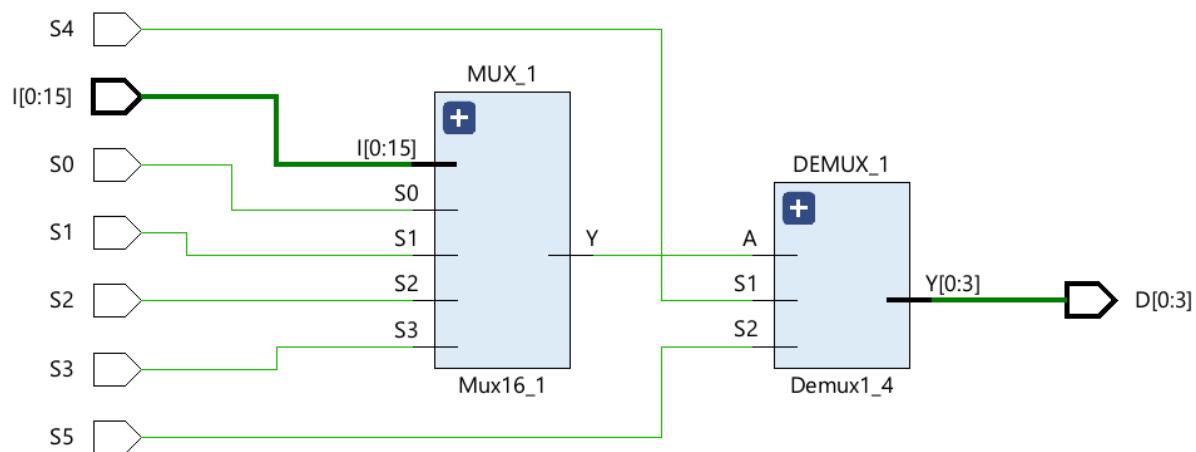


Riportiamo di seguito gli schematici dei due componenti.

Demux 1:4 :



16 sorgenti e 4 destinazioni :



Implementazione

L'implementazione del progetto è stata fatta in maniera strutturale, utilizzando i componenti appena creati. Sono presenti 26 segnali, 16 segnali di ingresso(quelli di ingresso del multiplexer), 6 segnali di selezione, 4 per i multiplexer e due per il demux, infine sono presenti i 4 segnali di uscita.

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

entity Sorgenti_Destinazioni is
    Port (
        I : in STD_LOGIC_VECTOR (0 to 15); -- 16 sorgenti
        S0 : in STD_LOGIC; -- Selezione per MUX
        S1 : in STD_LOGIC;
        S2 : in STD_LOGIC;
        S3 : in STD_LOGIC;
        --
        S4 : in STD_LOGIC;
        S5: in STD_LOGIC;
        D: out STD_LOGIC_VECTOR (0 to 3) -- 4 destinazioni
    );
end Sorgenti_Destinazioni;
```

```
architecture Behavioral of Sorgenti_Destinazioni is
```

```
-- Componenti
component MUX16_1 is
    Port (
        I : in STD_LOGIC_VECTOR (0 to 15); -- 16 ingressi
        S0 : in STD_LOGIC;
        S1 : in STD_LOGIC;
        S2 : in STD_LOGIC;
        S3 : in STD_LOGIC;
        Y : out STD_LOGIC
    );
end component;
```

```
end component;
```

```
component Demux1_4 is
```

```
Port (
```

```
    A : in STD_LOGIC;  
    Y : out STD_LOGIC_VECTOR (0 to 3);  
    S1 : in STD_LOGIC;  
    S2 : in STD_LOGIC
```

```
);
```

```
end component;
```

```
-- Segnale intermedio
```

```
signal mux_out : STD_LOGIC:='0';
```

```
begin
```

```
-- Istanza del MUX16_1 per selezionare una delle 16 sorgenti
```

```
MUX_1 : MUX16_1
```

```
port map (
```

```
    I => I,      -- Collegamento delle 16 sorgenti  
    S0 => S0,    -- Selezione S0  
    S1 => S1,    -- Selezione S1  
    S2 => S2,    -- Selezione S2  
    S3 => S3,    -- Selezione S3
```

```
    Y => mux_out -- Uscita del MUX
```

```
);
```

```
-- Istanza del DEMUX1_4 per distribuire l'uscita a 4 destinazioni
```

```
DEMUX_1 : Demux1_4
```

```
port map (
```

```
    A => mux_out, -- Uscita dal MUX
```

```
    Y => D, -- Collegamento alle 4 destinazioni
```

```

S1 => S4,      -- Selezione per DEMUX
S2 => S5      -- Selezione per DEMUX
);

end Behavioral;

```

Simulazione

Per assicurarsi che sia tutto funzionante sono stati testati i singoli componenti.
Il multiplexer è stato testando verificando che con le esatte combinazioni di selezione e di ingresso si attivassero i multiplexer desiderati.

LIBRARY ieee;

```

USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
use IEEE.STD_LOGIC_1164.ALL;

```

```

ENTITY mux16_1_tb IS
END mux16_1_tb;

```

```
ARCHITECTURE behavior OF mux16_1_tb IS
```

```
COMPONENT Mux16_1
```

```

PORT (
    I : IN std_logic_vector(0 TO 15);
    S0 : IN std_logic;
    S1 : IN std_logic;
    S2 : IN std_logic;
    S3 : IN std_logic;
    Y : OUT std_logic
);
```

```
END COMPONENT;
```

```
SIGNAL I : std_logic_vector(0 TO 15) := (OTHERS => '0');
```

```
SIGNAL S0, S1, S2, S3 : std_logic := '0';
SIGNAL Y : std_logic;

BEGIN
  UUT: Mux16_1 PORT MAP (I, S0, S1, S2, S3, Y);
```

PROCESS

BEGIN

-- Test case 1: Select input 0

```
I(0) <= '1'; S0 <= '0'; S1 <= '0'; S2 <= '0'; S3 <= '0';
WAIT FOR 10 ns;
I(0) <= '0';
```

-- Test case 2: Select input 5

```
I(5) <= '1'; S0 <= '1'; S1 <= '0'; S2 <= '0'; S3 <= '1';
WAIT FOR 10 ns;
I(5) <= '0';
```

-- Test case 2: Select input 5

```
I(5) <= '1'; S0 <= '0'; S1 <= '1'; S2 <= '0'; S3 <= '1';
WAIT FOR 10 ns;
I(5) <= '0';
```

-- Test case 3: Select input 10

```
I(10) <= '1'; S0 <= '0'; S1 <= '1'; S2 <= '0'; S3 <= '1';
WAIT FOR 10 ns;
I(10) <= '0';
```

-- Test case 3: Select input 10

```
I(10) <= '1'; S0 <= '1'; S1 <= '0'; S2 <= '1'; S3 <= '0';
WAIT FOR 10 ns;
I(10) <= '0';
```

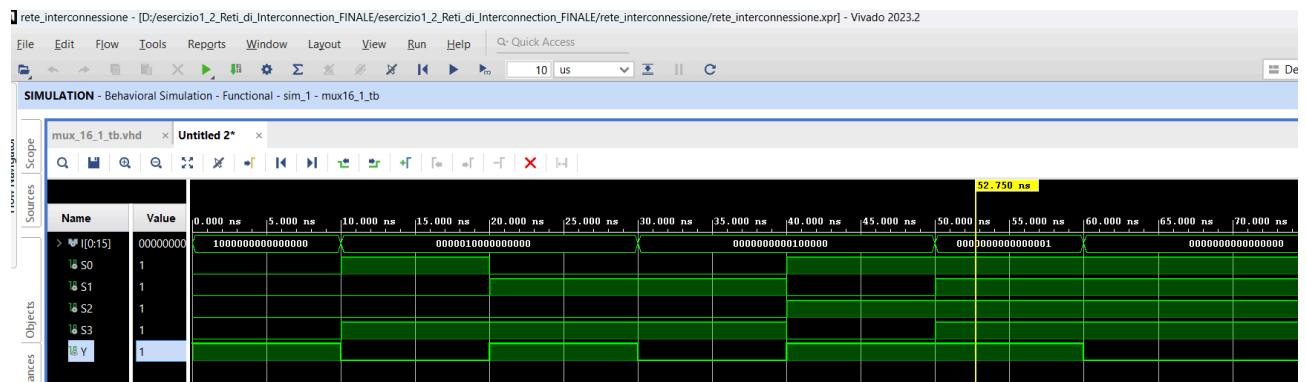
```
-- Test case 4: Select input 15
I(15) <= '1'; S0 <= '1'; S1 <= '1'; S2 <= '1'; S3 <= '1';
WAIT FOR 10 ns;
I(15) <= '0';

WAIT;
END PROCESS;
```

Abbiamo controllato che :

- TEST CASE 1 : I(0) <= '1'; S0 <= '0'; S1 <= '0'; S2 <= '0'; S3 <= '0'; RISULTA funzionante, con l'uscita alta grazie al fatto che è stato selezionato il MUX_4_1 con S0S1= '00' e con S2S3 = '00'.
- TEST CASE 2 : I(5) <= '1'; S0 <= '1'; S1 <= '0'; S2 <= '0'; S3 <= '1'; Risulta una configurazione esatta, con l'uscita a zero, in quanto l'ingresso prevede input I(5) quindi il secondo multiplexer ma è presente una configurazione errata dell'ingresso di selezione.
- TEST CASE 3: I(5) <= '1'; S0 <= '0'; S1 <= '1'; S2 <= '0'; S3 <= '1' Risulta funzionante con ingresso di uscita alto, avendo inserito la combinazione esatta di ingressi di selezione.
- TEST CASE 4: I(10) <= '1'; S0 <= '0'; S1 <= '1'; S2 <= '0'; S3 <= '1'. Risulta funzionante in quanto l'uscita è bassa perchè è stata scelta la combinazione esatta di ingressi di selezione.
- TEST CASE 5: I(10) <= '1'; S0 <= '1'; S1 <= '0'; S2 <= '1'; S3 <= '0': Risulta funzionante.
- TEST CASE 6: I(15) <= '1'; S0 <= '1'; S1 <= '1'; S2 <= '1'; S3 <= '1'. Risulta funzionante.

Abbiamo dato in ingresso i segnali minimi affinché possano attivarsi tutti i multiplexer.



DEMUX_1_4

Il test bench per il demultiplexer è stato creato in modo tale da assicurarci che in base all'ingresso di selezione otteniamo l'uscita desiderata.

- TEST CASE 1 : A <= '1'; S1 <= '0'; S2 <= '0'. Otteniamo l'uscita Y= '1000'
- TEST CASE 2 : A <= '1'; S1 <= '0'; S2 <= '1'. Otteniamo l'uscita Y='0100'
- TEST CASE 3: A <= '1'; S1 <= '1'; S2 <= '0'. Otteniamo l'uscita Y='0010'
- TEST CASE 4: A <= '1'; S1 <= '1'; S2 <= '1'. Otteniamo l'uscita Y='0001'

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Demux1_4_tb IS
END Demux1_4_tb;

ARCHITECTURE behavioral OF Demux1_4_tb IS
COMPONENT Demux1_4
PORT(
A : IN std_logic;
Y : OUT std_logic_vector(0 TO 3);
S1 : IN std_logic;
S2 : IN std_logic
);
END COMPONENT;

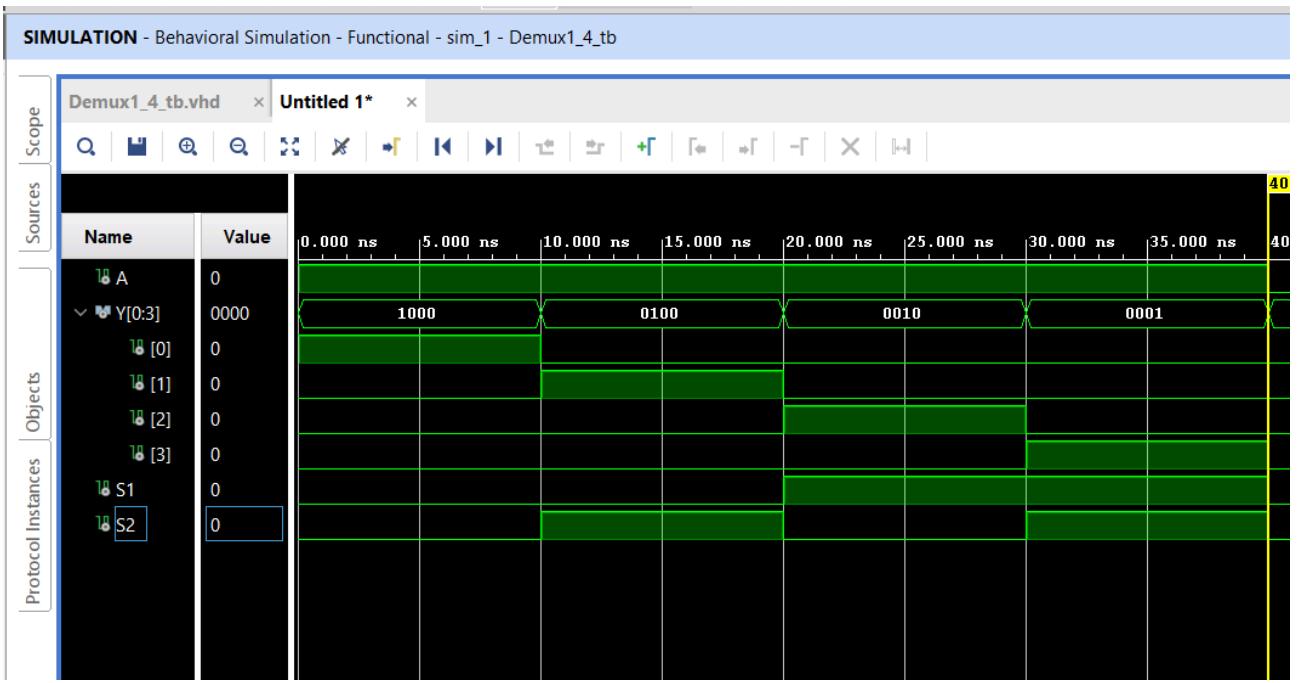
SIGNAL A : std_logic := '0';
SIGNAL Y : std_logic_vector(0 TO 3);
SIGNAL S1 : std_logic := '0';
SIGNAL S2 : std_logic := '0';

BEGIN
uut: Demux1_4 PORT MAP (
A => A,
Y => Y,
S1 => S1,
S2 => S2
);

stim_proc: PROCESS
BEGIN

```

```
-- Test case 1: A = '1', S1 = '0', S2 = '0' : Y = '1000'  
A <= '1'; S1 <= '0'; S2 <= '0';  
WAIT FOR 10 ns;  
  
-- Test case 2: A = '1', S1 = '0', S2 = '1' : Y ='0100'  
A <= '1'; S1 <= '0'; S2 <= '1';  
WAIT FOR 10 ns;  
  
-- Test case 3: A = '1', S1 = '1', S2 = '0' : Y ='0010'  
A <= '1'; S1 <= '1'; S2 <= '0';  
WAIT FOR 10 ns;  
  
-- Test case 4: A = '1', S1 = '1', S2 = '1' : Y ='0001'  
A <= '1'; S1 <= '1'; S2 <= '1';  
WAIT FOR 10 ns;  
  
-- Test case 5: A = '0', S1 = '0', S2 = '0' : Y ='0000'  
A <= '0'; S1 <= '0'; S2 <= '0';  
WAIT FOR 10 ns;  
  
-- Test case 6: A = '0', S1 = '1', S2 = '1' : Y ='0100'  
A <= '0'; S1 <= '1'; S2 <= '1';  
WAIT FOR 10 ns;  
  
WAIT;  
END PROCESS;  
END behavioral;
```



Essendoci assicurati che ogni componente funzioni correttamente possiamo proseguire col testbench del progetto principale.

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

```
entity tb_Sorgenti_Destinazioni is
end tb_Sorgenti_Destinazioni;
```

```
architecture behavior of tb_Sorgenti_Destinazioni is
```

```
-- Componenti del DUT
component Sorgenti_Destinazioni is
    Port (
        I : in STD_LOGIC_VECTOR (0 to 15); -- 16 sorgenti
        S0 : in STD_LOGIC; -- Selezione per MUX
        S1 : in STD_LOGIC;
        S2 : in STD_LOGIC;
        S3 : in STD_LOGIC;
        S4 : in STD_LOGIC;
        S5 : in STD_LOGIC;
```

```

D: out STD_LOGIC_VECTOR (0 to 3) -- 4 destinazioni
);

end component;

-- Segnali per il testbench
signal I : STD_LOGIC_VECTOR (0 to 15);
signal S0, S1, S2, S3, S4 ,S5: STD_LOGIC;
signal D : STD_LOGIC_VECTOR (0 to 3);

begin

-- Istanza del DUT (Device Under Test)
uut: Sorgenti_Destinazioni
port map (
    I => I,
    S0 => S0,
    S1 => S1,
    S2 => S2,
    S3 => S3,
    S4 => S4,
    S5 => S5,
    D => D
);

-- Stimolo (Generazione delle vettoriali e delle selezioni)
process
begin
wait for 100 ns;
-- Inizializzazione delle sorgenti (I) con valori distinti per testare le selezioni
I <= "1110000000000000";
S0 <= '0';
S1 <= '0';

```

```
S2 <= '0';
S3 <= '0';
S4 <= '0';
S5 <= '0';

-- Terminazione del processo di simulazione
```

```
wait for 20ns;

I <= "0000111000000000";
S0 <= '0';
S1 <= '1';
S2 <= '0';
S3 <= '1';
S4 <= '0';
S5 <= '1';

-- Terminazione del processo di simulazione
```

```
wait for 20ns;

I <= "0000000011100000";
S0 <= '1';
S1 <= '0';
S2 <= '1';
S3 <= '0';
S4 <= '1';
S5 <= '0';

-- Terminazione del processo di simulazione
```

```
wait for 20ns;

I <= "0000000000000111";
S0 <= '1';
S1 <= '1';
S2 <= '1';
S3 <= '1';
```

```

S4 <= '1';
S5 <= '1';
-- Terminazione del processo di simulazione

```

```

wait for 20ns;

I <= "1110000000000000";
S0 <= '0';
S1 <= '0';
S2 <= '0';
S3 <= '0';
S4 <= '0';
S5 <= '0';
-- Terminazione del processo di simulazione

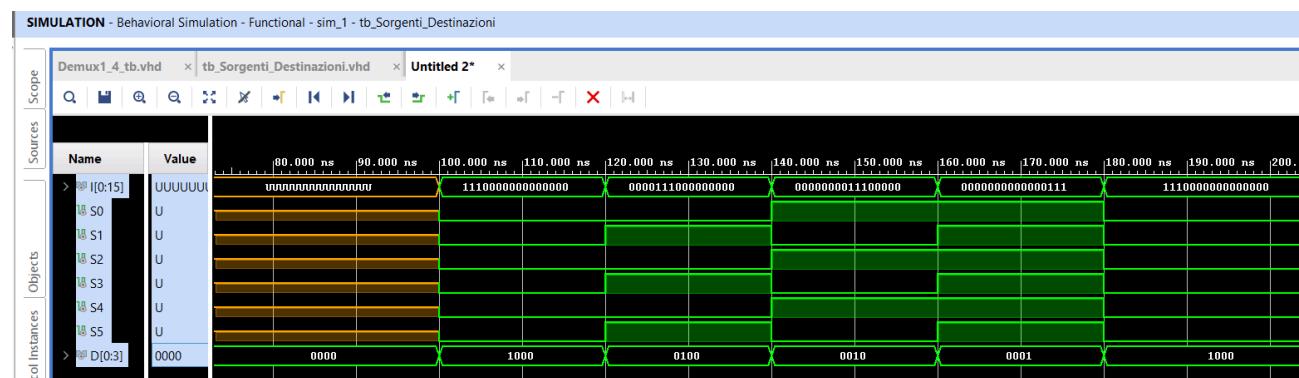
```

```

wait for 20ns;
wait;
end process;

end behavioral;

```



Esercizio 1.3: Implementazione della rete di interconnessione su board

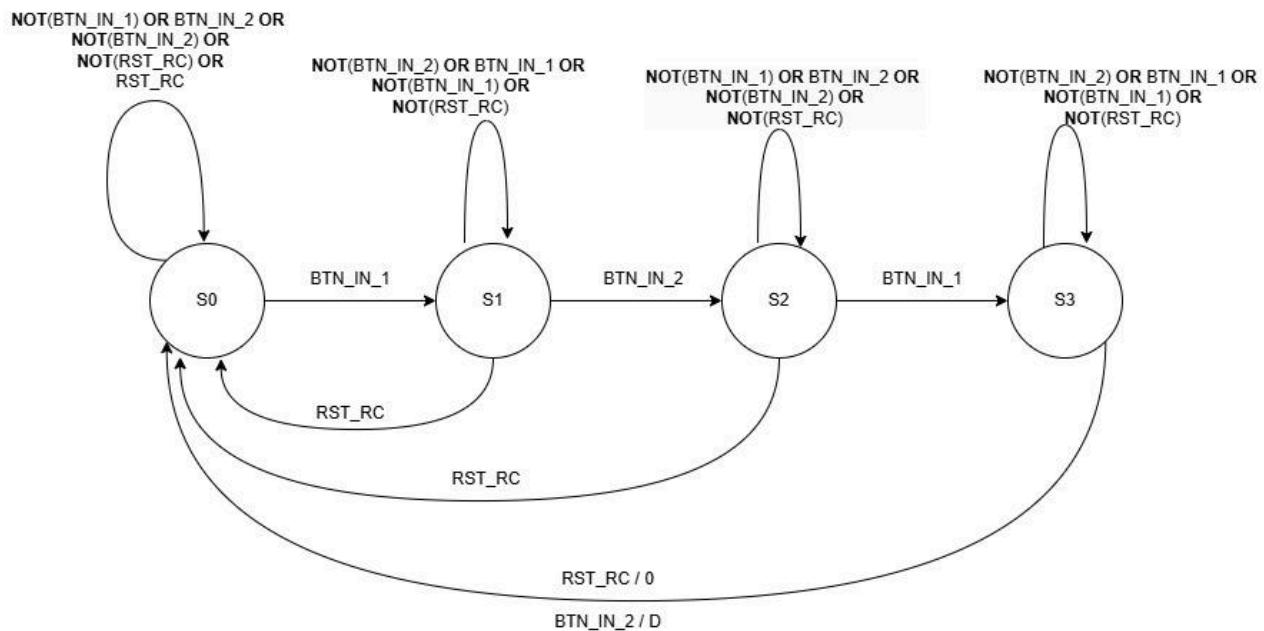
Sintetizzare ed implementare su board il progetto della rete di interconnessione sviluppato al punto 1.2, utilizzando gli switch per fornire gli input di selezione e i led per visualizzare i 4 bit di uscita. Per quanto riguarda i 16 bit dato in input, essi devono essere immessi mediante switch, 8 bit alla volta, sviluppando un'apposita “rete di controllo” per l’acquisizione che utilizzi due bottoni della board per caricare rispettivamente la prima e la seconda metà del dato in ingresso.

Sintesi su board di sviluppo

Per fare ciò è stata realizzata una rete di controllo avente i seguenti input :

- Clock
- Reset
- Bottone 1
- Bottone 2
- Switch
- Out_led

Di seguito è riportato l’automa a stati finiti della rete di controllo da noi realizzata



Il funzionamento è il seguente:

- In S0 (stato iniziale) l’utente può inserire i 4 bit di selezione del mux attraverso la pressione del tasto **BTN_IN_1** (BTNL sulla board)
- In S1 l’utente può inserire i 2 bit di selezione del demux attraverso la pressione del tasto **BTN_IN_2** (BTNR sulla board)
- In S2 vengono acquisiti gli 8 bit più significativi (msb) che passano dal mux al demux

- In S3 vengono acquisiti gli 8 bit meno significativi (lsb) che passano dal mux al demux e, successivamente, alla pressione del tasto BTN_IN_2 (BTNR sulla board), il sistema torna nello stato S0. Al termine di ciò l'uscita verrà visualizzata sui led della board.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Sorgenti_Destinazioni_board is
Port ( CLK : in STD_LOGIC;
        RST : in STD_LOGIC;
        BTN_1 : in STD_LOGIC;
        BTN_2 : in STD_LOGIC;
        IN_SWITCH : in STD_LOGIC_VECTOR(0 TO 7);
        OUT_LED : out STD_LOGIC_VECTOR(0 TO 3)
        );

```

```
end Sorgenti_Destinazioni_board;
```

```
architecture Structural of Sorgenti_Destinazioni_board is
signal I : STD_LOGIC_VECTOR(15 DOWNTO 0) := (others => '0');
signal S0 : STD_LOGIC := '0';
signal S1 : STD_LOGIC := '0';
signal S2 : STD_LOGIC := '0';
signal S3 : STD_LOGIC := '0';
signal S4 : STD_LOGIC := '0';
signal S5 : STD_LOGIC := '0';
signal S6 : STD_LOGIC := '0';
signal btn_1_out_clean : STD_LOGIC := '0';
signal btn_2_out_clean : STD_LOGIC := '0';
signal btn_RST_out_clean : STD_LOGIC := '0';
```

```
component Sorgenti_Destinazioni is
```

```
Port (
    I : in STD_LOGIC_VECTOR (0 to 15); -- 16 sorgenti
    S0 : in STD_LOGIC; -- Selezione per MUX
    S1 : in STD_LOGIC;
    S2 : in STD_LOGIC;
    S3 : in STD_LOGIC;
    S4 : in STD_LOGIC;
    S5: in STD_LOGIC;
    D: out STD_LOGIC_VECTOR (0 to 3) -- 4 destinazioni
);
end component;
```

```
component rete_di_controllo
port(
    CLK_RC : in STD_LOGIC;
    RST_RC : in STD_LOGIC;
    SWITCH_IN: in STD_LOGIC_VECTOR(0 to 7);
    BTN_IN_1: in STD_LOGIC;
    BTN_IN_2: in STD_LOGIC;
    b : out STD_LOGIC_VECTOR (0 to 15);
    S0 : out STD_LOGIC; -- Selezione per MUX
    S1 : out STD_LOGIC;
    S2 : out STD_LOGIC;
    S3 : out STD_LOGIC;
    --
    S4 : out STD_LOGIC; -- selezione demux
    S5: out STD_LOGIC;
);
end component;
```

```
component ButtonDebouncer
generic(
```

```
CLK_period : integer := 10; -- periodo del clock (in ns)
btn_noise_time : integer := 10000000 --durata dell'oscillazione (in ns)
);
port(
    CLK : in STD_LOGIC;
    RST : in STD_LOGIC;
    BTN : in STD_LOGIC;
    CLEARED_BTN : out STD_LOGIC
);
end component;
```

```
begin
```

```
SD_16_4 : Sorgenti_Destinazioni
```

```
Port map(
```

```
    I => I,
    S0 => S0,
    S1 => S1,
    S2 => S2,
    S3 => S3,
    --
    S4 => S4,
    S5 => S5,
    D => OUT_LED
);
```

```
RDC : rete_di_controllo
```

```
Port map(
```

```
    CLK_RC => CLK,
    RST_RC => RST,
    SWITCH_IN => IN_SWITCH,
    BTN_IN_1 => BTN_1,
```

```
BTN_IN_2 => BTN_2,
```

```
b => I,
```

```
S0 => S0,
```

```
S1 => S1,
```

```
S2 => S2,
```

```
S3 => S3,
```

```
-- selezioni del DEMUX
```

```
S4 => S4,
```

```
S5 => S5
```

```
);
```

```
BTN_DEB_1 : ButtonDebouncer
```

```
Port map(
```

```
    RST => btn_rst_out_clean,
```

```
    CLK => CLK,
```

```
    BTN => BTN_1,
```

```
    CLEARED_BTN => btn_1_out_clean
```

```
);
```

```
BTN_DEB_2 : ButtonDebouncer
```

```
Port map(
```

```
    CLK => CLK,
```

```
    RST => btn_rst_out_clean,
```

```
    BTN => BTN_2,
```

```
    CLEARED_BTN => btn_2_out_clean
```

```
);
```

```
BTN_DEB_RST : ButtonDebouncer
```

```
Port map(
```

```
    CLK => CLK,
```

```
    RST => '0',
```

```
    BTN => RST,
```

```
CLEARED_BTN => btn_RST_out_clean  
);
```

end Structural;

MAPPING :

Sono stati scelti i primi 8 switch come segnali di ingresso, in base allo stato in cui si trova la macchina hanno uno scopo diverso (i bit più significativi vanno da sinistra verso destra).

Inoltre i primi 4 led come uscita (H17, K15 , J13, N14), H17 indica il bit meno significativo mentre N14 il bit più significativo.

Il bottone uno che funge da selezione dei multiplixer e inserimento dei segnali è mappato con BUTTON P17,
il bottone due che funge da selezione per i demux sono mappati con BUTTON M17,
il bottone di reset che funge da reset è mappato con BUTTON N17

##Switches

```
set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports { IN_SWITCH[0] }];  
#IO_L24N_T3_RS0_15 Sch=sw[0]  
  
set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports { IN_SWITCH[1] }];  
#IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]  
  
set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 } [get_ports { IN_SWITCH[2] }];  
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]  
  
set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 } [get_ports { IN_SWITCH[3] }];  
#IO_L13N_T2_MRCC_14 Sch=sw[3]  
  
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 } [get_ports { IN_SWITCH[4] }];  
#IO_L12N_T1_MRCC_14 Sch=sw[4]  
  
set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMOS33 } [get_ports { IN_SWITCH[5] }];  
#IO_L7N_T1_D10_14 Sch=sw[5]  
  
set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMOS33 } [get_ports { IN_SWITCH[6] }];  
#IO_L17N_T2_A13_D29_14 Sch=sw[6]  
  
set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMOS33 } [get_ports { IN_SWITCH[7] }];  
#IO_L5N_T0_D07_14 Sch=sw[7]
```

LEDs

```
set_property -dict { PACKAGE_PIN H17      IOSTANDARD LVCMOS33 } [get_ports { OUT_LED[0] }];  
#IO_L18P_T2_A24_15 Sch=led[0]  
  
set_property -dict { PACKAGE_PIN K15      IOSTANDARD LVCMOS33 } [get_ports { OUT_LED[1] }];  
#IO_L24P_T3_RS1_15 Sch=led[1]  
  
set_property -dict { PACKAGE_PIN J13      IOSTANDARD LVCMOS33 } [get_ports { OUT_LED[2] }];  
#IO_L17N_T2_A25_15 Sch=led[2]
```

```

set_property -dict { PACKAGE_PIN N14      IOSTANDARD LVCMOS33 } [get_ports { OUT_LED[3] }];
#IO_L8P_T1_D11_14 Sch=led[3]

##Buttons

set_property -dict { PACKAGE_PIN N17      IOSTANDARD LVCMOS33 } [get_ports { RST }]; #IO_L9P_T1_DQS_14
Sch=btnc

set_property -dict { PACKAGE_PIN P17      IOSTANDARD LVCMOS33 } [get_ports { BTN_1 }];
#IO_L12P_T1_MRCC_14 Sch=btnl

set_property -dict { PACKAGE_PIN M17      IOSTANDARD LVCMOS33 } [get_ports { BTN_2 }];
#IO_L10N_T1_D15_14 Sch=btnr

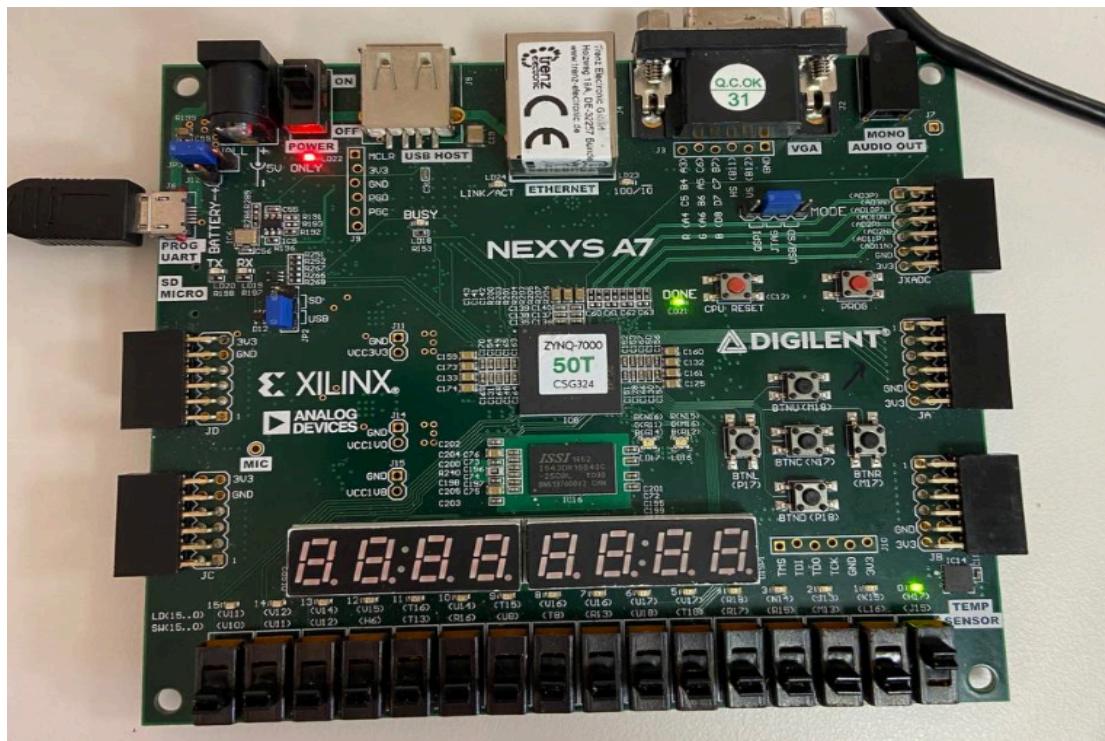
```

-TEST CASE 1 : è stata data la seguente configurazione in input:

MUX : 0000

DEMUX : 00

MULTIPLEXER : '0000-0000-0000-0001

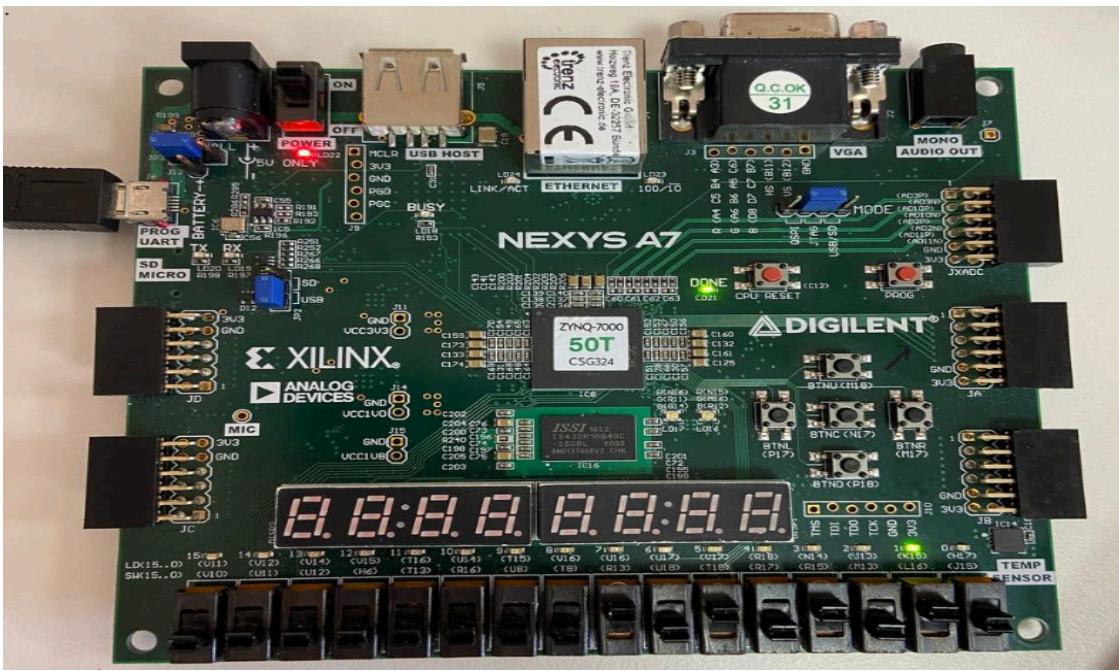


-TEST CASE 2 : è stata data la seguente configurazione in input:

MUX : 0101

DEMUX : 01

MULTIPLEXER : '0101-0101-1010-1010

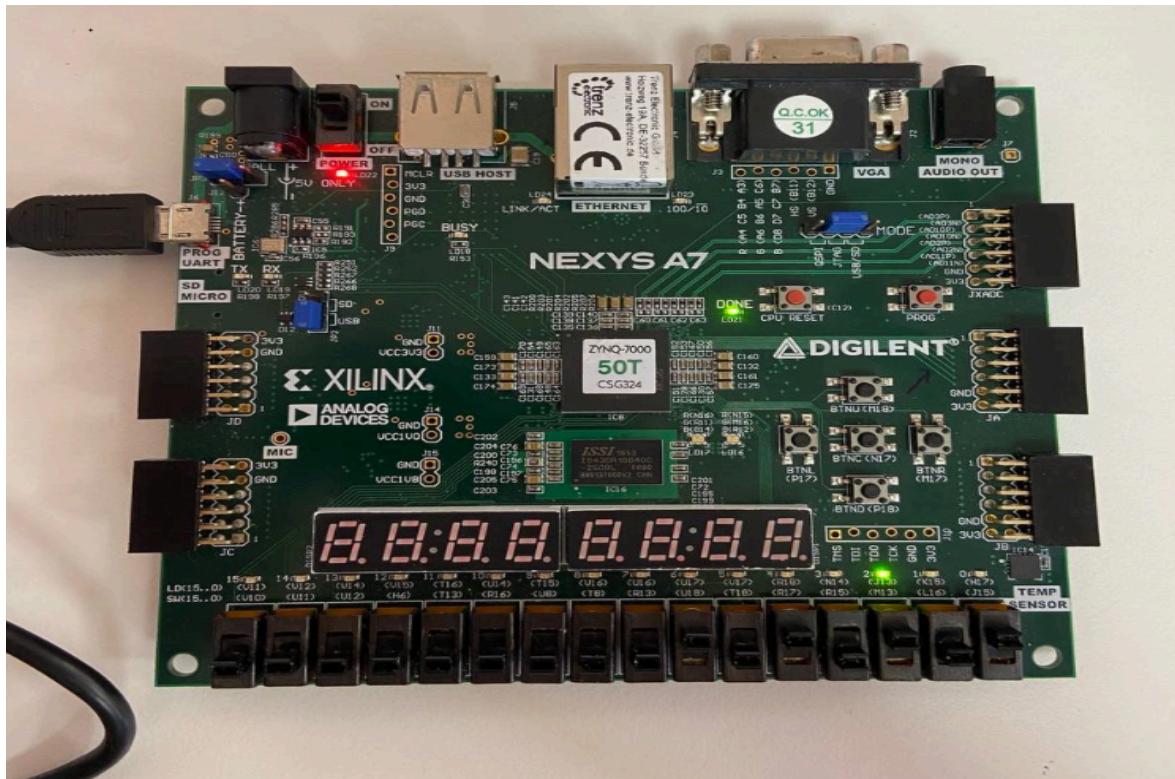


-TEST CASE 3 : è stata data la seguente configurazione in input:

MUX : 1010

DEMUX : 10

MULTIPLEXER : 1010-1010-1010-0101

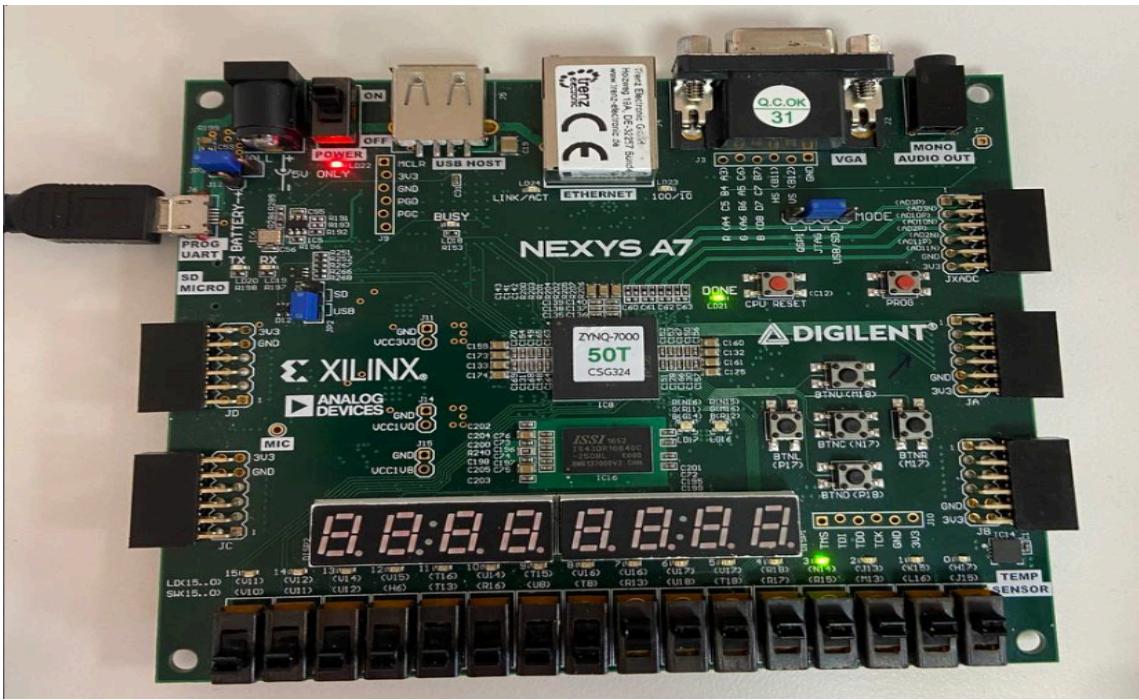


-TEST CASE 4 : è stata data la seguente configurazione in input:

MUX : 1111

DEMUX : 11

MULTIPLEXER : 1111-1111-1111-1111'



Esercizio 2 : Sistema ROM+M

Esercizio 2.1

Progettare, implementare in VHDL e testare mediante simulazione un sistema S composto da una ROM puramente combinatoria di 16 locazioni da 8 bit ciascuna e da una macchina combinatoria M che opera come segue: fornito al sistema un indirizzo A di 4 bit, il sistema restituisce il valore contenuto nella ROM all'indirizzo A opportunamente "trasformato" attraverso la macchina M. Il comportamento della macchina M è totalmente a scelta dello studente, l'unico vincolo è che essa prenda in ingresso 8 bit e ne fornisca in uscita 4.

Progetto e architettura

Il sistema è stato realizzato in maniera Strutturale unendo appositamente i due componenti ROM e Macchina M.

La macchina M prende in ingresso il bit memorizzato in memoria e restituisce i 4 bit più significativi.

Implementazione

Il sistema è stato implementato tramite una memoria ROM (vedi appendice) e una macchina combinatoria M.

Nella ROM sono stati precaricati 16 valori da 8 bit ciascuno dei valori :

```
constant ROM : rom_type := (
```

```
X"00",-- 0000-0000
```

```
X"11",-- 0001-0001
```

```
X"22",-- 0010-0010
```

```
X"33",-- 0011-0011
```

```
X"44",--0100-0100
```

```

X"55", -- 0101-0101
X"66", -- 0110-0110
X"77", -- 0111-0111
X"88", -- 1000-1000
X"99", -- 1001-1001
X"aa", -- 1010-1010
X"bb", -- 1011-1011
X"cc", -- 1100-1100
X"dd",-- 1101-1101
X"ee",-- 1110-1110
X"ff"--1111-1111
);

```

Macchina M

La macchina M prende in ingresso L'indirizzo memorizzato in memoria e restituisce in uscita i 4 bit più significativi del valore letto dalla ROM.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity macchina_M is
    Port ( DATA_IN : in STD_LOGIC_VECTOR (7 downto 0);
           DATA_OUT : out STD_LOGIC_VECTOR (3 downto 0));
end macchina_M;

```

architecture Behavioral of macchina_M is

```

begin
    DATA_OUT <= DATA_IN(7 downto 4);
end Behavioral;

```

SISTEMA FINALE

L'architettura scelta è strutturale, la macchina prende in ingresso 4 bit che invia alla rom. La rom in base a questi 4 bit restituisce un indirizzo(il dato di 8 bit) che andranno in ingresso alla macchina e restituisce le cifre significative

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity sistema_S is
  Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
         RST : in STD_LOGIC;
         OUTPUT : out STD_LOGIC_VECTOR (3 downto 0)
       );
end sistema_S;
```

```
architecture structural of sistema_S is
```

```
signal tmp : std_logic_vector(7 downto 0) := "00000000";
```

```
component rom_16_8 is
```

```
  Port ( ADDR : in STD_LOGIC_VECTOR (3 downto 0);
         DATA : out STD_LOGIC_VECTOR (7 downto 0);
         RST : in STD_LOGIC);
end component;
```

```
component macchina_M is
```

```
  Port ( DATA_IN : in STD_LOGIC_VECTOR (7 downto 0);
         DATA_OUT : out STD_LOGIC_VECTOR (3 downto 0));
end component;
```

```
begin
```

```
  rom : rom_16_8
```

```
    port map(
      ADDR => A,
      DATA => tmp,
      RST => RST
```

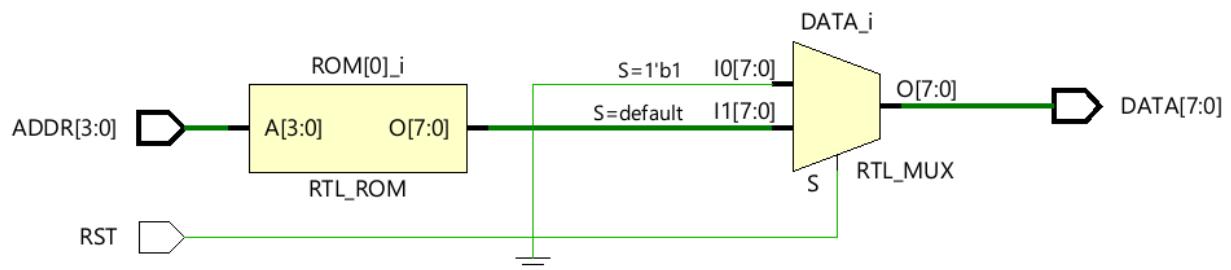
```

    );
m : macchina_M
port map(
    DATA_IN => tmp,
    DATA_OUT => OUTPUT
);

```

end structural;

Schematic :



Simulazione

I test bench sono stati effettuati in maniera tale che il sistema riceve in ingresso 4 valori e dà li saranno letti in memoria i valori presenti per poi far si che la macchina M possa operare.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

```

```

entity sistema_S_tb is
end sistema_S_tb;

```

```
architecture Behavioral of sistema_S_tb is
```

```

COMPONENT sistema_S
PORT(
    A : IN std_logic_vector(3 downto 0);

```

```

        RST : IN std_logic;
        OUTPUT : OUT std_logic_vector(3 downto 0)
    );
END COMPONENT;

-- Signals
SIGNAL A_tb : std_logic_vector(3 downto 0) := (others => '0');
SIGNAL RST_tb : std_logic := '0';
SIGNAL OUTPUT_tb : std_logic_vector(3 downto 0);

begin
    uut: sistema_S PORT MAP (
        A => A_tb,
        RST => RST_tb,
        OUTPUT => OUTPUT_tb
    );

    stim_proc: process
    begin
        -- Attendi 100ns prima di iniziare il test
        wait for 100 ns;

        -- Test 1: Indirizzo 1000
        A_tb <= "1000";
        wait for 15 ns;

        -- Test 2: Indirizzo 0000
        A_tb <= "0000";
        wait for 15 ns;

        -- Test 3: Indirizzo 0001

```

```
A_tb <= "0001";
```

```
wait for 15 ns;
```

```
-- Test 4: Indirizzo 1111
```

```
A_tb <= "1111";
```

```
wait for 15 ns;
```

```
-- Test 5: Indirizzo 0101
```

```
A_tb <= "0101";
```

```
wait for 15 ns;
```

```
-- Test Reset
```

```
RST_tb <= '1';
```

```
wait for 15 ns;
```

```
RST_tb <= '0';
```

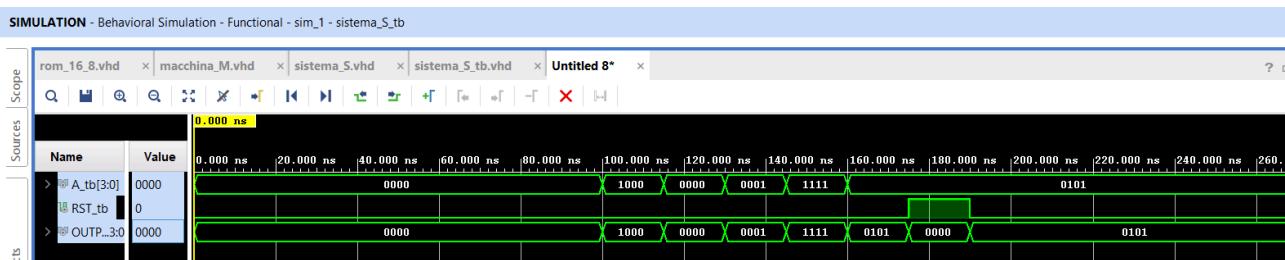
```
wait for 15 ns;
```

```
-- Fine test
```

```
wait;
```

```
end process;
```

```
end Behavioral;
```



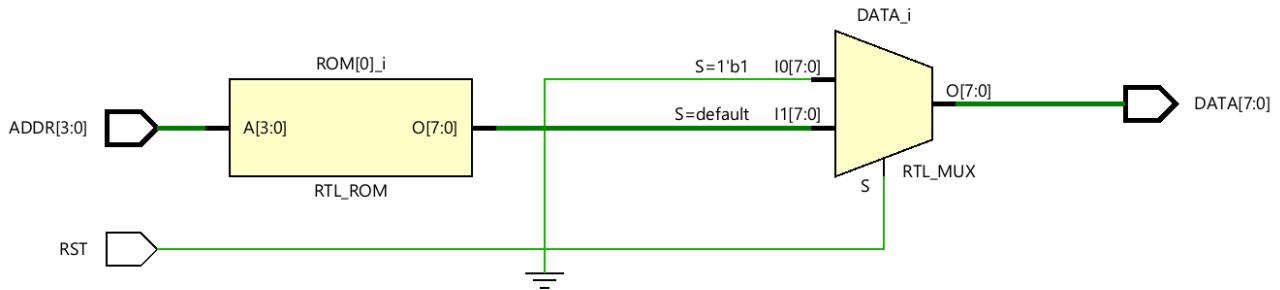
Esercizio 2.2: Implementazione sistema ROM+M su board

Sintetizzare ed implementare su board il progetto del sistema ROM+M sviluppato al punto 2.1, utilizzando gli switch per fornire l'indirizzo della ROM da cui leggere i valori da trasformare e i led per visualizzare i 4 bit di uscita.

Sintesi su board di sviluppo

Per l'implementazione su board della ROM non è stato necessario aggiungere alcun elemento rispetto al sistema dell'esercizio precedente.

SCHEMATIC (ROM + M + BOARD)



E' stato sufficiente modificare i file di constraint come segue:

- Sono stati utilizzati gli switch "a sinistra"(da V10 a H6)

```

set_property -dict { PACKAGE_PIN H6  IOSTANDARD LVCMOS33 } [get_ports { A[0] }];
#IO_L24P_T3_35 Sch=sw[12]

set_property -dict { PACKAGE_PIN U12  IOSTANDARD LVCMOS33 } [get_ports { A[1] }];
#IO_L20P_T3_A08_D24_14 Sch=sw[13]

set_property -dict { PACKAGE_PIN U11  IOSTANDARD LVCMOS33 } [get_ports { A[2] }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]

set_property -dict { PACKAGE_PIN V10  IOSTANDARD LVCMOS33 } [get_ports { A[3] }];
#IO_L21P_T3_DQS_14 Sch=sw[15]

```

- I led di uscita sono gli stessi dell'esercizio precedente :

```

## LEDs

set_property -dict { PACKAGE_PIN H17  IOSTANDARD LVCMOS33 } [get_ports { OUTPUT[0] }];
#IO_L18P_T2_A24_15 Sch=led[0]

set_property -dict { PACKAGE_PIN K15  IOSTANDARD LVCMOS33 } [get_ports { OUTPUT[1] }];
#IO_L24P_T3_RS1_15 Sch=led[1]

set_property -dict { PACKAGE_PIN J13  IOSTANDARD LVCMOS33 } [get_ports { OUTPUT[2] }];
#IO_L17N_T2_A25_15 Sch=led[2]

set_property -dict { PACKAGE_PIN N14  IOSTANDARD LVCMOS33 } [get_ports { OUTPUT[3] }];
#IO_L8P_T1_D11_14 Sch=led[3]

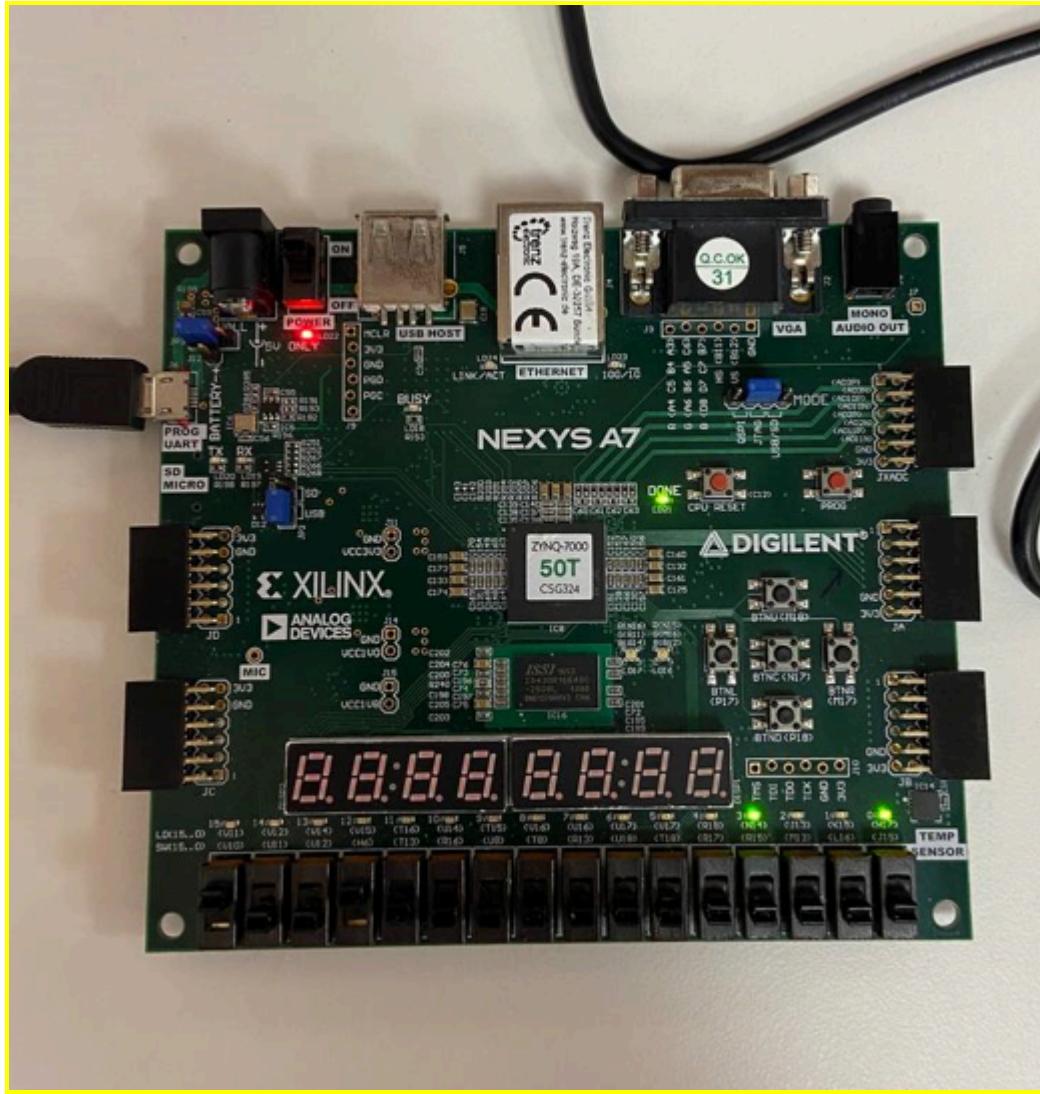
```

- Il bottone di reset è lo stesso dell'esercizio precedente :

```
##Buttons
```

```
set_property -dict { PACKAGE_PIN N17 IOSTANDARD LVCMOS33 } [get_ports { RST }];
#IO_L9P_T1_DQS_14 Sch=btnc
```

Nella foto si evince che è stata selezionata la configurazione di ingresso 1001. Poiché in quella posizione è memorizzato il valore '1001-1001', in uscita otterremo i 4 bit più significativi, ovvero 1001.



Capitolo 2: Reti sequenziali elementari

Esercizio 3 : Riconoscitore di sequenze

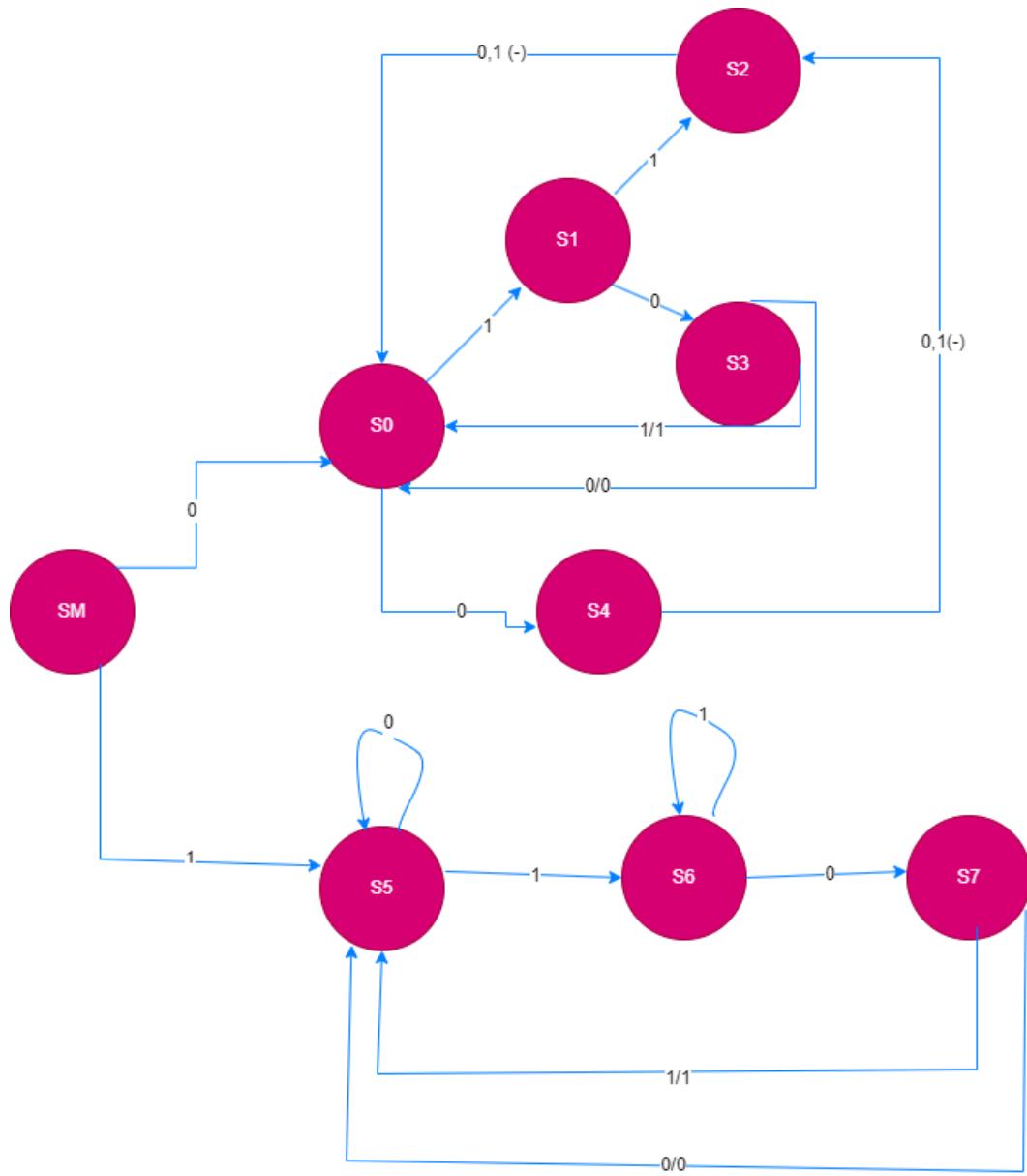
Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza 101. La macchina prende in ingresso un segnale binario i che rappresenta il dato, un segnale A di temporizzazione e un segnale M di modo, che ne disciplina il funzionamento, e fornisce un'uscita Y alta quando la sequenza viene riconosciuta. In particolare,

- se $M=0$, la macchina valuta i bit seriali in ingresso a gruppi di 3 (sequenze non sovrapposte),

- se $M=1$, la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta (sequenze parzialmente sovrapposte).

Progetto e architettura

Innanzitutto abbiamo descritto il funzionamento di questa macchina tramite un automa a stati finiti di Mealy in cui viene distinto il funzionamento a seconda del valore di M . Se M è basso, l'automa valuta le sequenze non sovrapposte mentre, in caso M sia alto, l'automa valuta le sequenze parzialmente sovrapposte.



L'automa prevede diversi stati, definiti nella variabile **curr_state**, che regolano il flusso del riconoscimento:

1. Stato SM (Selezione modalità)

- All'avvio, il sistema si trova in **SM**.
 - Se **load_m** è attivo, il valore di **M** determina quale modalità sarà utilizzata:
 - Se **M = 0**, il sistema entra nello stato **S0** (modalità senza sovrapposizione).
 - Se **M = 1**, il sistema entra nello stato **S5** (modalità con sovrapposizione).

2. Modalità senza sovrapposizione ($M = 0$)

- Il sistema inizia in **S0**.
 - Se il primo bit ricevuto è '1', il sistema passa a **S1**.

- Se successivamente riceve uno '0', passa a **S3**.
- Se dopo **S3** riceve un '1', la sequenza "101" è stata riconosciuta e il segnale di uscita **y** viene impostato a '1'.
- Il sistema poi ritorna in **S0**, pronto per cercare una nuova sequenza.

3. Modalità con sovrapposizione (**M = 1**)

- Il sistema inizia in **S5**.
- Se riceve un '1', passa a **S6**.
- Se il bit successivo è '0', si sposta in **S7**.
- Se dopo **S7** riceve un '1', viene riconosciuta la sequenza "101" e **y** viene impostato a '1'.
- Tuttavia, invece di ripartire da zero, il sistema torna in **S5** per riutilizzare l'ultimo '1' come punto di partenza per la prossima sequenza.

Implementazione

Il riconoscitore ha i seguenti segnali di ingresso e uscita:

- **M**: Modalità di funzionamento (0 per la modalità senza sovrapposizione, 1 per la modalità con sovrapposizione).
- **a**: Bit di ingresso che rappresenta il flusso di dati.
- **clk**: Segnale di clock.
- **rst**: Segnale di reset sincrono che riporta il sistema allo stato iniziale.
- **load_m**: Segnale di abilitazione per la configurazione della modalità.
- **load_a**: Segnale di abilitazione per il caricamento dei dati.
- **y**: Segnale di uscita che si attiva quando la sequenza "101" viene riconosciuta.

Si è scelto di usare un approccio Behavioral per l'implementazione in VHDL della macchina. È doveroso precisare che il riconoscitore di sequenze implementato in VHDL è una **macchina a stati finiti sincrona**, ovvero un circuito sequenziale in cui i cambiamenti di stato avvengono in corrispondenza di un segnale di clock. In particolare, questa macchina si aggiorna a **ogni fronte di salita del clock** (rising edge), garantendo così un funzionamento ordinato e prevedibile. Nel codice VHDL riportato di seguito, la macchina si aggiorna usando la condizione "*if rising_edge(clk) then*", la quale, garantisce che il blocco di codice al suo interno venga eseguito **solamente quando il segnale clk passa da 0 a 1**. Ogni cambiamento di stato avviene quindi in momenti ben definiti, evitando aggiornamenti incontrollati del sistema. Di seguito è riportato il codice:

```
library IEEE;
```

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

```

entity ric_sequence is

Port (
    M      : in std_logic; -- Modalità (0: non sovrapposte, 1: sovrapposte)
    a      : in std_logic; -- dato
    clk    : in std_logic;
    rst   : in std_logic; --segnale di reset
    --clock_frequency_in : integer := 50000000;
    --clock_frequency_out : integer := 500;
    load_m : in std_logic;
    load_a: in std_logic;
    y     : out std_logic -- Segnale di uscita: alto se la sequenza 101 è riconosciuta
);


```

```
end ric_sequence;
```

```

architecture Behavioral of ric_sequence is

type state_sig is (sM, s0, s1, s2, s3, s4, s5, s6, s7);
signal curr_state : state_sig;

begin
y_state: process(clk)
begin
if rising_edge(clk) then
    if(rst='1') then
        curr_state <= sM;
        y <= '0';
    end if;
    if(curr_state = s0) then
        curr_state <= s1;
        y <= '1';
    end if;
    if(curr_state = s1) then
        curr_state <= s2;
        y <= '0';
    end if;
    if(curr_state = s2) then
        curr_state <= s3;
        y <= '1';
    end if;
    if(curr_state = s3) then
        curr_state <= s4;
        y <= '0';
    end if;
    if(curr_state = s4) then
        curr_state <= s5;
        y <= '1';
    end if;
    if(curr_state = s5) then
        curr_state <= s6;
        y <= '0';
    end if;
    if(curr_state = s6) then
        curr_state <= s7;
        y <= '1';
    end if;
    if(curr_state = s7) then
        curr_state <= sM;
        y <= '0';
    end if;
end if;
end process;

```

```

else

case curr_state is

when sM =>          --blocco di codice che setta la modalità di funzionamento guardando
l'abilitazione e il valore di M

if(load_m = '1') then

if(M = '0') then

curr_state <= s0;

y <= '0';

else

curr_state <= s5;

y <= '0';

end if;

end if;

when s0 =>

if(load_a = '1') then

if(a = '0')then

curr_state <= s4;

y <= '0';

else

curr_state <= s1;

y <= '0';

end if;

end if;

when s1 =>

if(load_a = '1') then

if(a = '0')then

```

```
curr_state <= s3;  
y <= '0';  
else  
curr_state <= s2;  
y <= '0';  
end if;  
end if;
```

```
when s2 =>  
if(load_a = '1') then  
if(a = '-')then  
curr_state <= s0;  
y <= '0';  
end if;  
end if;
```

```
when s3 =>  
if(load_a = '1') then  
if(a = '0')then  
curr_state <= s0;  
y <= '0';  
else  
curr_state <= s0;  
y <= '1';  
end if;  
end if;
```

```
when s4 =>

if(load_a = '1') then

    if(a = '-')then

        curr_state <= s2;

        y <= '0';

    end if;

end if;
```

```
when s5 =>

if(load_a = '1') then

    if(a = '0')then

        curr_state <= s5;

        y <= '0';

    else

        curr_state <= s6;

        y <= '0';

    end if;

end if;
```

```
when s6 =>

if(load_a = '1') then

    if(a = '0')then

        curr_state <= s7;

        y <= '0';

    else
```

```

curr_state <= s6;
y <= '0';
end if;
end if;

when s7 =>
if(load_a = '1') then
  if(a = '0')then
    curr_state <= s5;
    y <= '0';
  else
    curr_state <= s5;
    y <= '1';
  end if;
end if;

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

end Behavioral;

```

Simulazione

Il testbench da noi realizzato per il riconoscitore di sequenze è stato strutturato in due parti, affinché potessimo verificare il corretto comportamento della macchina in entrambe le modalità; la prima operazione eseguita è il reset del sistema, che assicura che la macchina inizi la simulazione nello stato corretto. Successivamente, viene testata la modalità non sovrapposta ("M = 0"). In questa fase, la modalità viene caricata e si fornisce la sequenza "101" verificando che il segnale di uscita "y" si attivi solo quando l'intera sequenza è stata completata. Si verifica poi una sequenza non valida, come "010", controllando che

"y" rimanga a zero. Dopo un ulteriore reset, viene testata la modalità sovrapposta ("M = 1"). In questo caso, il test verifica che il riconoscitore sia in grado di identificare più occorrenze consecutive della sequenza "101" senza dover ripartire dallo stato iniziale. Se il riconoscitore funziona correttamente, il segnale "y" dovrebbe attivarsi ogni volta che una nuova sequenza "101" viene rilevata, anche se questa condivide alcuni bit con la precedente.

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

entity ric_sequence_tb is
    -- Entity vuota per il testbench
end ric_sequence_tb;

architecture Behavioral of ric_sequence_tb is
    -- Dichiarazione dei segnali per testare il modulo
    signal M      : std_logic := '0';
    signal a      : std_logic := '0';
    signal clk    : std_logic := '0';
    signal rst    : std_logic := '0';
    signal load_m : std_logic := '0';
    signal load_a : std_logic := '0';
    signal y      : std_logic;

    constant clock_period : time := 10 ns; -- Periodo del clock (100 MHz)

begin
    -- Instanziare l'unità sotto test (UUT)
    uut: entity work.ric_sequence
        port map (
            M => M,
            a => a,
            clk => clk,
```

```

rst => rst,
load_m => load_m,
load_a => load_a,
y => y
);

-- Generazione del segnale di clock
clk_process: process
begin
clk <= '0';
wait for clock_period / 2;
clk <= '1';
wait for clock_period / 2;
end process;

-- Stimoli del testbench
stim_proc: process
begin
-- Resetta il sistema
rst <= '1';
wait for clock_period;
rst <= '0';
wait for clock_period;

-- Modalità 0 (sequenze non sovrapposte)
M <= '0';
load_m <= '1'; -- Abilita il caricamento della modalità
wait for clock_period;
load_m <= '0';
wait for clock_period;

-- Sequenza: 101 (non sovrapposta)

```

```
a <= '1'; load_a <= '1'; wait for clock_period;  
a <= '0'; load_a <= '1'; wait for clock_period;  
a <= '1'; load_a <= '1'; wait for clock_period;  
load_a <= '0'; -- Pausa  
wait for clock_period;
```

-- Sequenza: 010 (non valida)

```
a <= '0'; load_a <= '1'; wait for clock_period;  
a <= '1'; load_a <= '1'; wait for clock_period;  
a <= '0'; load_a <= '1'; wait for clock_period;  
load_a <= '0'; -- Pausa  
wait for clock_period;
```

```
rst <= '1';  
wait for clock_period;  
rst <= '0';  
wait for clock_period;
```

-- Modalità 1 (sequenze sovrapposte)

```
M <= '1';  
load_m <= '1'; -- Abilita il caricamento della modalità  
wait for clock_period;  
load_m <= '0';  
wait for clock_period;
```

-- Sequenza: 101 (sovrapposta)

```
a <= '1'; load_a <= '1'; wait for clock_period;  
a <= '0'; load_a <= '1'; wait for clock_period;  
a <= '1'; load_a <= '1'; wait for clock_period; -- `y` dovrebbe andare a '1'  
a <= '0'; load_a <= '1'; wait for clock_period; -- Inizia una nuova sequenza  
a <= '1'; load_a <= '1'; wait for clock_period; -- `y` dovrebbe andare di nuovo a '1'  
load_a <= '0'; -- Pausa
```

```
wait for clock_period;
```

```
-- Concludere il test
```

```
wait;
```

```
end process;
```

```
end Behavioral;
```

Di seguito è riportato l'output del riconoscitore di sequenza ottenuto mediante la simulazione:



Timing analysis

Successivamente alla sintesi è stata effettuata la Timing Analysis del sistema complessivo, precisamente dopo la fase di place&route per avere delle stime più accurate. E' stato valutato il funzionamento del sistema con un clock di frequenza 100MHz (periodo 10ns). Prima di parlare della timing analysis del progetto è opportuno definire i parametri che caratterizzano un'analisi della temporizzazione di un circuito:

- Slack: differenza tra il tempo di arrivo di un segnale e il tempo richiesto per soddisfare il vincolo che si sta valutando. Se la differenza è positiva allora i vincoli (di setup e di hold) sono rispettati.
- Tempo di setup: stabilisce per quanto tempo un segnale deve essere stabile prima del fronte attivo del segnale di abilitazione (clock). Il vincolo di setup è valutato per ogni percorso del circuito ed è detto soddisfatto se si ha uno slack positivo, in particolare se il parametro WNS (Worst Negative Slack) è positivo. Il tempo di setup ed il relativo slack sono utili per determinare la massima frequenza operativa del circuito poiché riguarda l'analisi del ritardo massimo del circuito.
- Tempo di hold: stabilisce per quanto tempo un segnale deve essere stabile dopo il fronte attivo del segnale di abilitazione (clock). Il vincolo di hold è valutato per ogni percorso del circuito ed è detto soddisfatto se si ha uno slack positivo, in particolare se il parametro WHS (Worst Hold Slack) è positivo. L'analisi del tempo di hold equivale all'analisi del ritardo minimo del circuito
- Larghezza d'impulso: la larghezza d'impulso o "pulse width" (PW). Lo slack relativo (WPWS - Worst Pulse Width Slack) indica quanto margine c'è tra la larghezza effettiva di un impulso di clock (o di un segnale) e la larghezza minima richiesta, se è positivo il vincolo sulla larghezza degli impulsi è rispettata.

Di seguito è riportata la Timing Analysis del riconoscitore di sequenza implementato (esercizio 3.1):

```

Timing Summary - Post-Place Phys Opt Design - impl_1
C:/Users/mauro/Desktop/magistrale/ASD/esercizi_vivado/esercizio3_finale/ric_sequence/runs/impl_1/control_unit_timing_summary_physopted.rpt
Read-only

196: -----
197: Path Group      From Clock      To Clock
198: (none)          sys_clk_pin   -----
199: (none)          sys_clk_pin   sys_clk_pin
200: -----
201: -----
202: -----
203: -----
204: -- Timing Details
205: | -----
206: | -----
207: | -----
208: | -----
209: | -----
210: | -----
211: | From Clock: sys_clk_pin
212: | To Clock:  sys_clk_pin
213: | -----
214: | Setup :    0 Failing Endpoints, Worst Slack      5.143ns, Total Violation      0.000ns
215: | Hold :     0 Failing Endpoints, Worst Slack      0.233ns, Total Violation      0.000ns
216: | PW :       0 Failing Endpoints, Worst Slack      4.500ns, Total Violation      0.000ns
217: -----

```

Dalla figura si può notare che tutti i vincoli sono rispettati e che non ci sono violazioni di nessun tipo. E' possibile calcolare il minor periodo che soddisfi il vincolo di setup come:

$$T_{max} = T - WNS = 10 - 5,143 = 4,857\text{ns}$$

La massima frequenza operativa del circuito è data dalla formula:

$$F_{max} = 1/T_{max} = 1/4,857 \cong 0,20\text{GHz}$$

Esercizio 3.2: Implementazione riconoscitore di sequenza su board

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

Sintesi su board di sviluppo

Basandoci sul componente realizzato nel precedente esercizio, è stata introdotta una nuova entità “*unità di controllo*”, in cui sono dichiarati due *Button_Debouncer*, uno per il bottone B1 e uno per il bottone B2. Questi ultimi servono, rispettivamente, per mappare l'ingresso i e il load tramite gli switch. Infine, è presente un componente “*ric_sequence*” che è, di fatto, il modulo realizzato precedentemente in grado di riconoscere la sequenza 101.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity control_unit is
PORT(
    B1 : IN STD_LOGIC;
    B2 : IN STD_LOGIC;

```

```

S1 : IN STD_LOGIC; -- switch per D
S2 : IN STD_LOGIC; -- switch per M
CLK : IN STD_LOGIC;
led: OUT STD_LOGIC

);end control_unit;

architecture Behavioral of control_unit is

COMPONENT ButtonDebouncer IS
GENERIC (
    CLK_period: integer := 10;
    btn_noise_time: integer := 10000000
);
PORT ( RST : in STD_LOGIC;
       CLK : in STD_LOGIC;
       BTN : in STD_LOGIC;
       CLEARED_BTN : out STD_LOGIC);
end COMPONENT;

COMPONENT ric_sequence is
Port (
    M      : in  std_logic; -- Modalità (0: non sovrapposte, 1: sovrapposte)
    a      : in  std_logic; -- dato
    clk    : in  std_logic;
    rst   : in std_logic; --segnale di reset
    --clock_frequency_in : integer := 50000000;

```

```

--clock_frequency_out : integer := 500;
load_m : in std_logic;
load_a: in std_logic;
y      : out std_logic -- Segnale di uscita: alto se la sequenza 101 è riconosciuta
);

end COMPONENT;

SIGNAL cleared_i : STD_LOGIC ;
SIGNAL cleared_m : STD_LOGIC;

begin
deb_i : ButtonDebouncer
PORT MAP(
    RST => '0',
    CLK => CLK,
    BTN => B1,
    CLEARED_BTN => cleared_i
);

deb_m : ButtonDebouncer
PORT MAP(
    RST => '0',
    CLK => CLK,
    BTN => B2,
    CLEARED_BTN => cleared_m
);

```

```

ric : ric_sequence
PORT MAP(
    M => S2,
    a => S1,
    clk => CLK,
    rst => '0',
    load_m => cleared_m,
    load_a => cleared_i,
    y => led
);

end Behavioral;

```

Possiamo dividere il funzionamento di questa macchina in due sezioni:

Gestione della pressione dei pulsanti (Debouncing)

- Sono presenti due pulsanti(B1 e B2) i cui segnali possono essere soggetti a disturbi dovuti alle oscillazioni. Per garantire che il sistema interpreti solo pressioni effettive e non impulsi indesiderati, i segnali dei pulsanti vengono prima filtrati da due istanze del componente **ButtonDebouncer**.
- Questo componente prende come ingresso il segnale grezzo del pulsante (BTN), il clock (CLK) e restituisce un segnale pulito (CLEARED_BTN), privo di oscillazioni indesiderate.
- **cleared_i** e **cleared_m** rappresentano rispettivamente i segnali puliti derivanti dai pulsanti B1 e B2

Riconoscimento della sequenza binaria

- Il componente **ric_sequence** è un modulo in grado di riconoscere una particolare sequenza di bit (101).
- Questo componente prende in ingresso:
 - M (S2): modalità di funzionamento, che potrebbe determinare se il riconoscimento avviene in modalità sovrapposta o non sovrapposta.
 - a (S1): il dato in ingresso, cioè il bit che alimenta il riconoscitore di sequenza.

- `clk` (CLK): il segnale di clock per la sincronizzazione.
- `rst` (reset, fissato a '`0`', quindi non viene mai attivato).
- `load_m` e `load_a`: segnali derivati dai pulsanti B2 e B1, utilizzati per caricare nuovi valori.
- Quando la sequenza "101" viene riconosciuta correttamente, l'uscita `y` del componente `ric_sequence` si attiva, accendendo così il LED (`led`).

Di seguito viene riportato il file di constraint mediante il quale sono stati mappati i vari componenti.

```

## Clock signal

set_property -dict { PACKAGE_PIN E3      IO_STANDARD LVCMOS33 } [get_ports {
CLK }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5}
[get_ports {CLK}];

##Switches

set_property -dict { PACKAGE_PIN J15     IO_STANDARD LVCMOS33 } [get_ports {
S1 }]; #IO_L24N_T3_RS0_15 Sch=sw[0]

set_property -dict { PACKAGE_PIN L16     IO_STANDARD LVCMOS33 } [get_ports {
S2 }]; #IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]

#set_property -dict { PACKAGE_PIN M13     IO_STANDARD LVCMOS33 } [get_ports
{ SW[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]

## LEDs

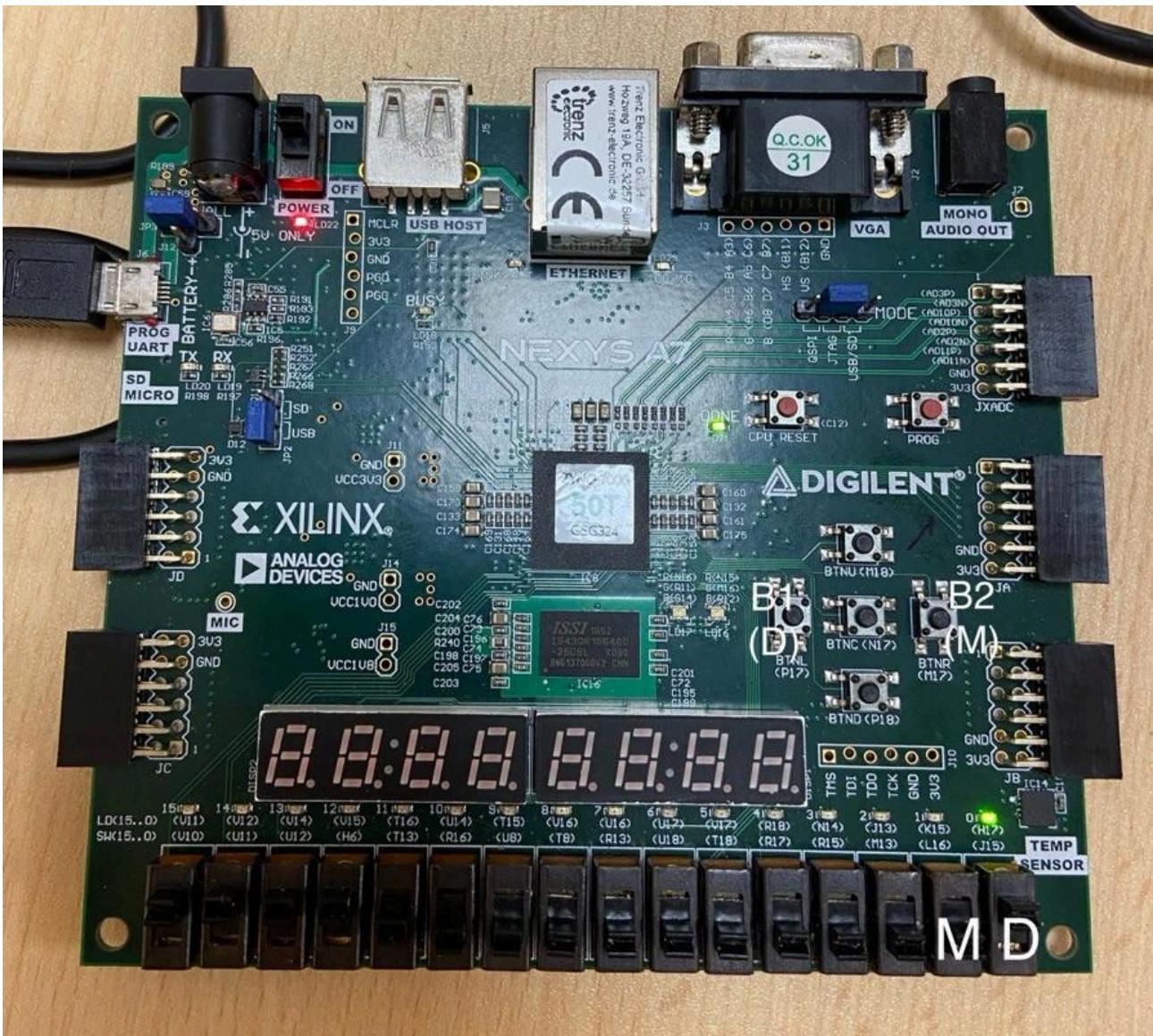
set_property -dict { PACKAGE_PIN H17     IO_STANDARD LVCMOS33 } [get_ports {
led }]; #IO_L18P_T2_A24_15 Sch=led[0]

##Buttons

set_property -dict { PACKAGE_PIN P17     IO_STANDARD LVCMOS33 } [get_ports {
B1 }]; #IO_L12P_T1_MRCC_14 Sch=btnl

set_property -dict { PACKAGE_PIN M17     IO_STANDARD LVCMOS33 } [get_ports {
B2 }]; #IO_L10N_T1_D15_14 Sch=btnr

```



Esercizio 4: Shift Register

Progettare, implementare in VHDL e testare mediante simulazione un registro a scorrimento di N bit in grado di shiftare a destra o a sinistra di un numero Y variabile di posizioni a seconda di una opportuna selezione. In particolare, i valori possibili di Y sono 1 e 2. L'utente tramite selezione deve scegliere di quante posizioni shiftare. Il componente deve essere realizzato utilizzando sia a) approccio comportamentale sia un b) approccio strutturale.

Nota: il numero di bit del registro deve essere implementato come un generic, e dall'esterno deve poter essere scelta la modalità di funzionamento mediante opportuni segnali di selezione.

Progetto e architettura

Per lo sviluppo dello shift register abbiamo adottato un approccio comportamentale, realizzando un registro capace di memorizzare N bit, dove N è definito come parametro generico.

Il componente prevede in ingresso:

- X: il dato seriale da inserire,

- Y: il segnale che stabilisce se effettuare uno shift di una o due posizioni (Y='0' per una posizione, Y='1' per due posizioni),
- M: il segnale che indica la direzione dello shift (M='0' per uno shift verso destra, M='1' per uno shift verso sinistra),
- clk: segnale di clock,
- rst: segnale di reset asincrono attivo alto.

L'uscita è rappresentata da:

- output: il bit seriale in uscita, che corrisponde al primo o all'ultimo bit del registro a seconda della direzione,
- temp_out: il contenuto completo del registro, utile a scopo di debug.

All'interno dell'architettura è stato dichiarato un segnale ausiliario chiamato temp, un vettore di N bit, che rappresenta il contenuto effettivo del registro.

Il comportamento è definito nel seguente modo:

- Reset: quando rst è attivo ('1'), il contenuto del registro viene azzerato (tutti i bit posti a '0').
- Fronte di salita del clock:
 - Se Y = '0' (shift di una posizione):
 - M = '0': shift verso destra. Il nuovo dato (X) viene inserito nella prima posizione (bit 0), mentre tutti gli altri bit vengono traslati di una posizione verso destra.
 - M = '1': shift verso sinistra. Il dato seriale (X) viene inserito nell'ultima posizione (bit N-1) e tutti gli altri bit si spostano di una posizione verso sinistra.
 - Se Y = '1' (shift di due posizioni):
 - M = '0': shift verso destra. I primi due bit vengono aggiornati: nella prima posizione viene posto il nuovo dato (X) e nella seconda il vecchio bit 0. Gli altri bit avanzano di due posizioni verso destra.
 - M = '1': shift verso sinistra. Gli ultimi due bit vengono aggiornati: nella penultima posizione si inserisce il nuovo dato (X), mentre l'ultimo bit mantiene il valore precedente. I restanti bit vengono traslati di due posizioni verso sinistra.

Infine, il valore dell'uscita seriale (output) viene selezionato dinamicamente:

- Se lo shift è verso destra (M='0'), viene prelevato il primo bit (temp(0)).
- Se lo shift è verso sinistra (M='1'), viene prelevato l'ultimo bit (temp(N-1)).

In questo modo, il registro realizza un comportamento serie-serie, accettando un dato seriale in ingresso e producendo un dato seriale in uscita, adattandosi dinamicamente sia alla direzione sia all'ampiezza dello shift.

Implementazione (Behavioral)

```
entity shift_register is
```

```
  generic(
```

```

N: integer := 4
);

Port (
    Y : in std_logic; --ingresso che serve per decidere se shiftare di uno o due posizioni
    M: in std_logic; --ingresso utilizzato per decidere se shiftare verso destra o sinistra
    X: in std_logic; --input

    clk: in std_logic;
    rst: in std_logic;

    output: out std_logic;
    temp_out : out std_logic_vector(N-1 downto 0) -- Debug: registro interno
);
end shift_register;

```

architecture Behavioral of shift_register is

```

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

begin
    process(clk, rst)
        begin
            if(rising_edge(clk)) then
                if(rst = '1') then
                    temp <= (others => '0');
                else
                    if(Y = '0') then --se y=0 si shifta solo di una posizione
                        if(M = '0') then --se M=0 si imposta lo scorrimento verso destra
                            temp(0) <= X;
                            temp(N-1 downto 1) <= temp(N-2 downto 0);
                        else      --gestiamo il caso in cui M=1 e si imposta lo scorrimento verso sinistra
                            temp(N-1) <= X;
                        end if;
                    end if;
                end if;
            end if;
        end process;
    end;

```

```

temp(N-2 downto 0) <= temp(N-1 downto 1); --?
end if;

else --gestiamo il caso in cui Y=1 e si shifta di due posizioni
if(M = '0') then --se M=0 si imposta lo scorrimento verso destra
    temp(1 downto 0) <= (X & temp(0));
    temp(N-1 downto 2) <= temp(N-3 downto 0);
else --gestiamo il caso in cui M=1 e si imposta lo scorrimento verso sinistra
    temp(N-1 downto N-2) <= (temp(N-1) & X);
    temp(N-3 downto 0) <= temp(N-1 downto 2); --?
end if;

end if;
end if;

end if;

end process;

with M select
output <= temp(0) when '0',
temp(N-1) when '1',
'-' when others;

temp_out <= temp;
end Behavioral;

```

Simulazione (Behavioral)

```

entity tb_shift_register_M1_Y1 is
end tb_shift_register_M1_Y1;

```

```

architecture sim of tb_shift_register_M1_Y1 is
constant N : integer := 4;

```

```
signal clk, rst, M, Y, X : std_logic := '0';
signal output : std_logic;
signal temp_out : std_logic_vector(N-1 downto 0); -- Per monitorare il registro interno
```

```
component shift_register is
```

```
    generic (N : integer := 4);
    port (
        Y : in std_logic;
        M : in std_logic;
        X : in std_logic;
        clk : in std_logic;
        rst : in std_logic;
        output : out std_logic;
        temp_out : out std_logic_vector(N-1 downto 0) -- Debug
    );

```

```
end component;
```

```
begin
```

```
    uut: shift_register
        generic map (N => N)
        port map (Y => Y, M => M, X => X, clk => clk, rst => rst, output => output, temp_out => temp_out);
```

```
clk_process: process
```

```
begin
```

```
    while true loop
```

```
        clk <= '1';
```

```
        wait for 10 ns;
```

```
        clk <= '0';
```

```
        wait for 10 ns;
```

```
    end loop;
```

```
end process;
```

stimulus: process

begin

rst <= '1'; wait for 20 ns; rst <= '0'; -- Reset

Y <= '0'; M <= '1'; -- Shift di 1 verso sinistra

X <= '1'; wait for 20 ns;

X <= '0'; wait for 20 ns;

X <= '1'; wait for 20 ns;

wait for 20ns;

rst <= '1'; wait for 20 ns; rst <= '0';

Y <= '0'; M <= '0';

X <= '1'; wait for 20 ns;

X <= '0'; wait for 20 ns;

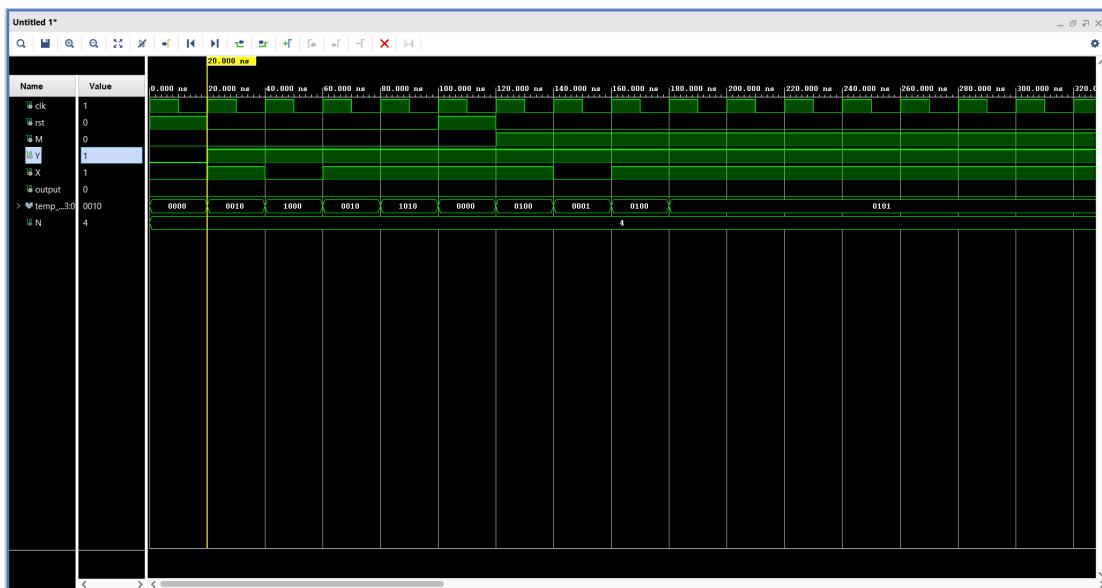
X <= '1'; wait for 20 ns;

wait;

end process;

end sim;

Di seguito i risultati della simulazione:



Implementazione (Structural)

Nel caso dell'implementazione strutturale, il primo passo è stato progettare l'architettura del flip-flop D, adottando un approccio comportamentale. Questo flip-flop rappresenta uno dei due blocchi fondamentali necessari per costruire il registro.

L'altro componente richiesto è il multiplexer 4:1 composto dall'unione di multiplexer 2:1 (vedi *Appendice*). Di seguito il codice del flip-flop di tipo D e del componente principale shift register.

```
entity flip_flop_d is
    Port ( d : in STD_LOGIC; -- Dato di ingresso
           enable : in STD_LOGIC; -- Segnale di abilitazione alla lettura di d, spesso temporizzato
           q : out STD_LOGIC); -- Dato salvato in uscita
end flip_flop_d;

architecture Dataflow of flip_flop_d is

begin
    ff: process(enable)
    begin
        if(rising_edge(enable)) then -- Serve che venga abilitato l'enable
            q <= d; -- Sovrascrittura di q
        end if;
    end process;

end Dataflow;

-----
entity shift_register is
    Generic(
        Size : positive := 4); -- Dimensione del registro (con valore predefinito in modo da rappresentarne la schematic)
    Port(
        value_in : in STD_LOGIC; -- Valore da inserire una o due volte
        data_in : in STD_LOGIC_VECTOR(Size-1 downto 0); -- Registro su cui inserire il valore
```

```

load : in STD_LOGIC; -- Abilitazione del caricamento su registro
clock : in STD_LOGIC; -- Segnale di temporizzazione
shift_en : in STD_LOGIC; -- Segnale di abilitazione allo shift effettivo
direction : in STD_LOGIC; -- Direzione dello shift ('0' se destra, '1' se sinistra)
length : in STD_LOGIC; -- Numero di inserimenti ('0' un inserimento, '1' due inserimenti)
data_out : out STD_LOGIC_VECTOR(Size-1 downto 0)); -- Valore (o coppia di valori) che esce dal registro
dopo lo shift
end shift_register;

```

architecture Structural of shift_register is

```

component gen_mux_2_1 is
  Generic( Size : natural); -- Dimensione dei dati
  Port ( a0 : in STD_LOGIC_VECTOR(Size-1 downto 0); -- Input 1
         a1 : in STD_LOGIC_VECTOR(Size-1 downto 0); -- Input 2
         en : in STD_LOGIC; -- Segnale di Enable: l'output cambia quando è '1'
         s : in STD_LOGIC; -- Segnale di Selezione dell'Input
         y : out STD_LOGIC_VECTOR(Size-1 downto 0)); -- Output
end component;

```

```

component mux_4_1 is
  Port ( a : in STD_LOGIC_VECTOR (3 downto 0); -- I 4 Input, in un vettore
         en : in STD_LOGIC; -- Segnale di Enable: l'output cambia quando è '1'
         s : in STD_LOGIC_VECTOR (1 downto 0); -- Segnali codificati di Selezione dell'Input
         y : out STD_LOGIC); -- Output
end component;

```

```

component flip_flop_d is
  Port ( d : in STD_LOGIC; -- Dato di ingresso
         enable : in STD_LOGIC; -- Segnale di abilitazione alla lettura di d, spesso temporizzato
         q : out STD_LOGIC); -- Dato salvato in uscita
end component;

```

```
type mux_4_data IS ARRAY (Size-1 downto 0) of STD_LOGIC_VECTOR(3 downto 0); -- Definiamo un
typedef per un array di vettori per i vari multiplexer 4:1
```

```
signal mux_in : mux_4_data; -- Definiamo tale array, dove terremo i vari valori con cui sarà possibile
sostituire in shift il dato originale
```

```
signal clock_ff : STD_LOGIC; -- Segnale che abilita i flip flop
```

```
signal ff_in : STD_LOGIC_VECTOR(Size-1 downto 0); -- Dato temporaneo in ingresso al flip-flop (deciso da
mux_in)
```

```
signal ff_out : STD_LOGIC_VECTOR(Size-1 downto 0); -- Dato temporaneo in uscita dal flip-flop (sarà sia
rielaborato che l'output)
```

```
signal data_tmp : STD_LOGIC_VECTOR(Size-1 downto 0); -- Dato temporaneo su cui salvare in caso di
doppio shift (serve rifare l'operazione)
```

```
begin
```

```
data_out <= ff_out; -- Salvataggio dell'output
```

```
clock_ff <= clock AND shift_en; -- I flip flop devono essere sincronizzati, ma anche attivi solo in fase di
shift
```

```
-- Siccome la dimensione dei dati è generic, ci serve un numero variabile di componenti: ne dichiareremo
alcuni per ogni bit
```

```
component_manager: for i in Size-1 downto 0 generate
```

```
-- Un Multiplexer 4:1 si occupa di selezionare la modalità di shift, fornendo il dato che poi sostituiranno
```

```
shifter_mux: mux_4_1 port map(
```

```
a => mux_in(i), -- a(0): shift di 1 a destra, a(1): shift di 2 a destra, a(2): shift di 1 a sinistra, a(3): shift di
2 a sinistra
```

```
en => '1', -- Abilitazione perennemente attiva
```

```
s(0) => length, -- Permette di scegliere fra a(0)/a(2) o a(1)/a(3)
```

```
s(1) => direction, -- Permette di scegliere fra a(0)/a(1) o a(2)/a(3)
```

```
y => data_tmp(i)); -- Questo sarà il dato con il quale si effettuerà la sostituzione
```

```
-- Un Multiplexer 2:1 che permette una modifica dei dati solo quando si ha un'abilitazione
```

```
loader_mux: gen_mux_2_1 generic map(Size => 1)
```

```
port map(
    a0(0) => data_in(i), -- In caso di non shift il dato in uscita deve essere quello originale
    a1(0) => data_tmp(i), -- In caso di shift si avrà il nuovo dato, proveniente dal multiplexer 4:1
    en => '1', -- Abilitazione perennemente attiva
    s => load, -- Il load funge da selezione
    y(0) => ff_in(i)); -- L'uscita sarà ciò che effettivamente entrerà nel flip flop, o il dato originale o lo
shiftato
```

-- Un Flip-Flop D che sovrascrive i registri

```
ff_gen: flip_flop_d port map(
    d => ff_in(i), -- Dato in ingresso
    enable => clock, -- Abilitazione temporizzata dal clock
    q => ff_out(i)); -- Dato modificato
```

```
end generate;
```

-- Dobbiamo specificare agli shifter_mux cosa sostituire: dipenderà dalla Size

```
linkage_2 : if (Size = 2) generate
```

```
    msb: mux_in(1) <= value_in & ff_out(0) & value_in & value_in;
```

```
    lsb: mux_in(0) <= value_in & value_in & value_in & ff_out(1);
```

```
end generate;
```

-- Altro caso particolare, dove esiste anche un valore intermedio

```
linkage_3 : if (Size = 3) generate
```

```
    msb: mux_in(2) <= ff_out(0) & ff_out(1) & value_in & value_in;
```

```
    middle_bit: mux_in(1) <= value_in & ff_out(0) & value_in & ff_out(2);
```

```
    lsb: mux_in(0) <= value_in & value_in & ff_out(2) & ff_out(1);
```

```
end generate;
```

-- Caso generale: i quattro estremi sono sempre gli stessi e gli eventuali intermedi sono ricorsivi

```
linkage_size: if (Size > 3) generate
```

```
    msb: mux_in(Size-1) <= ff_out(Size-3) & ff_out(Size-2) & value_in & value_in;
```

```

msb_min : mux_in(Size-2) <= ff_out(Size-4) & ff_out(Size-3) & value_in & ff_out(Size-1);

middle_bits: for i in Size-3 downto 2 generate

    mux_in(i) <= ff_out(i-2) & ff_out(i-1) & ff_out(i+2) & ff_out(i+1);

end generate;

lsb_max : mux_in(1) <= value_in & ff_out(0) & ff_out(3) & ff_out(2);

lsb : mux_in(0) <= value_in & value_in & ff_out(2) & ff_out(1);

end generate;

```

end Structural;

architecture Behavioral of shift_register is

```
signal data_tmp : STD_LOGIC_VECTOR(Size-1 downto 0); -- Registro temporaneo per sostituire
```

begin

```
data_out <= data_tmp; -- Sovrascrittura dell'Output
```

main: process(clock) -- Processo temporizzato

begin

if (rising_edge(clock)) then

-- Caricamento del dato

if (load = '1') then

data_tmp <= data_in;

-- Operazione di shift

elsif (shift_en = '1') then

-- Le varie situazioni di shift

if (length = '0' AND direction = '0') then

data_tmp <= value_in & data_tmp (Size-1 downto 1);

elsif (length = '0' AND direction = '1') then

data_tmp <= data_tmp(Size-2 downto 0) & value_in;

```

elsif (length = '1' AND direction = '0') then
    data_tmp <= value_in & value_in & data_tmp (Size-1 downto 2);
else
    data_tmp <= data_tmp(Size-3 downto 0) & value_in & value_in;
end if;
end if;
end if;
end process;
end Behavioral;

```

Simulazione (structural)

```

entity shift_register_structural_tb is
end shift_register_structural_tb;

```

```

architecture shift_register_structural of shift_register_structural_tb is

```

```

signal input : STD_LOGIC := '0';
signal data : STD_LOGIC_VECTOR (5-1 downto 0) := "10110";
signal output : STD_LOGIC_VECTOR (5-1 downto 0) := (others => '0');
signal direction : STD_LOGIC := '0';
signal length : STD_LOGIC := '0';
signal clock : STD_LOGIC := '0';
signal load : STD_LOGIC := '0';
signal enable : STD_LOGIC := '0';

```

```

constant clock_period : TIME := 10 ns;

```

```

begin

```

```

-- Non ♦ necessario dichiarare il component, essendo incluso nella libreria work
utt : entity work.shift_register(structural) generic map(Size => 5)

```

```
port map(
    data_in => data,
    value_in => input,
    load => load,
    clock => clock,
    shift_en => enable,
    direction => direction,
    length => length,
    data_out => output);
```

```
time_proc : process
begin
    clock <= NOT (clock);
    wait for clock_period/2;
    clock <= NOT (clock);
    wait for clock_period/2;
end process;
```

```
main : process
begin
    load <= '0';
    enable <= '0';
    direction <= '0';
    length <= '0';
    wait until clock = '0';
    input <= '0';
    wait until clock = '0';

    input <= '1';
    wait until clock = '0';
    length <= '1';
```

```
input <= '1';
wait until clock = '0';
input <= '0';
wait until clock = '0';
input <= '1';
wait until clock = '0';
input <= '0';
wait until clock = '0';
```

```
direction <= '1';
wait until clock = '0';
length <= '0';
input <= '0';
wait until clock = '0';
```

```
input <= '1';
wait until clock = '0';
length <= '1';
input <= '0';
wait until clock = '0';
input <= '1';
wait until clock = '0';
input <= '0';
wait until clock = '0';
input <= '1';
wait until clock = '0';
```

```
load <= '0';
enable <= '1';
direction <= '0';
length <= '0';
wait until clock = '0';
```

```
input <= '0';
```

```
wait until clock = '0';
```

```
input <= '1';
```

```
wait until clock = '0';
```

```
length <= '1';
```

```
input <= '1';
```

```
wait until clock = '0';
```

```
input <= '0';
```

```
wait until clock = '0';
```

```
input <= '1';
```

```
wait until clock = '0';
```

```
input <= '0';
```

```
wait until clock = '0';
```

```
direction <= '1';
```

```
wait until clock = '0';
```

```
length <= '0';
```

```
input <= '0';
```

```
wait until clock = '0';
```

```
input <= '1';
```

```
wait until clock = '0';
```

```
length <= '1';
```

```
input <= '0';
```

```
wait until clock = '0';
```

```
input <= '1';
```

```
wait until clock = '0';
```

```
length <= '0';
```

```
wait until clock = '0';
```

```
input <= '1';
```

```
wait until clock = '0';
```

```
load <= '1';
```

```
enable <= '0';
```

```
direction <= '0';
```

```
length <= '0';
```

```
wait until clock = '0';
```

```
input <= '0';
```

```
wait until clock = '0';
```

```
input <= '1';
```

```
wait until clock = '0';
```

```
length <= '1';
```

```
input <= '1';
```

```
wait until clock = '0';
```

```
input <= '0';
```

```
wait until clock = '0';
```

```
input <= '1';
```

```
wait until clock = '0';
```

```
input <= '0';
```

```
wait until clock = '0';
```

```
direction <= '1';
```

```
wait until clock = '0';
```

```
length <= '0';
```

```
input <= '0';
```

```
wait until clock = '0';
```

```
input <= '1';
```

```
wait until clock = '0';
```

```
length <= '1';
```

```
input <= '0';
```

```
wait until clock = '0';
```

```
input <= '1';
wait until clock = '0';
input <= '0';
wait until clock = '0';
input <= '1';
wait until clock = '0';
```

```
load <= '1';
enable <= '1';
direction <= '0';
length <= '0';
wait until clock = '0';
input <= '0';
wait until clock = '0';
```

```
input <= '1';
wait until clock = '0';
length <= '1';
input <= '1';
wait until clock = '0';
input <= '0';
wait until clock = '0';
input <= '1';
wait until clock = '0';
input <= '0';
wait until clock = '0';
```

```
direction <= '1';
wait until clock = '0';
length <= '0';
input <= '0';
wait until clock = '0';
```

```

    input <= '1';

    wait until clock = '0';

    length <= '1';

    input <= '0';

    wait until clock = '0';

    input <= '1';

    wait until clock = '0';

    input <= '0';

    wait until clock = '0';

    input <= '1';

    wait until clock = '0';

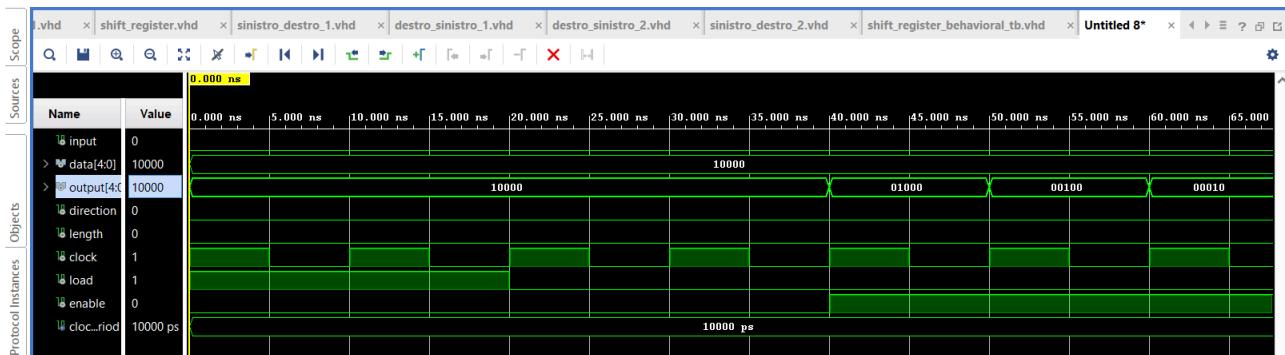
    wait;

end process;

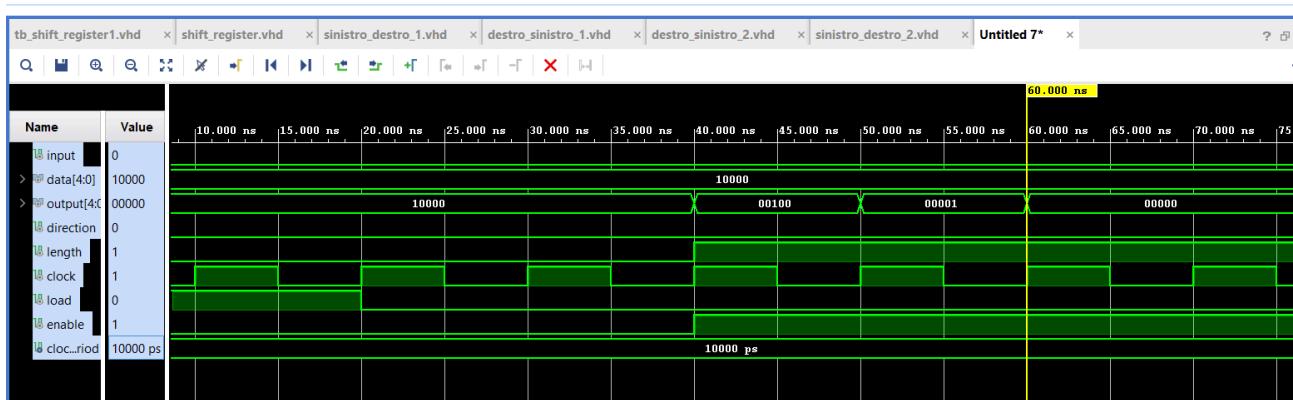
```

end shift_register_structural;

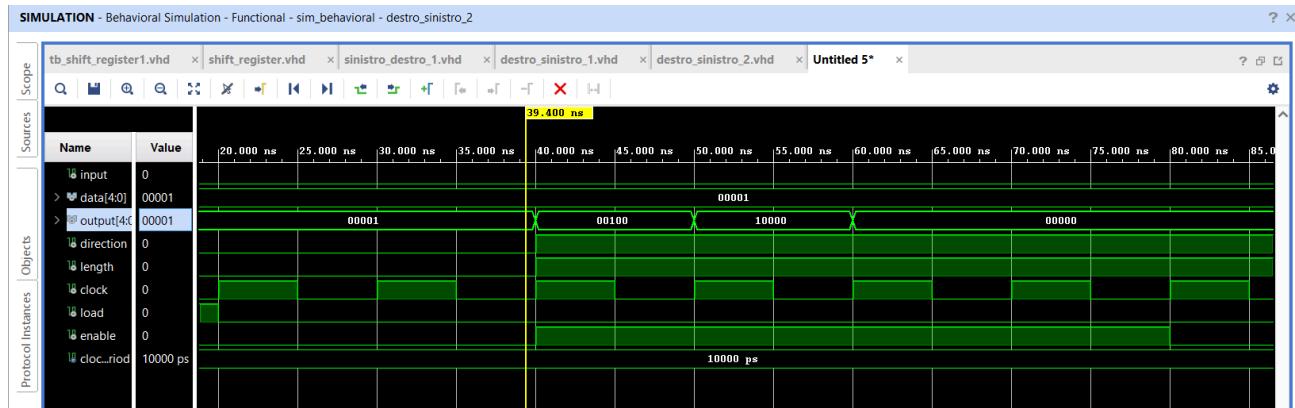
Di seguito i risultati ottenuti, provando varie modalità di funzionamento:



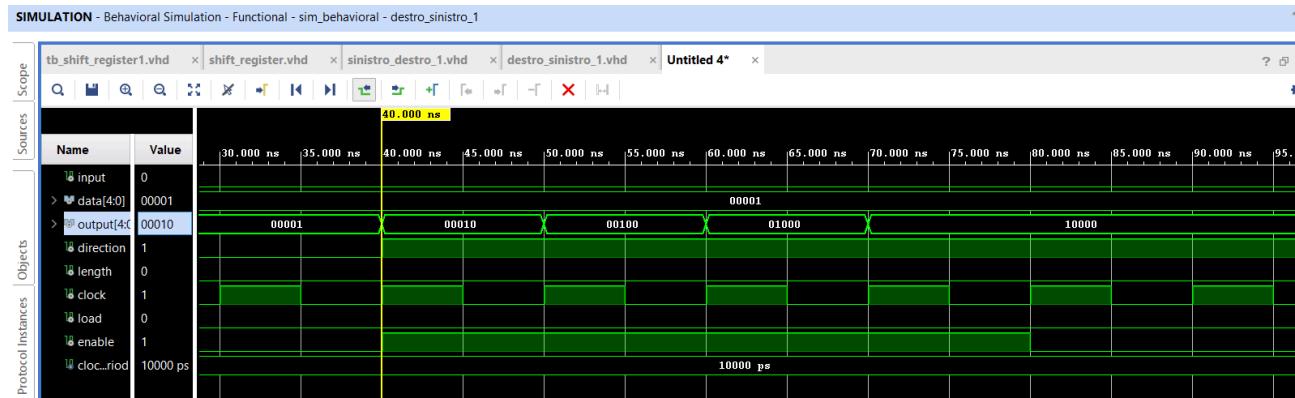
1 - Shifting a destra di una posizione



2 - Shifting a destra di due posizioni



3 - Shifting a sinistra di due posizioni



4 - Shifting a sinistra di una posizione

Esercizio 5: Cronometro

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

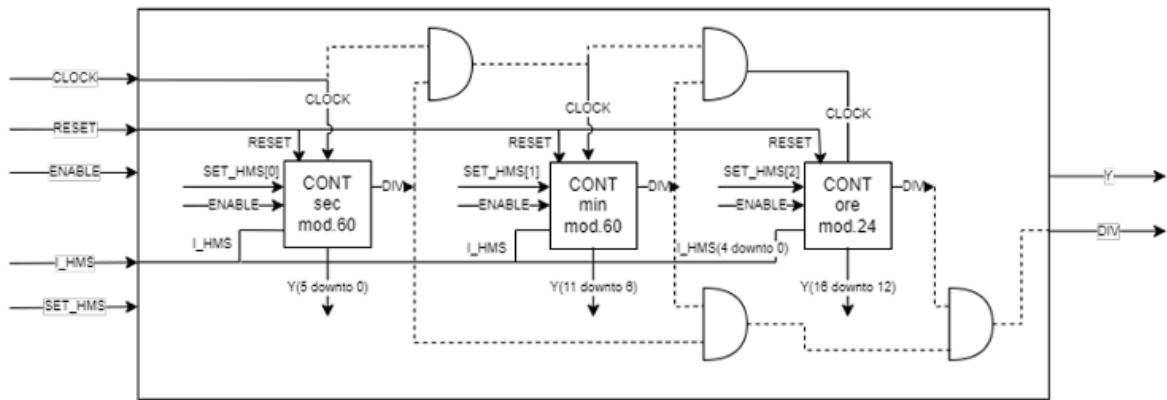
Progetto e architettura

Per poter realizzare il cronometro è stato utilizzato un solo componente: il contatore generico. E' stato implementato tramite un approccio comportamentale e la sua implementazione è disponibile nell'appendice a fine elaborato, tramite esso è stato possibile implementare i 3 tipi di contatori che formano il cronometro:

- Un contatore modulo 60 per i secondi
- Un contatore modulo 60 per i minuti
- Un contatore modulo 24 per le ore

Una volta individuate le componenti del cronometro abbiamo progettato il cronometro secondo uno schema parallelo, la cui figura è riportata successivamente. Si può evincere dalla figura che l'uscita di ogni contatore è posta in una AND con il segnale di clock che piloterà l'ingresso del contatore successivo. In pratica il contatore successivo "conta" quando è disponibile il segnale di clock e il contatore precedente ha

terminato il conteggio (ha alzato il segnale DIV).



Implementazione

Di seguito è riportato il codice VHDL del cronometro:

entity Cronometro is

```

    port(
        I_HMS : in STD_LOGIC_VECTOR(5 DOWNTO 0); -- input set ore, minuti o secondi
        EN_C : in STD_LOGIC; -- enable cronometro
        SET_HMS : in STD_LOGIC_VECTOR(2 DOWNTO 0); -- segnale set ore, minuti o secondi
        CLK_C : in STD_LOGIC; -- clock cronometro
        RST_C : in STD_LOGIC; -- reset cronometro
        DIV_C : out STD_LOGIC; -- uscita divisore (alta quando il contatore raggiunge il massimo)
        Y : out STD_LOGIC_VECTOR(16 DOWNTO 0) -- uscita corrente (ore, minuti, secondi)
    );

```

end Cronometro;

architecture Structural of Cronometro is

```

    signal exit_cont : STD_LOGIC_VECTOR(2 DOWNTO 0) := (others => '0'); -- uscite contatori
    signal exit_and : STD_LOGIC_VECTOR(1 DOWNTO 0) := (others => '0'); -- uscite porte and

```

component Cont_mod_N is

generic(N : in integer);

port(

```
I : in std_logic_vector((integer(ceil(log2(real(N))))-1) DOWNTO 0);
```

```
EN : in std_logic;
SET : in std_logic;
RST : in std_logic;
CLK : in std_logic;
DIV : out std_logic;
CONT : out std_logic_vector((integer(ceil(log2(real(N)))))-1 DOWNTO 0)
);
end component;
```

```
begin
cont_s : cont_mod_N
Generic map(
    N => 60
)
Port map(
    I => I_HMS,
    EN => EN_C,
    SET => SET_HMS(0),
    CLK => CLK_C,
    RST => RST_C,
    CONT => Y(5 DOWNTO 0),
    DIV => exit_cont(0)
);
```

```
exit_and(0) <= CLK_C and exit_cont(0);
```

```
cont_m : cont_mod_N
Generic map(
    N => 60
)
Port map(
    I => I_HMS,
    EN => EN_C,
```

```

SET => SET_HMS(1),
CLK => exit_and(0),
RST => RST_C,
CONT => Y(11 DOWNTO 6),
DIV => exit_cont(1)

);

exit_and(1) <= exit_and(0) and exit_cont(1);

cont_h : cont_mod_N
Generic map(
N => 24
)
Port map(
I => I_HMS(4 DOWNTO 0),
EN => EN_C,
SET => SET_HMS(2),
CLK => exit_and(1),
RST => RST_C,
CONT => Y(16 DOWNTO 12),
DIV => exit_cont(2)
);
DIV_C <= exit_cont(2) and (exit_cont(1) and exit_cont(0));

end Structural;

```

Simulazione

Nel testbench si è testato il funzionamento del cronometro settando l'orario di partenza a 00:59:00, cioè sono stati precaricati 59 minuti. Di seguito è indicato il codice VHDL corrispondente:

```

entity Cronometro_sim is
end Cronometro_sim;

```

```

architecture Behavioral of Cronometro_sim is

```

component Cronometro is

```
port(
    I_HMS : in STD_LOGIC_VECTOR(5 DOWNTO 0);
    EN_C : in STD_LOGIC;
    SET_HMS : in STD_LOGIC_VECTOR(2 DOWNTO 0);
    CLK_C : in STD_LOGIC;
    RST_C : in STD_LOGIC;
    DIV_C : out STD_LOGIC;
    Y : out STD_LOGIC_VECTOR(16 DOWNTO 0)
);
```

end component;

```
signal I_sim : STD_LOGIC_VECTOR(5 DOWNTO 0) := (others => '0');
```

```
signal EN_sim : STD_LOGIC := '0';
```

```
signal SET_sim: STD_LOGIC_VECTOR(2 DOWNTO 0) := (others => '0');
```

```
signal CLK_sim : STD_LOGIC := '0';
```

```
signal RST_sim : STD_LOGIC := '0';
```

```
signal DIV_sim : STD_LOGIC := '0';
```

```
signal Y_sim : STD_LOGIC_VECTOR(16 DOWNTO 0) := (others => '0');
```

```
constant CLK_PERIOD : time := 10 ns;
```

begin

uut: Cronometro

Port map(

```
    I_HMS => I_sim,
```

```
    EN_C => EN_sim,
```

```
    SET_HMS => SET_sim,
```

```
    CLK_C => CLK_sim,
```

```
    RST_C => RST_sim,
```

```
    DIV_C => DIV_sim,
```

```
    Y => Y_sim
```

```
);
```

```
clk_process: process
begin
    CLK_sim <= '0';
    wait for CLK_PERIOD/2;
    CLK_sim <= '1';
    wait for CLK_PERIOD/2;
end process clk_process;
```

```
-- stim_proc: process
-- begin
--     wait for 100 ns;
--     SET_sim <= "000";
--     RST_sim <= '1';
--     wait for 10 ns;
--     RST_sim <= '0';
--     wait for 10 ns;
--     EN_sim <= '1';
--     wait for 10ns;
--     wait;
-- end process stim_proc;
```

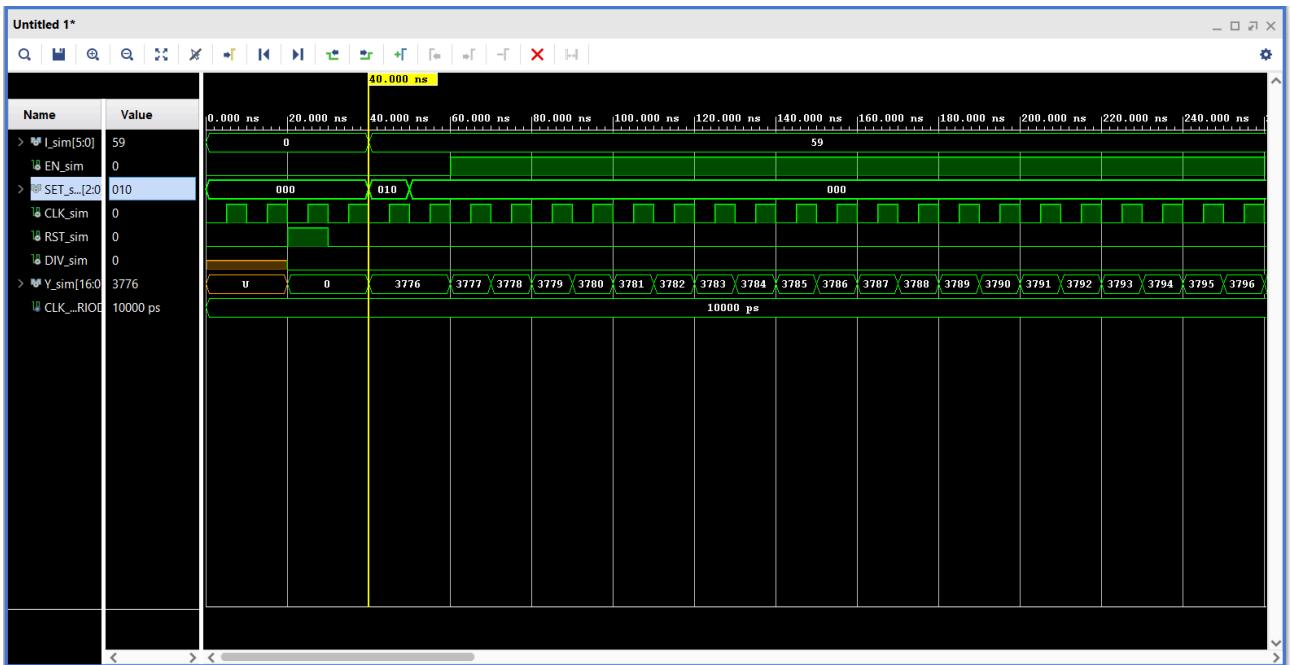
```
stim_proc: process
```

```
begin
    wait for 20ns;
    RST_sim <= '1';
    wait for 10ns;
    RST_sim <= '0';
    wait for 10ns;
    I_sim <= "111011";
    SET_sim <= "010";
```

```
    wait for 10ns;  
    SET_sim <= "000";  
    wait for 10ns;  
    EN_sim <= '1';  
    wait for 10ns;  
    wait;  
end process stim_proc;
```

```
-- stim_proc: process  
-- begin  
--   wait for 20ns;  
--   RST_sim <= '1';  
--   wait for 10ns;  
--   RST_sim <= '0';  
--   wait for 10ns;  
--   I_sim <= "111011";  
--   SET_sim <= "010";  
--   wait for 10ns;  
--   I_sim <= "010111";  
--   SET_sim <= "100";  
--   wait for 10ns;  
--   SET_sim <= "000";  
--   wait for 10ns;  
--   EN_sim <= '1';  
--   wait for 10ns;  
--   wait;  
-- end process stim_proc;
```

```
end Behavioral;
```



Si può notare che dal momento in cui il valore 59 è caricato nei minuti l'uscita del cronometro assume il valore “3776” il quale è una rappresentazione in decimale della stringa “00000_111011_000000”, i tratti sono stati aggiunti per separare ore, minuti e secondi.

Esercizio 5.2: implementazione su board

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

Progetto e architettura

Per la sintesi sulla board del cronometro, l’architettura utilizzata come base è stata quella dell’esercizio precedente (5.1). Abbiamo adottato un approccio strutturale, creando una nuova entity “Cronometro_board”, composta dai seguenti elementi:

- Due bottoni, per il set dell’orario e per il reset
- Un divisore di frequenza: permette di ottenere, a partire dal clock della board, un clock con frequenza pari ad 1 Hz, utilizzato dal cronometro
- Un’unità di controllo (CU)
- Il cronometro
- Tre convertitori Conv_Dec_Bin: ognuno rispettivamente per secondi, minuti ed ore. Permette di realizzare la conversione del segnale di 17 bit (ottenuto dal cronometro), in un segnale che sia compatibile con l’ingresso del display a sette segmenti. Infatti, quest’ultimo richiede la codifica di ogni cifra su 4 bit
- Un display a sette segmenti, utilizzato per visualizzare l’output.

Per l’acquisizione degli ingressi vengono utilizzati 6 switch e due bottoni sulla board. Il componente “ButtonDebouncer” (vedi appendice) è stato necessario per una corretta acquisizione della pressione dei pulsanti.

(inserire immagine schema a blocchi)

Da notare che le cifre da visualizzare sul display sono 6 e ciò ci porta ad utilizzare i primi 24 bit meno significativi dell'ingresso da 32 bit del valore da visualizzare sul display, perchè come detto prima il display a sette segmenti necessita di una codifica su 4 bit (6 cifre su 4 bit => $6 \times 4 = 24$).

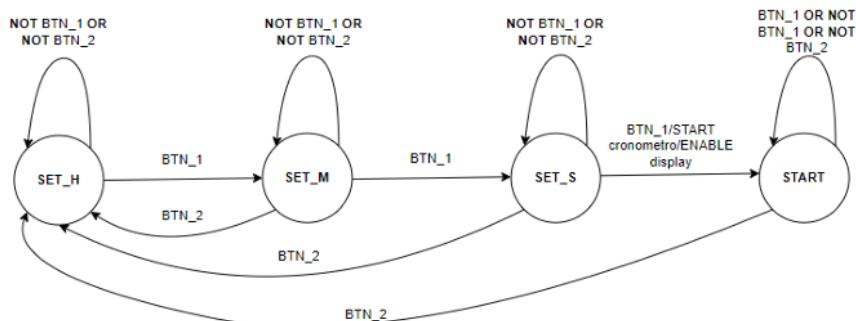
L'unità di controllo riceve in ingresso 4 segnali:

1. gli switch (board)
2. il segnale "pulito" di set
3. il segnale "pulito" di reset
4. il segnale di clock (board)

e ne fornisce in uscita 5:

1. il segnale di enable per il display a sette segmenti
2. il segnale di reset del cronometro
3. il segnale di enable del cronometro
4. il segnale di set del cronometro
5. il segnale di input del cronometro

Per descriverne il comportamento, è stato realizzato il seguente automa a stati finiti:



SET_H: stato iniziale in cui l'utente può inserire l'ora di partenza su 5 bit attraverso gli switch (acquisito alla pressione del tasto **BTN_1**, o BTNL sulla board).

SET_M: stato in cui l'utente può inserire i minuti di partenza su 6 bit attraverso gli switch (acquisiti alla pressione del tasto **BTN_1**, o BTNL sulla board).

SET_S: stato in cui l'utente può inserire i secondi di partenza su 6 bit attraverso gli switch (acquisiti alla pressione del tasto **BTN_1**, o BTNL sulla board).

START: stato in cui viene data l'abilitazione al cronometro e al display, in modo tale che l'orario possa essere visualizzato sulla board.

N.B. in qualsiasi stato ci troviamo, se viene premuto il bottone **BTN_2** (BTNR sulla board), si ritorna allo stato iniziale, ovvero **SET_H**.

Implementazione

Di seguito sono riportati i codici dell'entità del cronometro sulla board e della rispettiva unità di controllo.

entity Cronometro_board is

port(

CLK : in STD_LOGIC;

```

    BTN_1 : in STD_LOGIC;
    BTN_2 : in STD_LOGIC;
    INPUT_SWITCH : in STD_LOGIC_VECTOR(5 DOWNTO 0);
    ANODES : out STD_LOGIC_VECTOR(7 DOWNTO 0);
    CATHODES : out STD_LOGIC_VECTOR(7 DOWNTO 0)
);

end Cronometro_board;

```

architecture Structural of Cronometro_board is

```

signal btn_1_out_tmp : STD_LOGIC := '0';
signal btn_2_out_tmp : STD_LOGIC := '0';
signal rst_cr_tmp : STD_LOGIC := '0';
signal enable_cr_tmp : STD_LOGIC := '0';
signal set_hms_cr_tmp : STD_LOGIC_VECTOR(2 DOWNTO 0) := (others => '0');
signal input_hms_cr_tmp : STD_LOGIC_VECTOR(5 DOWNTO 0) := (others => '0');
signal clock_1Hz_tmp : STD_LOGIC := '0';
signal enable_display_tmp : STD_LOGIC_VECTOR(7 DOWNTO 0) := (others => '0');
signal out_tmp : STD_LOGIC_VECTOR(16 DOWNTO 0) := (others => '0');
signal value_tmp : STD_LOGIC_VECTOR(31 DOWNTO 0) := (others => '0');

```

component Cronometro

```

port(
    I_CRN : in std_logic_vector(5 DOWNTO 0);
    EN_CRN : in std_logic;
    SET_CRN : in std_logic_vector(2 DOWNTO 0);
    RST_CRN : in std_logic;
    CLK_CRN : in std_logic;
    DIV_CRN : out std_logic;
    OUTPUT : out std_logic_vector(16 DOWNTO 0)
);

```

```
end component;

component ControlUnit
port(
    CLK_CU: in STD_LOGIC;
    BTN_1 : in STD_LOGIC;
    BTN_2 : in STD_LOGIC;
    SWITCH_IN : in STD_LOGIC_VECTOR(5 DOWNTO 0);
    RST_CR: out STD_LOGIC;
    SET_HMS_CR : out STD_LOGIC_VECTOR(2 DOWNTO 0);
    INPUT_CR : out STD_LOGIC_VECTOR(5 DOWNTO 0);
    EN_DISPLAY : out STD_LOGIC_VECTOR(7 DOWNTO 0);
    EN_CR : out STD_LOGIC
);
end component;
```

```
component display_seven_segments
generic(
    CLKIN_freq : integer := 100000000;
    CLKOUT_freq : integer := 500);
port(
    CLK : in STD_LOGIC;
    RST : in STD_LOGIC;
    VALUE : in STD_LOGIC_VECTOR (31 DOWNTO 0);
    ENABLE : in STD_LOGIC_VECTOR (7 DOWNTO 0);
    DOTS : in STD_LOGIC_VECTOR (7 DOWNTO 0);
    ANODES : out STD_LOGIC_VECTOR (7 DOWNTO 0);
    CATHODES : out STD_LOGIC_VECTOR (7 DOWNTO 0)
);
end component;
```

```
component ButtonDebouncer
```

```

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

```

```

component Conv_Dec_Bin
generic(N : integer := 6; -- numero di bit del numero binario di input
       M : integer := 2 -- numero di cifre decimali risultanti
);
port(
    CLK : in STD_LOGIC;
    input : in STD_LOGIC_VECTOR(N-1 DOWNTO 0);
    output : out STD_LOGIC_VECTOR(4*M - 1 DOWNTO 0)
);
end component;

```

```

component clock_divider
generic(
    clock_frequency_in : in integer := 100000000;
    clock_frequency_out : in integer := 1);
port(
    CLK_IN : in STD_LOGIC;
    CLK_OUT : out STD_LOGIC
);
end component;

```

```
begin
```

```
CRONOM : Cronometro
```

```
Port map(  
    I_CRN => input_hms_cr_tmp,  
    EN_CRN => enable_cr_tmp,  
    SET_CRN => set_hms_cr_tmp,  
    CLK_CRN => clock_1Hz_tmp,  
    RST_CRN => rst_cr_tmp,  
    OUTPUT => out_tmp  
);
```

```
CU : ControlUnit
```

```
Port map(  
    CLK CU => CLK,  
    BTN_1 => btn_1_out_tmp,  
    BTN_2 => btn_2_out_tmp,  
    SWITCH_IN => INPUT_SWITCH,  
    RST_CR => rst_cr_tmp,  
    SET_HMS_CR => set_hms_cr_tmp,  
    INPUT_CR => input_hms_cr_tmp,  
    EN_DISPLAY => enable_display_tmp,  
    EN_CR => enable_cr_tmp  
);
```

```
BD1 : ButtonDebouncer
```

```
Port map(  
    CLK => CLK,  
    RST => rst_cr_tmp,  
    BTN => BTN_1,  
    CLEARED_BTN => btn_1_out_tmp
```

);

BD2 : ButtonDebouncer

Port map(

CLK => CLK,

RST => rst_cr_tmp,

BTN => BTN_2,

CLEARED_BTN => btn_2_out_tmp

);

DIS : display_seven_segments

Port map(

CLK => CLK,

RST => rst_cr_tmp,

VALUE => value_tmp,

ENABLE => enable_display_tmp,

DOTS => "00010100",

ANODES => ANODES,

CATHODES => CATHODES

);

DTB_S : Conv_Dec_Bin

Generic map(6,2)

Port map(

CLK => CLK,

input => out_tmp(5 DOWNTO 0),

output => value_tmp(7 DOWNTO 0)

);

DTB_M : Conv_Dec_Bin

Generic map(6,2)

Port map(

```

CLK => CLK,
input => out_tmp(11 DOWNTO 6),
output => value_tmp(15 DOWNTO 8)
);

DTB_H : Conv_Dec_Bin
Generic map(5,2)
Port map(
CLK => CLK,
input => out_tmp(16 DOWNTO 12),
output => value_tmp(23 DOWNTO 16)
);

CLK_DIV_FREQ : clock_divider
Port map(
CLK_IN => CLK,
CLK_OUT => clock_1Hz_tmp
);

end Structural;

-----
entity ControlUnit is
port(
CLK_CU: in STD_LOGIC;
BTN_1 : in STD_LOGIC;
BTN_2 : in STD_LOGIC;
SWITCH_IN : in STD_LOGIC_VECTOR(5 DOWNTO 0);
RST_CR: out STD_LOGIC;
SET_HMS_CR : out STD_LOGIC_VECTOR(2 DOWNTO 0);
INPUT_CR : out STD_LOGIC_VECTOR(5 DOWNTO 0);
EN_DISPLAY : out STD_LOGIC_VECTOR(7 DOWNTO 0);
EN_CR : out STD_LOGIC

```

```

);

end ControlUnit;

architecture Behavioral of ControlUnit is

type stato is (SET_H, SET_M, SET_S, START);

signal stato_corr : stato := SET_H;
signal input_cr_tmp : STD_LOGIC_VECTOR(5 DOWNTO 0) := (others => '0');
signal set_hms_cr_tmp : STD_LOGIC_VECTOR(2 DOWNTO 0) := (others => '0');
signal enable_cr_tmp : STD_LOGIC := '0';
signal enable_display_tmp : STD_LOGIC_VECTOR(7 DOWNTO 0) := (others => '0');
signal rst_cr_tmp : STD_LOGIC := '0';

begin

proc : process(CLK CU)
begin
  if(rising_edge(CLK CU)) then
    if(BTN_2 = '1') then
      set_hms_cr_tmp <= "000";
      stato_corr <= SET_H;
      rst_cr_tmp <= '1';
      enable_display_tmp <= (others => '0');
    else
      rst_cr_tmp <= '0';
    end if;
    case stato_corr is
      when SET_H =>
        if(BTN_1 = '1') then
          input_cr_tmp <= SWITCH_IN;
          set_hms_cr_tmp <= "100";
          stato_corr <= SET_M;
        end if;
    end case;
  end if;
end process;
end Behavioral;

```

```

when SET_M =>

    set_hms_cr_tmp <= "000";
    if(BTN_1 = '1') then
        input_cr_tmp <= SWITCH_IN;
        set_hms_cr_tmp <= "010";
        stato_corr <= SET_S;
    end if;

when SET_S =>

    set_hms_cr_tmp <= "000";
    if(BTN_1 = '1') then
        input_cr_tmp <= SWITCH_IN;
        set_hms_cr_tmp <= "001";
        stato_corr <= START;
    end if;

when START =>

    set_hms_cr_tmp <= "000";
    enable_cr_tmp <= '1';
    enable_display_tmp <= "00111111";
    stato_corr <=START;
end case;
end if;
end if;
end process;

```

```

INPUT_CR <= input_cr_tmp;
SET_HMS_CR <= set_hms_cr_tmp;
EN_CR <= enable_cr_tmp;
EN_DISPLAY <= enable_display_tmp;
RST_CR <= rst_cr_tmp;

```

```
end Behavioral;
```

Sintesi su board di sviluppo

File di constraints utilizzato nel progetto:

```
## Clock signal

set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { CLK }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK}];

##Switches

set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { INPUT_SWITCH[0] }];
#IO_L24N_T3_RSO_15 Sch=sw[0]

set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { INPUT_SWITCH[1] }];
#IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]

set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { INPUT_SWITCH[2] }];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]

set_property -dict { PACKAGE_PIN R15 IOSTANDARD LVCMOS33 } [get_ports { INPUT_SWITCH[3] }];
#IO_L13N_T2_MRCC_14 Sch=sw[3]

set_property -dict { PACKAGE_PIN R17 IOSTANDARD LVCMOS33 } [get_ports { INPUT_SWITCH[4] }];
#IO_L12N_T1_MRCC_14 Sch=sw[4]

set_property -dict { PACKAGE_PIN T18 IOSTANDARD LVCMOS33 } [get_ports { INPUT_SWITCH[5] }];
#IO_L7N_T1_D10_14 Sch=sw[5]

#set_property -dict { PACKAGE_PIN U18 IOSTANDARD LVCMOS33 } [get_ports { SW[6] }];
#IO_L17N_T2_A13_D29_14 Sch=sw[6]

#set_property -dict { PACKAGE_PIN R13 IOSTANDARD LVCMOS33 } [get_ports { SW[7] }];
#IO_L5N_T0_D07_14 Sch=sw[7]

#set_property -dict { PACKAGE_PIN T8 IOSTANDARD LVCMOS18 } [get_ports { SW[8] }]; #IO_L24N_T3_34
Sch=sw[8]

#set_property -dict { PACKAGE_PIN U8 IOSTANDARD LVCMOS18 } [get_ports { SW[9] }]; #IO_25_34
Sch=sw[9]

#set_property -dict { PACKAGE_PIN R16 IOSTANDARD LVCMOS33 } [get_ports { SW[10] }];
#IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]

#set_property -dict { PACKAGE_PIN T13 IOSTANDARD LVCMOS33 } [get_ports { SW[11] }];
#IO_L23P_T3_A03_D19_14 Sch=sw[11]
```

```

#set_property -dict { PACKAGE_PIN H6  IOSTANDARD LVCMOS33 } [get_ports { SW[12] }]; #IO_L24P_T3_35
Sch=sw[12]

#set_property -dict { PACKAGE_PIN U12  IOSTANDARD LVCMOS33 } [get_ports { SW[13] }];
#IO_L20P_T3_A08_D24_14 Sch=sw[13]

#set_property -dict { PACKAGE_PIN U11  IOSTANDARD LVCMOS33 } [get_ports { SW[14] }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]

#set_property -dict { PACKAGE_PIN V10  IOSTANDARD LVCMOS33 } [get_ports { SW[15] }];
#IO_L21P_T3_DQS_14 Sch=sw[15]

##7 segment display

set_property -dict { PACKAGE_PIN T10  IOSTANDARD LVCMOS33 } [get_ports { CATHODES[0] }];
#IO_L24N_T3_A00_D16_14 Sch=ca

set_property -dict { PACKAGE_PIN R10  IOSTANDARD LVCMOS33 } [get_ports { CATHODES[1] }]; #IO_25_14
Sch=cb

set_property -dict { PACKAGE_PIN K16  IOSTANDARD LVCMOS33 } [get_ports { CATHODES[2] }]; #IO_25_15
Sch=cc

set_property -dict { PACKAGE_PIN K13  IOSTANDARD LVCMOS33 } [get_ports { CATHODES[3] }];
#IO_L17P_T2_A26_15 Sch=cd

set_property -dict { PACKAGE_PIN P15  IOSTANDARD LVCMOS33 } [get_ports { CATHODES[4] }];
#IO_L13P_T2_MRCC_14 Sch=ce

set_property -dict { PACKAGE_PIN T11  IOSTANDARD LVCMOS33 } [get_ports { CATHODES[5] }];
#IO_L19P_T3_A10_D26_14 Sch=cf

set_property -dict { PACKAGE_PIN L18  IOSTANDARD LVCMOS33 } [get_ports { CATHODES[6] }];
#IO_L4P_T0_D04_14 Sch=cg

set_property -dict { PACKAGE_PIN H15  IOSTANDARD LVCMOS33 } [get_ports { CATHODES[7] }];
#IO_L19N_T3_A21_VREF_15 Sch=dp

set_property -dict { PACKAGE_PIN J17  IOSTANDARD LVCMOS33 } [get_ports { ANODES[0] }];
#IO_L23P_T3_FOE_B_15 Sch=an[0]

set_property -dict { PACKAGE_PIN J18  IOSTANDARD LVCMOS33 } [get_ports { ANODES[1] }];
#IO_L23N_T3_FWE_B_15 Sch=an[1]

set_property -dict { PACKAGE_PIN T9   IOSTANDARD LVCMOS33 } [get_ports { ANODES[2] }];
#IO_L24P_T3_A01_D17_14 Sch=an[2]

set_property -dict { PACKAGE_PIN J14  IOSTANDARD LVCMOS33 } [get_ports { ANODES[3] }];
#IO_L19P_T3_A22_15 Sch=an[3]

set_property -dict { PACKAGE_PIN P14  IOSTANDARD LVCMOS33 } [get_ports { ANODES[4] }];
#IO_L8N_T1_D12_14 Sch=an[4]

set_property -dict { PACKAGE_PIN T14  IOSTANDARD LVCMOS33 } [get_ports { ANODES[5] }];
#IO_L14P_T2_SRCC_14 Sch=an[5]

```

```

set_property -dict { PACKAGE_PIN K2  IOSTANDARD LVCMOS33 } [get_ports { ANODES[6] }];
#IO_L23P_T3_35 Sch=an[6]

set_property -dict { PACKAGE_PIN U13  IOSTANDARD LVCMOS33 } [get_ports { ANODES[7] }];
#IO_L23N_T3_A02_D18_14 Sch=an[7]

##Buttons

#set_property -dict { PACKAGE_PIN C12  IOSTANDARD LVCMOS33 } [get_ports { CPU_RESETN }];
#IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

#set_property -dict { PACKAGE_PIN N17  IOSTANDARD LVCMOS33 } [get_ports { BTNC }];
#IO_L9P_T1_DQS_14 Sch=btnc

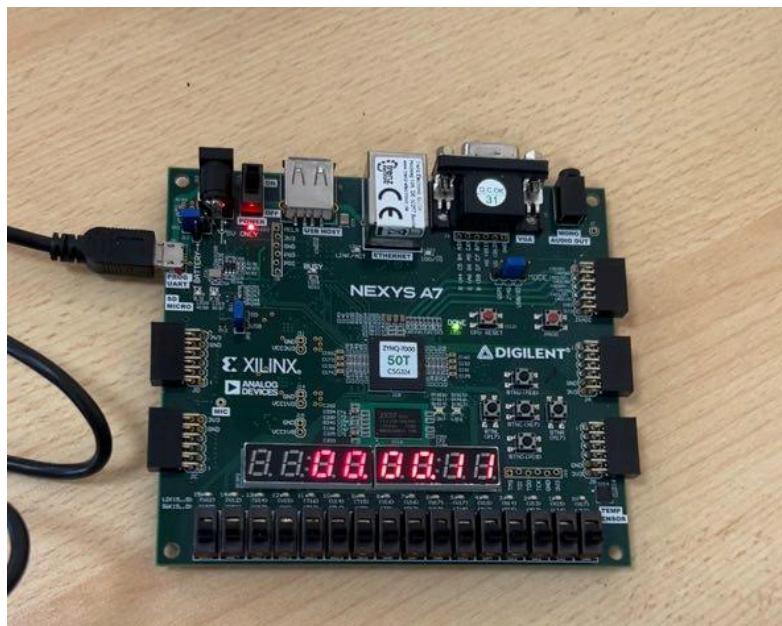
#set_property -dict { PACKAGE_PIN M18  IOSTANDARD LVCMOS33 } [get_ports { BTNU }];
#IO_L4N_T0_D05_14 Sch=btnu

set_property -dict { PACKAGE_PIN P17  IOSTANDARD LVCMOS33 } [get_ports { BTN_1 }];
#IO_L12P_T1_MRCC_14 Sch=btnl

set_property -dict { PACKAGE_PIN M17  IOSTANDARD LVCMOS33 } [get_ports { BTN_2 }];
#IO_L10N_T1_D15_14 Sch=btnr

#set_property -dict { PACKAGE_PIN P18  IOSTANDARD LVCMOS33 } [get_ports { BTND }];
#IO_L9N_T1_DQS_D13_14 Sch=btnd

```



Esercizio 6: Sistema di lettura-elaborazione-scrittura PO_PC

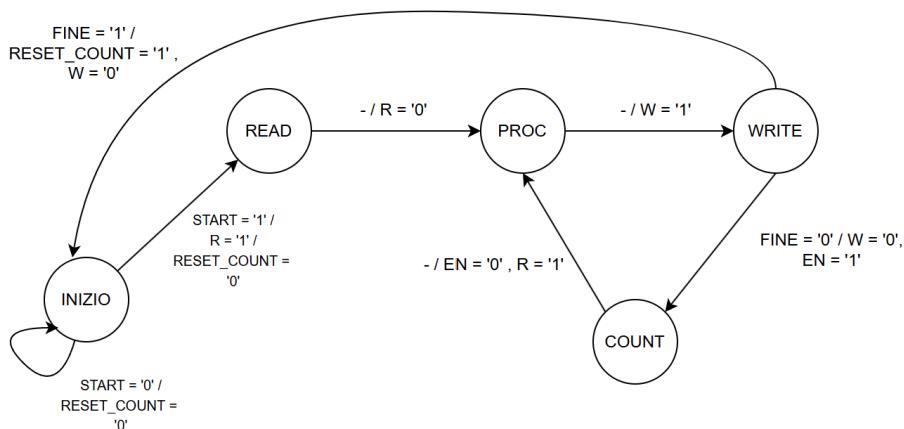
Progettare, implementare in VHDL e verificare mediante simulazione un sistema dotato di una memoria ROM di N locazioni da 8 bit ciascuna, una macchina combinatoria M in grado di trasformare (secondo una funzione a scelta dello studente) la stringa di 8 bit letta dalla ROM in una stringa di 4 bit, e una memoria MEM di N locazioni che memorizza la stringa in output da M.

Il sistema si avvia in corrispondenza di un segnale di START che viene fornito esternamente. Una volta avviato, tramite un'apposita unità di controllo che gestisce la tempificazione del sistema, viene scandita una locazione alla volta della ROM e viene scritta la corrispondente locazione di MEM. Gli indirizzi di memoria sono forniti da un contatore. Le memorie ROM e MEM hanno rispettivamente un read e un write sincrono.

Progetto e architettura

Il sistema che abbiamo progettato riceve in ingresso un segnale di START e un segnale di clock di sistema CLK_sis, ed è costituito dalle seguenti componenti principali:

- **COUNTER**: un contatore a modulo N (vedi Appendice) che scandisce gli indirizzi della ROM e della MEMORIA. È stato progettato in modo che, al termine del conteggio, si azzeri automaticamente e generi un segnale di fine conteggio END_COUNT;
- **ROM**: una memoria di sola lettura composta da N locazioni, ciascuna di 8 bit (vedi Appendice);
- **MC**: una macchina combinatoria che prende in ingresso i dati letti dalla ROM, esegue operazioni logiche di OR su coppie di bit e concatena i risultati ottenuti;
- **MEM**: una memoria che riceve l'output della MC e lo memorizza all'indirizzo specificato dal COUNTER (vedi Appendice);
- **CU**: la control unit, realizzata come un ASF di Mealy (schema riportato di seguito), che gestisce il processo di lettura dalla ROM, l'elaborazione tramite la MC e la successiva scrittura nella MEM.



La CU gestisce il flusso del sistema attraverso i seguenti stati:

- **INIZIO**: rappresenta lo stato iniziale, in cui viene azzerato il contatore impostando il segnale RESET_CONT a 0. Se il segnale di avvio START è attivo (alto), si passa allo stato READ, attivando il segnale di lettura R.
- **READ**: una volta effettuata la lettura, il segnale R viene disattivato e il sistema avanza allo stato PROC.

- **PROC**: in questa fase, dopo aver completato l'elaborazione nella MC, si abilita la scrittura in memoria portando il segnale W a 1, per poi procedere allo stato WRITE.
- **WRITE**: si disattiva la scrittura (W basso). Se il segnale di fine conteggio FINE è attivo, il sistema ritorna allo stato INIZIO, resettando il contatore (RESET_CONT alto); altrimenti si abilita il segnale EN e si passa allo stato COUNT.
- **COUNT**: il contatore viene disabilitato, si attiva nuovamente il segnale di lettura R, e il sistema torna allo stato READ.

Implementazione

Di seguito sono riportati gli scripts VHDL delle componenti realizzate e collegate opportunamente in un'entità finale chiamata: *sistema*.

```
entity macchina_combinatoria is
  Port(
    input_m : in std_logic_vector(7 downto 0);
    output_m : out std_logic_vector(3 downto 0)
  );
end macchina_combinatoria;
```

```
architecture DataFlow of macchina_combinatoria is
```

```
begin
  output_m <= (input_m(7) OR input_m(0))& (input_m(6) OR input_m(1)) & (input_m(5) OR input_m(2)) &
  (input_m(4) OR input_m(3));
end DataFlow;
```

```
entity control_unit is
```

```
  Generic( len_add : positive := 4);
```

```
  Port(
```

```
    CLK_cu : in std_logic;
```

```
    START : in std_logic;
```

```
    FINE : in std_logic;
```

```
    R : out std_logic;
```

```

W : out std_logic;
RESET_CONT : out std_logic;
EN : out std_logic
);
end control_unit;

architecture Behavioral of control_unit is
constant N : positive := 2**len_add;

type STATO is (INIZIO, READ, WRITE, PROC, COUNT);
signal stato_corrente : STATO := INIZIO;
signal stato_prossimo : STATO;

begin
comb : process(START, FINE, stato_corrente)
begin
  case stato_corrente is
    when INIZIO =>
      RESET_CONT <= '0';
      if START = '1' then
        stato_prossimo <= READ;
        R <= '1';
      end if;
    when COUNT =>
      EN <= '0';
      R <= '1';
      stato_prossimo <= READ;
    when READ =>
      R <= '0';
  end case;
end process;
end;

```

```

stato_prossimo <= PROC;
when PROC =>
    W <= '1';
    stato_prossimo <= WRITE;
when WRITE =>
    W <= '0';
    if FINE = '1' then
        stato_prossimo <= INIZIO;
        RESET_CONT <= '1';
    else
        EN <= '1';
        stato_prossimo <= COUNT;
    end if;

end case;
end process comb;

```

```

seq: process(CLK_cu)
begin
    if rising_edge(CLK_cu) then
        stato_corrente <= stato_prossimo;
    end if;
end process seq;

```

```
end Behavioral;
```

```

entity sistema is
    generic( len_add : positive := 4);
    Port (
        CLK_sis : in std_logic;
        START : in std_logic;
        Y : out std_logic_vector(3 downto 0)

```

```
 );
end sistema;
```

architecture Behavioral of sistema is

COMPONENT memoria is

```
Generic(
    len_add : positive := 4
);
Port(
    CLK_mem : in std_logic;
    write : in std_logic;
    address : in std_logic_vector (len_add-1 downto 0);
    inp_val : in std_logic_vector(3 downto 0);
    out_val : out std_logic_vector(3 downto 0)
);
```

END COMPONENT;

COMPONENT ROM_N_8 IS

```
GENERIC(
    len_add : positive := 4
);
PORT (
    CLK_rom : in std_logic;
    address : in std_logic_vector(len_add - 1 downto 0);
    read : in std_logic;
    dout : out std_logic_vector(7 downto 0)
);
END COMPONENT;
```

component contatore is

```

generic( N: positive := 16);

port(
    clock : in std_logic;
    reset : in std_logic;
    load : in std_logic_vector((integer(ceil(log2(real(N)))))-1 downto 0); --IN QUESTO CASO NON SERVE
    enable_contatore : in std_logic; -- PER ABILITARE IL CONTATORE
    enable_caricamento: in std_logic; --NON SERVE
    cont : out std_logic_vector((integer(ceil(log2(real(N)))))-1 downto 0);
    div : out std_logic --variabile di uscita alta solo quando il contatore raggiunge il massimo
);
end component;

```

component macchina_combinatoria is

```

Port(
    input_m : in std_logic_vector(7 downto 0);
    output_m : out std_logic_vector(3 downto 0)
);
end component;

```

component control_unit is

```
Generic( len_add : positive := 4);
```

```

Port(
    CLK_cu : in std_logic;
    START : in std_logic;
    FINE : in std_logic;
    R : out std_logic;
    W : out std_logic;
    RESET_CONT : out std_logic;
    EN : out std_logic
);

```

end component;

```

signal temp_R : std_logic;
signal temp_W : std_logic;
signal temp_RESET_CONT : std_logic;
signal temp_EN : std_logic;
signal temp_END_COUNT : std_logic;
signal temp_address : std_logic_vector(len_add -1 downto 0);
signal temp_input_mc : std_logic_vector(7 downto 0);
signal temp_out_mc : std_logic_vector(3 downto 0);

begin

cu: control_unit
PORT MAP(
    CLK_cu => CLK_sis,
    START => START,
    FINE => temp_END_COUNT,
    R => temp_R,
    W => temp_W,
    RESET_CONT => temp_RESET_CONT,
    EN => temp_EN
);

counter: contatore
port map(
    clock => CLK_sis,
    reset =>temp_RESET_CONT,
    load => (others => '0'), --IN QUESTO CASO NON SERVE
    enable_contatore => temp_EN, -- PER ABILITARE IL CONTATORE
    enable_caricamento => '0', --NON SERVE
    cont => temp_address ,
    div => temp_END_COUNT--variabile di uscita alta solo quando il contatore raggiunge il massimo

```

```
);
```

```
ROM : ROM_N_8
```

```
PORT MAP(
```

```
    CLK_rom => CLK_sis,  
    address => temp_address,  
    read => temp_R,  
    dout => temp_input_mc
```

```
);
```

```
MC : macchina_combinatoria
```

```
Port map(
```

```
    input_m => temp_input_mc,  
    output_m => temp_out_mc  
);
```

```
MEM : memoria
```

```
Port map(
```

```
    CLK_mem => CLK_sis,  
    write => temp_W,  
    address => temp_address,  
    inp_val => temp_out_mc,  
    out_val => Y
```

```
);
```

```
end Behavioral;
```

Simulazione

Per la simulazione abbiamo scritto il testbench in cui abbiamo implementato il processo di simulazione del clock e abbiamo alzato il segnale di START. Nell'istanziare il sistema abbiamo considerato il generic `len_add` pari a 4 specificando quindi che sia ROM che MEM hanno $N = 2^{len_add}$ locazioni.

```
entity sistema_tb is
```

```
end;
```

```
architecture bench of sistema_tb is
```

```
component sistema
```

```
generic( len_add : positive := 4);
```

```
Port (
```

```
    CLK_sis : in std_logic;
```

```
    START : in std_logic;
```

```
    Y : out std_logic_vector(3 downto 0)
```

```
);
```

```
end component;
```

```
signal CLK_sis: std_logic;
```

```
signal START: std_logic;
```

```
signal Y: std_logic_vector(3 downto 0) ;
```

```
constant CLK_period : time := 5 ns;
```

```
begin
```

```
    uut: sistema generic map ( len_add => 4 )
```

```
        port map ( CLK_sis => CLK_sis,
```

```
                    START => START,
```

```
                    Y => Y );
```

```
    CLK_process :process
```

```
        begin
```

```
            CLK_sis <= '0';
```

```
            wait for CLK_period/2;
```

```
            CLK_sis <= '1';
```

```
            wait for CLK_period/2;
```

```
        end process;
```

```
stimulus: process
```

```
begin
```

```
    wait for 100ns;
```

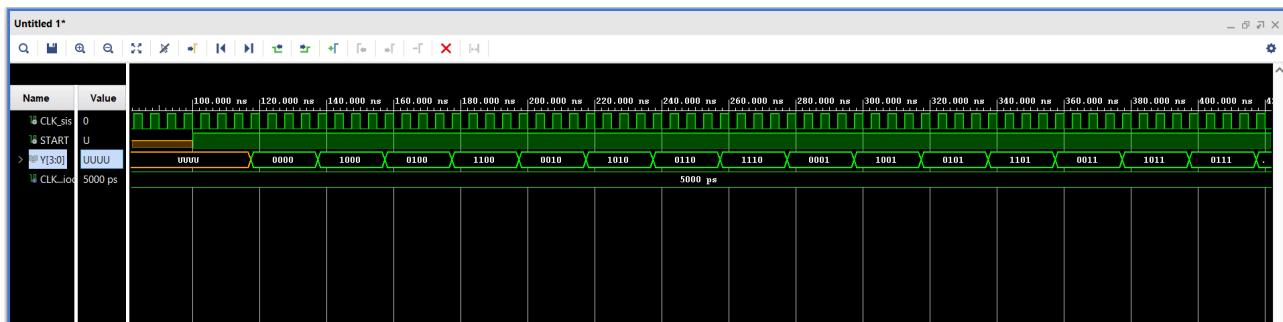
```
    START <= '1';
```

```
    wait;
```

```
end process;
```

```
end;
```

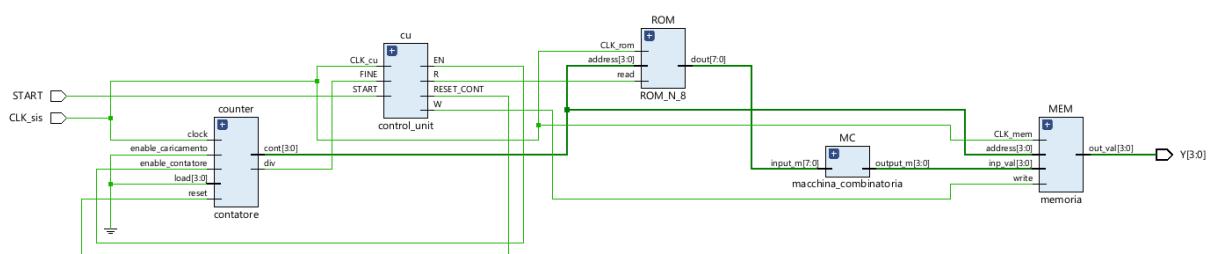
I risultati ottenuti dalla simulazione del testbench sono i seguenti:



Esercizio 6.2: implementazione su board

Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando due pulsanti per i segnali di read e reset rispettivamente e i led per la visualizzazione delle uscite della macchina istante per istante.

Per adeguare il progetto alle specifiche richieste per la sintesi su board, si sono rese necessarie alcune modifiche rispetto alla versione precedente del *sistema* e della *control_unit*.



Nella nuova versione, il segnale R è stato trasformato in un input, collegato a un pulsante attraverso il *sistema*. È stato inoltre introdotto un nuovo segnale di ingresso, RESET_cu, che consente di azzerare il contatore premendo un pulsante dedicato sulla board.

Questi aggiornamenti hanno reso necessario modificare il processo combinatorio, rendendolo sensibile anche ai segnali R e RESET_cu, oltre a richiedere piccoli adattamenti alla logica.

Di seguito è riportato il codice aggiornato.

```
entity control_unit_board is
```

```
    Generic( len_add : positive := 4);
```

```
    Port(
```

```
        CLK_cu : in std_logic;
```

```
        START : in std_logic;
```

```
        FINE : in std_logic;
```

```
        RESET_cu : in std_logic;
```

```
        R : in std_logic;
```

```
        W : out std_logic;
```

```
        RESET_CONT : out std_logic;
```

```
        EN : out std_logic
```

```
    );
```

```
end control_unit_board;
```

```
architecture Behavioral of control_unit_board is
```

```
constant N : positive := 2**len_add;
```

```
type STATO is (INIZIO, READ, WRITE, PROC, COUNT);
```

```
signal stato_corrente : STATO := INIZIO;
```

```
signal stato_prossimo : STATO;
```

```
begin
```

```
comb : process(START, FINE, stato_corrente, RESET_cu, R)
```

```
begin
```

```

if RESET_cu = '1' then
    stato_prossimo <= INIZIO;
    RESET_CONT <= '1';
else
    case stato_corrente is
        when INIZIO =>
            RESET_CONT <= '0';
            if START = '1' then
                if R = '1' then
                    stato_prossimo <= READ;
                else
                    stato_prossimo <= INIZIO;
                end if;
            else
                stato_prossimo <= INIZIO;
            end if;
        when COUNT =>
            if FINE = '1' then
                stato_prossimo <= INIZIO;
                RESET_CONT <= '1';
            else
                EN <= '0';
                stato_prossimo <= INIZIO;
            end if;
        when READ =>
            stato_prossimo <= PROC;
        when PROC =>
            W <= '1';
            stato_prossimo <= WRITE;
        when WRITE =>
            W <= '0';
            EN <= '1';
    end case;
end if;

```

```

    stato_prossimo <= COUNT;

end case;
end if;
end process comb;

seq: process(CLK_cu)
begin
    if rising_edge(CLK_cu) then
        stato_corrente <= stato_prossimo;
    end if;
end process seq;

```

end Behavioral;

All'interno di sistema_board sono state apportate le seguenti modifiche:

- Il segnale START_sis viene ora acquisito tramite uno switch della board e successivamente assegnato al segnale START della control_unit_board.
- È stata introdotta una nuova porta READ_sis, collegata a un pulsante della board, che permette di avviare la lettura di una locazione della ROM.
- È stata aggiunta anche la porta RESET_sis, collegata a un altro pulsante della board, utile per eseguire il reset del contatore.
- È stato implementato un debouncer per READ_sis: l'uscita stabilizzata cleared_read viene collegata sia al segnale R della control_unit_board sia al segnale read di ROM_N_8.
- Analogamente, è stato inserito un debouncer anche per RESET_sis, la cui uscita cleared_reset viene assegnata al segnale RESET_cu della control_unit_board.

Di seguito il codice VHDL:

```

entity sistema_board is
    generic( len_add : positive := 4);
    Port (
        CLK_sis : in std_logic;
        START_sis : in std_logic;
        READ_sis : in std_logic;
        RESET_sis : in std_logic;
        Y : out std_logic_vector(3 downto 0)
    );
end entity;

```

```
 );
end sistema_board;
```

architecture Behavioral of sistema_board is

COMPONENT memoria is

```
 Generic(
    len_add : positive := 4
);
Port(
    CLK_mem : in std_logic;
    write : in std_logic;
    address : in std_logic_vector (len_add-1 downto 0);
    inp_val : in std_logic_vector(3 downto 0);
    out_val : out std_logic_vector(3 downto 0)
);
```

END COMPONENT;

COMPONENT ROM_N_8 IS

```
 GENERIC(
    len_add : positive := 4
);
PORT (
    CLK_rom : in std_logic;
    address : in std_logic_vector(len_add - 1 downto 0);
    read : in std_logic;
    dout : out std_logic_vector(7 downto 0)
);
END COMPONENT;
```

component contatore is

```

generic( N: positive := 16);

port(
    clock : in std_logic;
    reset : in std_logic;
    load : in std_logic_vector((integer(ceil(log2(real(N)))))-1 downto 0); --IN QUESTO CASO NON SERVE
    enable_contatore : in std_logic; -- PER ABILITARE IL CONTATORE
    enable_caricamento: in std_logic; --NON SERVE
    cont : out std_logic_vector((integer(ceil(log2(real(N)))))-1 downto 0); --end_count
    div : out std_logic --variabile di uscita alta solo quando il contatore raggiunge il massimo -Y
);
end component;

```

component macchina_combinatoria is

```

Port(
    input_m : in std_logic_vector(7 downto 0);
    output_m : out std_logic_vector(3 downto 0)
);
end component;

```

component control_unit_board is

Generic(len_add : positive := 4);

```

Port(
    CLK_cu : in std_logic;
    START : in std_logic;
    FINE : in std_logic;
    RESET_cu : in std_logic;
    R : in std_logic;
    W : out std_logic;
    RESET_CONT : out std_logic;
    EN : out std_logic
);

```

end component;

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

```
signal temp_W : std_logic;
signal temp_RESET_CONT : std_logic;
signal temp_EN : std_logic;
signal temp_END_COUNT : std_logic;
signal temp_address : std_logic_vector(len_add -1 downto 0);
signal temp_input_mc : std_logic_vector(7 downto 0);
signal temp_out_mc : std_logic_vector(3 downto 0);
```

```
SIGNAL cleared_read : STD_LOGIC;
SIGNAL cleared_reset : STD_LOGIC;
```

```
begin
```

```
cu: control_unit_board
PORT MAP(
    CLK_cu => CLK_sis,
    START => START_sis,
    FINE => temp_END_COUNT,
```

```

RESET_cu => cleared_reset,
R => cleared_read,
W => temp_W,
RESET_CONT => temp_RESET_CONT,
EN => temp_EN
);

counter: contatore
port map(
    clock => CLK_sis,
    reset => temp_RESET_CONT,
    load => (others => '0'), --IN QUESTO CASO NON SERVE
    enable_contatore => temp_EN, -- PER ABILITARE IL CONTATORE
    enable_caricamento => '0', --NON SERVE
    cont => temp_address,
    div => temp_END_COUNT --variabile di uscita alta solo quando il contatore raggiunge il massimo
);

ROM : ROM_N_8
PORT MAP(
    CLK_rom => CLK_sis,
    address => temp_address,
    read => cleared_read,
    dout => temp_input_mc
);

MC : macchina_combinatoria
Port map(
    input_m => temp_input_mc,
    output_m => temp_out_mc
);

```

MEM : memoria

Port map(

```
CLK_mem => CLK_sis,  
write => temp_W,  
address => temp_address,  
inp_val => temp_out_mc,  
out_val => Y  
);
```

DebRead : ButtonDebouncer

Port map(

```
CLK => CLK_sis,  
RST => '0',  
BTN => READ_sis,  
CLEARED_BTN => cleared_read  
);
```

DebReset : ButtonDebouncer

Port map(

```
CLK => CLK_sis,  
RST => '0',  
BTN => RESET_sis,  
CLEARED_BTN => cleared_reset  
);
```

end Behavioral;

Sintesi su board di sviluppo

Le porte sono state mappate attraverso i seguenti constraint:

```
## Clock signal
```

```

set_property -dict { PACKAGE_PIN E3  IOSTANDARD LVCMOS33 } [get_ports { CLK_sis }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK_sis}];

##Switches

set_property -dict { PACKAGE_PIN J15  IOSTANDARD LVCMOS33 } [get_ports { START_sis }];
#IO_L24N_T3_RS0_15 Sch=sw[0]

## LEDs

set_property -dict { PACKAGE_PIN H17  IOSTANDARD LVCMOS33 } [get_ports { Y[0] }];
#IO_L18P_T2_A24_15 Sch=led[0]

set_property -dict { PACKAGE_PIN K15  IOSTANDARD LVCMOS33 } [get_ports { Y[1] }]; #IO_L24P_T3_RS1_15
Sch=led[1]

set_property -dict { PACKAGE_PIN J13  IOSTANDARD LVCMOS33 } [get_ports { Y[2] }]; #IO_L17N_T2_A25_15
Sch=led[2]

set_property -dict { PACKAGE_PIN N14  IOSTANDARD LVCMOS33 } [get_ports { Y[3] }]; #IO_L8P_T1_D11_14
Sch=led[3]

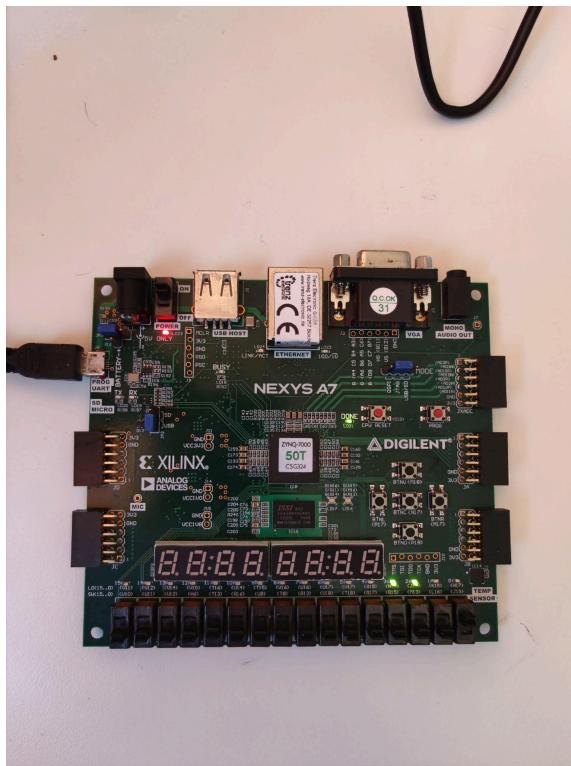
##Buttons

set_property -dict { PACKAGE_PIN P17  IOSTANDARD LVCMOS33 } [get_ports { READ_sis }];
#IO_L12P_T1_MRCC_14 Sch=btnl

set_property -dict { PACKAGE_PIN M17  IOSTANDARD LVCMOS33 } [get_ports { RESET_sis }];
#IO_L10N_T1_D15_14 Sch=btnr

```

Di seguito è fornito un esempio di configurazione:



Capitolo 3: Macchine aritmetiche

Esercizio 7: Moltiplicatore di Booth su 8 bit

Progettare, implementare in VHDL e simulare una macchina moltiplicatore di Booth in grado di effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna

Progetto e architettura

Un moltiplicatore è una macchina aritmetica che esegue l'operazione di moltiplicazione tra gli operandi che riceve in ingresso.

Il moltiplicatore di Booth è un ottimizzazione del moltiplicatore di Robertson, poiché sfrutta una particolare codifica, chiamata Codifica di Booth, che riduce il numero di somme necessarie, specialmente in lunghe sequenze di 1 e 0 nel moltiplicatore.

L'idea alla base è di trasformare il moltiplicatore in una nuova sequenza di bit, chiamata Successione di Booth, che permette di eseguire la moltiplicazione utilizzando un numero ridotto di operazioni.

Per applicare questa particolare codifica, si considera il moltiplicatore in gruppi di due bit alla volta.

Con l'obiettivo di gestire tutti i casi possibili, viene aggiunto uno 0 in più all'inizio del moltiplicatore, così da prelevare e considerare ogni possibile coppia generata da ogni singolo bit che compone il moltiplicatore.

Inizializzato il registro A (accumulatore), che rappresenta la somma parziale, a seconda dei bit che compongono la coppia estratta si esegue un'operazione diversa:

- 00 – 11: si esegue uno shift a destra
- 01: si somma il moltiplicando all'attuale somma parziale e si effettua uno shift a destra
- 10: si sottrae il moltiplicando dall'attuale somma parziale e si effettua uno shift a destra

Questa logica garantisce soprattutto una gestione intrinseca e automatica dell'operazione con eventuali numeri negativi, che nel moltiplicatore di Robertson venivano gestiti con una correzione effettuata sull'ultimo bit prelevato del moltiplicatore, mantenendo però un'architettura simile.

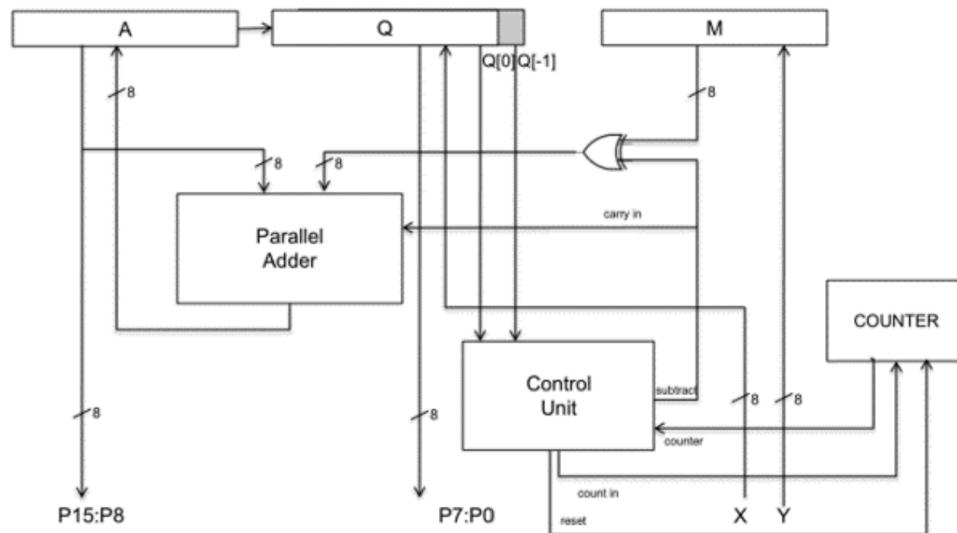
La progettazione del moltiplicatore si è basata sull'implementazione del moltiplicatore di Robertson, già fornito nel materiale didattico. Sebbene l'architettura di questa macchina presenti diverse somiglianze con quella del moltiplicatore di Booth, il nostro interesse si è concentrato soprattutto sulle differenze architettoniche tra i due.

Considerando due operandi da n bit ciascuno, le principali differenze sono le seguenti:

- Nel caso del moltiplicatore di Booth, il registro combinato AQ richiede un bit in più rispetto a quello di Robertson. Questo bit aggiuntivo, inizialmente impostato a 0, è fondamentale per permettere i confronti tra le coppie di bit necessari, portando così la lunghezza complessiva a $2n+1$ bit.
- Diversamente da Robertson, l'algoritmo di Booth non necessita del latch F, utilizzato per gestire i casi in cui uno degli operandi è negativo. Infatti, grazie al metodo delle coppie di bit, Booth è in grado di determinare autonomamente l'operazione da eseguire: uno shift semplice (per le coppie 00 o 11), un'addizione (01) oppure una sottrazione (10).
- Non è inoltre richiesto il MUX 2:1, che nella versione di Robertson ha il compito di scegliere tra l'operando M e lo zero da inviare al sommatore parallelo.

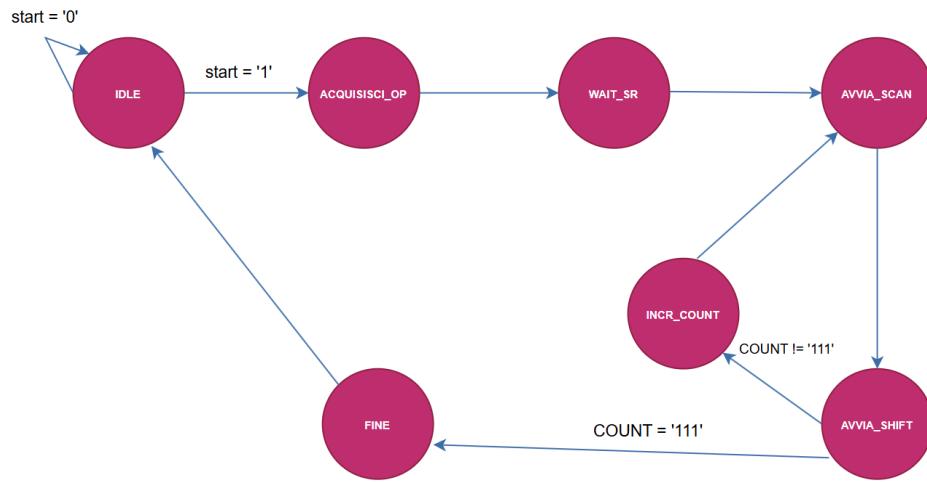
Tenendo conto di queste differenze, abbiamo apportato le modifiche necessarie all'unità operativa del moltiplicatore di Robertson, adattandola al funzionamento previsto dall'algoritmo di Booth.

Di seguito è riportata l'architettura implementata.



Anche a livello comportamentale emergono delle differenze significative, poiché l'algoritmo impiegato nel moltiplicatore di Booth si basa sull'analisi di coppie di bit adiacenti del moltiplicatore. Questo cambiamento nell'approccio computazionale ha reso necessario un intervento sull'unità di controllo. In particolare,

abbiamo sviluppato l'ASF (Automa a Stati Finiti) che descrive il funzionamento del moltiplicatore secondo l'algoritmo di Booth, come mostrato nella figura seguente.



Analizziamo ora più nel dettaglio gli stati e le transizioni della macchina a stati finiti che modella il comportamento del moltiplicatore di Booth:

- **Idle:** stato iniziale in cui la macchina resta in attesa finché il segnale di avvio (start) è basso. Al rilevamento di un segnale alto, si passa allo stato successivo *acquisisci_op*.
- **Acquisisci_op:** in questo stato vengono caricati gli operandi. Si abilita innanzitutto il caricamento del moltiplicando nel registro M (*loadM* = '1'), seguito dal caricamento del moltiplicatore, accompagnato da otto zeri iniziali, nel registro A.Q (*loadAQ* = '1'). Completata questa fase, si transita nello stato *wait_sr*.
- **Wait_sr:** si attende la conclusione del caricamento degli operandi. Una volta completata l'operazione, si procede allo stato **Avvia_scan**.
- **Avvia_scan:** in base al valore della coppia di bit q10, si decide l'operazione da eseguire:
 - Se q10 = "01": si abilita il caricamento del risultato della somma nel registro A.
 - Se q10 = "10": si abilita la sottrazione e il caricamento del relativo risultato in A.
 - Se q10 = "00" o q10 = "11": non viene eseguita alcuna operazione aritmetica, ma si passa direttamente allo shift.
- **Avvia_shift:** viene abilitato lo shift verso destra (*en_shift* = '1'). Se il contatore ha raggiunto il valore massimo (*count* = "111"), significa che il processo è completo e si passa allo stato **Fine**. Altrimenti, si continua con **Incr_count**.
- **Incr_count:** si abilita l'incremento del contatore (*count_in* = '1') e si ritorna allo stato **Avvia_scan** per proseguire con la successiva iterazione.

- **Fine:** viene segnalata la conclusione dell'operazione (`stop_cu = '1'`). Da qui, la macchina ritorna nello stato **Idle**, pronta per una nuova operazione.

Implementazione

Per la realizzazione della macchina, abbiamo sviluppato diverse componenti:

- Full Adder (vedi Appendice)
- Adder Carry Ripple (vedi Appendice)
- Adder Subber: utilizzando il Ripple Carry Adder per implementare sia la somma che la sottrazione tra stringhe di bit
- Mux 2:1 (vedi Appendice)
- Registro Parallelo - Parallelo (vedi Appendice): su 8 bit, utilizzato per mantenere il moltiplicando M
- Shift Register : su 17 bit, utilizzato per il registro A.Q, che conterrà la somma parziale e che bisogna scorrere verso destra per andare avanti con la moltiplicazione.
- Contatore Mod. 8 (vedi Appendice): per conteggiare le operazioni effettuate e capire dunque quando il moltiplicatore dovrà fermarsi.
- Unità Operativa
- Unità di Controllo

Implementazione: Adder Subber

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

-- Entità che implementa un sommatore/sottrattore a 8 bit
-- Quando cin = 0: Z = X + Y
-- Quando cin = 1: Z = X - Y (complemento a 2)

ENTITY adder_sub IS
    PORT (
        operand_a, operand_b : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- Operandi di ingresso
        operation_sel : IN STD_LOGIC;                      -- Selezione operazione: 0=somma, 1=sottrazione
        result : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);          -- Risultato dell'operazione
        carry_out : OUT STD_LOGIC                          -- Riporto in uscita
    );
END adder_sub;

```

ARCHITECTURE structural OF adder_sub IS

```

COMPONENT ripple_carry IS
  PORT (
    X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    c_in : IN STD_LOGIC;
    c_out : OUT STD_LOGIC;
    Z : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
  END COMPONENT;

-- Segnale per memorizzare il complemento del secondo operando
SIGNAL inverted_operand : STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN

  -- Generazione del complemento a 1 di operand_b quando operation_sel = 1
  -- Lascia operand_b invariato quando operation_sel = 0
  complemento_operand : FOR i IN 0 TO 7 GENERATE
    inverted_operand(i) <= operand_b(i) XOR operation_sel;
  END GENERATE;

  -- Istanziazione del sommatore a propagazione di riporto
  -- operation_sel viene usato anche come carry in iniziale per completare
  -- l'operazione di complemento a 2 in caso di sottrazione
  adder : ripple_carry PORT MAP(operand_a, inverted_operand, operation_sel, carry_out, result);

END structural;

```

Implementazione: Shift Register 17 bit

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

```

```

-- Registro a scorrimento a 17 bit con caricamento parallelo e scorrimento a destra
-- Utilizzato nell'algoritmo di Booth per gestire i registri A e Q

ENTITY shift_register IS

PORT (
    data_load : IN STD_LOGIC_VECTOR(16 DOWNTO 0);      -- Dati da caricare in parallelo
    shift_in : IN STD_LOGIC;                            -- Bit di ingresso per lo scorrimento
    clock, reset, enable_load, enable_shift : IN STD_LOGIC; -- Segnali di controllo
    data_out : OUT STD_LOGIC_VECTOR(16 DOWNTO 0)        -- Contenuto del registro
);

END shift_register;

```

ARCHITECTURE behavioural OF shift_register IS

```

    SIGNAL register_value : STD_LOGIC_VECTOR(16 DOWNTO 0); -- Valore corrente memorizzato nel registro

```

BEGIN

```

    -- Processo di aggiornamento del registro sincronizzato sul fronte di salita del clock
    shift_process : PROCESS (clock)

```

BEGIN

```

        IF (clock'event AND clock = '1') THEN

```

```

            IF (reset = '1') THEN

```

```

                -- Reset sincrono: azzera il contenuto del registro

```

```

                register_value <= (OTHERS => '0');

```

ELSE

```

            IF (enable_load = '1') THEN --caricamento iniziale del moltiplicatore

```

```

                -- Caricamento parallelo dei dati

```

```

                register_value <= data_load;

```

```

            ELSIF (enable_shift = '1') THEN

```

```

                -- Scorrimento a destra: tutti i bit si spostano di una posizione

```

```

                -- e il nuovo bit più significativo viene preso dall'ingresso seriale

```

```

                register_value(15 DOWNTO 0) <= register_value(16 DOWNTO 1);

```

```

register_value(16) <= shift_in;
END IF;
END IF;

END IF;
END PROCESS;

-- Assegnazione del valore interno all'uscita
data_out <= register_value;
END behavioural;

```

Implementazione: Unità Operativa

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

-- Unità operativa del moltiplicatore di Booth
-- Implementa la parte datapath dell'algoritmo di moltiplicazione
ENTITY unita_operativa IS
PORT (
    multiplicand, multiplier : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- Operandi di ingresso
    clock, reset : IN STD_LOGIC; -- Segnali di sincronizzazione
    load_result, enable_shift, load_multiplicand, subtract_op, select_input, enable_count : IN STD_LOGIC;
    -- Segnali di controllo
    iteration_count : OUT STD_LOGIC_VECTOR(2 DOWNTO 0); -- Contatore delle iterazioni
    product : OUT STD_LOGIC_VECTOR(16 DOWNTO 0) -- Risultato della moltiplicazione
);
END unita_operativa;

```

ARCHITECTURE structural OF unita_operativa IS

COMPONENT adder_sub IS

```
POR T (  
    operand_a, operand_b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);  
    operation_sel : IN STD_LOGIC;  
    result : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);  
    carry_out : OUT STD_LOGIC  
);
```

```
END COMPONENT;
```

```
COMPONENT registro8 IS  
POR T (  
    data_in : IN STD_LOGIC_VECTOR(7 DOWNTO 0);  
    clock, reset, enable : IN STD_LOGIC;  
    data_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)  
);
```

```
END COMPONENT;
```

```
COMPONENT mux_21 IS  
GENERIC (  
    width : INTEGER RANGE 0 TO 17 := 8  
);  
POR T (  
    input_0, input_1 : IN STD_LOGIC_VECTOR(width - 1 DOWNTO 0);  
    select_input : IN STD_LOGIC;  
    output : OUT STD_LOGIC_VECTOR(width - 1 DOWNTO 0)  
);
```

```
END COMPONENT;
```

```
COMPONENT shift_register IS  
POR T (
```

```
    data_load : IN STD_LOGIC_VECTOR(16 DOWNTO 0);
    shift_in : IN STD_LOGIC;
    clock, reset, enable_load, enable_shift : IN STD_LOGIC;
    data_out : OUT STD_LOGIC_VECTOR(16 DOWNTO 0)
);


```

```
END COMPONENT;
```

```
--component FFD is
--port( clock, reset, d: in std_logic;
--y: out std_logic);
--end component;
```

```
COMPONENT cont_mod8 IS
```

```
    PORT (
        clk, rst : IN STD_LOGIC;
        enable_count : IN STD_LOGIC;
        counter_value : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
);


```

```
END COMPONENT;
```

```
-- Segnali interni per l'interconnessione dei componenti
```

```
    SIGNAL stored_multiplicand : STD_LOGIC_VECTOR(7 DOWNTO 0); -- Valore del moltiplicando
memorizzato

    SIGNAL initial_reg_value : STD_LOGIC_VECTOR(16 DOWNTO 0); -- Valore iniziale per il registro A.Q

    SIGNAL reg_input : STD_LOGIC_VECTOR(16 DOWNTO 0); -- Ingresso del registro A.Q

    SIGNAL reg_output : STD_LOGIC_VECTOR(16 DOWNTO 0); -- Uscita del registro A.Q

    SIGNAL adder_result : STD_LOGIC_VECTOR(7 DOWNTO 0); -- Risultato dell'operazione di
somma/sottrazione
```

```
    SIGNAL updated_reg_value : STD_LOGIC_VECTOR(16 DOWNTO 0); -- Valore aggiornato dopo la
somma/sottrazione
```

```

SIGNAL carry_out : STD_LOGIC;           -- Riporto in uscita dell'adder (non utilizzato)
SIGNAL shift_in_bit : STD_LOGIC;         -- Bit di ingresso per lo shift

BEGIN

-- 1) Memorizzazione del moltiplicando in un registro
M_REG : registro8 PORT MAP(multiplier, clock, reset, load_multiplicand, stored_multiplicand);

-- 2) Preparazione dei valori per il registro A.Q

-- Valore iniziale: 8 bit a 0 + moltiplicando + 1 bit a 0
initial_reg_value <= "00000000" & multiplicand & "0";

-- Valore aggiornato dopo la somma/sottrazione: risultato dell'adder + parte bassa del registro
updated_reg_value <= adder_result & reg_output(8 DOWNTO 0);

-- Multiplexer per selezionare l'ingresso del registro A.Q
MUX_REG_INPUT : mux_21 GENERIC MAP(width => 17) PORT MAP(initial_reg_value, updated_reg_value,
select_input, reg_input);

-- 3) Gestione dell'ingresso seriale per lo shift aritmetico
shift_in_bit <= reg_output(16); -- Replica del bit più significativo (shift aritmetico)

-- 4) Registro A.Q per memorizzare il risultato parziale
SHIFT_REG : shift_register PORT MAP(reg_input, shift_in_bit, clock, reset, load_result, enable_shift,
reg_output);

-- 5) Sommatore/sottrattore per le operazioni aritmetiche
ARITHMETIC_UNIT : adder_sub PORT MAP(reg_output(16 DOWNTO 9), stored_multiplicand, subtract_op,
adder_result, carry_out);

-- 6) Contatore per tenere traccia delle iterazioni
ITERATION_COUNTER : cont_mod8 PORT MAP(clock, reset, enable_count, iteration_count);

```

```
-- 7) Assegnazione dell'uscita  
product <= reg_output;
```

```
END structural;
```

Implementazione: Unità di Controllo

```
library IEEE;  
use IEEE.std_logic_1164.all;  
use IEEE.numeric_std.all;  
  
-- Unità di controllo per il moltiplicatore di Booth  
-- Gestisce la sequenza delle operazioni dell'algoritmo attraverso una macchina a stati finiti  
ENTITY unita_controllo IS  
PORT (  
    booth_bits : IN STD_LOGIC_VECTOR(1 DOWNTO 0); -- Bit per la codifica di Booth (Q1,Q0)  
    clock, reset, start : IN STD_LOGIC; -- Segnali di controllo esterni  
    iteration_count : IN STD_LOGIC_VECTOR(2 DOWNTO 0); -- Contatore delle iterazioni  
    load_multiplicand, enable_count, load_result, enable_shift : OUT STD_LOGIC; -- Segnali di controllo  
    select_addsub, perform_subtract, operation_done : OUT STD_LOGIC -- Segnali di controllo  
);  
END unita_controllo;
```

```
ARCHITECTURE Behavioral OF unita_controllo IS  
-- Stati della macchina a stati finiti  
TYPE state IS (idle, acquisisci_op, wait_sr, avvia_scan, avvia_shift, incr_count, fine);  
SIGNAL current_state, next_state : state;
```

```
BEGIN
```

```
--selM <= q0;-- in ogni istante la selezione del mux è data dal bit meno significativo di A.Q (q0)
```

```
-- Registro di stato: memorizza lo stato corrente della FSM

reg_stato : PROCESS (clock)

BEGIN

IF (rising_edge(clock)) THEN

    IF (reset = '1') THEN

        current_state <= idle; -- Reset: torna allo stato iniziale

    ELSE

        current_state <= next_state; -- Aggiorna lo stato

    END IF;

END IF;

END PROCESS;
```

```
-- Logica combinatoria: determina lo stato successivo e i segnali di uscita

comb : PROCESS (current_state, start, iteration_count, booth_bits)

BEGIN

-- Attenzione! questo process si attiva ogni volta che c'è una variazione nei segnali della sensitivity list
-- current_state e count per loro natura variano sempre in corrispondenza del fronte di salita del clock
-- start viene dall'esterno: se non varia (sale e scende) col fronte del clock, si potrebbe avere una situazione
-- in cui il next_state varia ma non ha modo da stabilizzarsi (perchè current_state non è ancora variato)
-- quando il moltiplicatore sar? messo su board, START dovrà essere generato come uscita del button
debouncer
```

```
-- Inizializzazione dei segnali di controllo (valori di default)

enable_count <= '0';

perform_subtract <= '0';

select_addsub <= '0';

load_result <= '0';

load_multiplicand <= '0';

operation_done <= '0';

enable_shift <= '0';
```

CASE current_state IS

WHEN idle =>

-- Stato di attesa: aspetta il segnale di start

IF (start = '1') THEN

next_state <= acquisisci_op;

ELSE

next_state <= idle;

END IF;

WHEN acquisisci_op =>

-- Acquisizione operandi: carica i valori nei registri

load_multiplicand <= '1'; -- Carica il moltiplicando nel registro M

load_result <= '1'; -- Carica il moltiplicatore nello shift register A.Q

next_state <= wait_sr;

WHEN wait_sr =>

-- Stato di attesa per stabilizzazione dei registri

next_state <= avvia_scan;

WHEN avvia_scan =>

-- Analisi dei bit di Booth per determinare l'operazione da eseguire

IF (booth_bits = "01") THEN

-- Caso 01: Addizione (A = A + M)

select_addsub <= '1';

load_result <= '1';

next_state <= avvia_shift;

ELSIF (booth_bits = "10") THEN

-- Caso 10: Sottrazione (A = A - M)

perform_subtract <= '1';

select_addsub <= '1';

```

load_result <= '1';
next_state <= avvia_shift;

ELSIF (booth_bits = "00" OR booth_bits = "11") THEN
  -- Casi 00 e 11: Nessuna operazione
  next_state <= avvia_shift;
END IF;

WHEN avvia_shift =>
  -- Esecuzione dello shift aritmetico a destra
  enable_shift <= '1';

  -- Verifica se è l'ultima iterazione
  IF (iteration_count = "111") THEN
    next_state <= fine;
  ELSE
    next_state <= incr_count;
  END IF;

WHEN incr_count =>
  -- Incremento del contatore delle iterazioni
  enable_count <= '1';
  next_state <= avvia_scan;

WHEN fine =>
  -- Completamento dell'operazione
  operation_done <= '1';
  next_state <= idle;

END CASE;

END PROCESS;
END Behavioral;

```

Implementazione: Entità Finale

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

-- Moltiplicatore di Booth per numeri a 8 bit con segno
-- Implementa l'algoritmo di Booth per la moltiplicazione binaria

ENTITY molt_booth IS

    PORT (
        clock, reset, start : IN STD_LOGIC;          -- Segnali di controllo
        multiplicand, multiplier : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- Operandi di ingresso
        --stop: out std_logic; --a che serve?
        product : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);    -- Risultato della moltiplicazione
        operation_complete : OUT STD_LOGIC            -- Segnale di completamento operazione
    );
END molt_booth;

ARCHITECTURE structural OF molt_booth IS

COMPONENT unita_controllo IS

    PORT (
        booth_bits : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        clock, reset, start : IN STD_LOGIC;
        iteration_count : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
        load_multiplicand, enable_count, load_result, enable_shift : OUT STD_LOGIC;
        select_addsub, perform_subtract, operation_done : OUT STD_LOGIC
    );
END COMPONENT;

END COMPONENT;
```

```
COMPONENT unita_operativa IS
```

```

PORT (
    multiplicand, multiplier : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    clock, reset : IN STD_LOGIC;
    load_result, enable_shift, load_multiplicand, subtract_op, select_input, enable_count : IN STD_LOGIC;
    iteration_count : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
    product : OUT STD_LOGIC_VECTOR(16 DOWNTO 0)
);


```

```
END COMPONENT;
```

-- Segnali di interconnessione tra unità di controllo e unità operativa

```

SIGNAL lsb_bits : STD_LOGIC_VECTOR(1 DOWNTO 0);      -- Bit meno significativi per l'algoritmo di Booth
SIGNAL ctrl_select_aq, ctrl_subtract, ctrl_load_aq : STD_LOGIC;
SIGNAL ctrl_clock : STD_LOGIC;


```

```
SIGNAL counter_value : STD_LOGIC_VECTOR(2 DOWNTO 0); -- Contatore per tracciare le iterazioni
```

```
SIGNAL internal_product : STD_LOGIC_VECTOR(16 DOWNTO 0); -- Risultato interno (include bit extra)
```

```
SIGNAL enable_count, load_adder : STD_LOGIC;
```

```
SIGNAL count_complete : STD_LOGIC;
```

```
SIGNAL enable_shift : STD_LOGIC;
```

```
SIGNAL load_multiplicand : STD_LOGIC;
```

```
SIGNAL process_done : STD_LOGIC;                  -- Segnale di completamento interno
```

```
SIGNAL reset_to_uo : STD_LOGIC;                  -- Segnale di reset per l'unità operativa
```

```
BEGIN
```

-- Istanziazione dell'unità di controllo

-- Gestisce la sequenza delle operazioni dell'algoritmo di Booth

UC : unita_controllo PORT MAP

```
(
```

```
    lsb_bits, clock, reset, start,
```

```
    counter_value,
```

```
    load_multiplicand, enable_count, ctrl_load_aq, enable_shift,
```

```

ctrl_select_aq, ctrl_subtract, process_done
);

-- Istanziazione dell'unità operativa
-- Esegue le operazioni aritmetiche e di shift richieste dall'algoritmo
UO : unita_operativa PORT MAP
(
    multiplicand, multiplier, clock, reset, ctrl_load_aq, enable_shift, load_multiplicand,
    ctrl_subtract, ctrl_select_aq, enable_count, counter_value, internal_product
);

-- Connessione dei segnali tra le unità
lsb_bits <= internal_product(1 DOWNTO 0); -- Invio all'unità di controllo i due bit meno significativi del
registro A.Q
product <= internal_product(16 DOWNTO 1); -- Estrazione del risultato finale

-- la UO viene resettata sia se arriva un reset dall'esterno sia se l'operazione di moltiplicazione termina
-- reset_to_uo <= reset or process_done;

operation_complete <= process_done;
END structural;

```

Simulazione

Il testbench da noi realizzato contempla l'esecuzione di due operazioni di moltiplicazione: la prima con entrambi gli operandi positivi, la seconda con operandi di segno opposto.

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

-- Testbench per il moltiplicatore di Booth
ENTITY mbooth_tb IS
END mbooth_tb;

```

```

ARCHITECTURE behavioural OF mbooth_tb IS

COMPONENT molt_booth IS
PORT (
    clock, reset, start : IN STD_LOGIC;
    multiplicand, multiplier : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    product : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    operation_complete : OUT STD_LOGIC
);
END COMPONENT;

-- Definizione del periodo di clock per la simulazione
CONSTANT clk_period : TIME := 20 ns;

-- Segnali per gli operandi di ingresso e il risultato
SIGNAL multiplicand, multiplier : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL product : STD_LOGIC_VECTOR(15 DOWNTO 0);

-- Segnali di controllo
SIGNAL clock_sig, reset_sig, start_sig : STD_LOGIC;
SIGNAL operation_done : STD_LOGIC;

-- Segnale per terminare la simulazione
SIGNAL end_simulation : STD_LOGIC := '0';

BEGIN

-- Istanziazione del componente da testare (Unit Under Test)
uut : molt_booth PORT MAP(clock_sig, reset_sig, start_sig, multiplicand, multiplier, product,
operation_done);

-- Processo per la generazione del clock

```

```
clk_process : PROCESS
BEGIN
  WHILE (end_simulation = '0') LOOP
    clock_sig <= '1';
    WAIT FOR clk_period/2;
    clock_sig <= '0';
    WAIT FOR clk_period/2;
  END LOOP;
  WAIT;
END PROCESS;
```

-- SIMULARE PER 9000 NS

-- Processo principale di simulazione

```
sim : PROCESS
```

```
BEGIN
```

```
  WAIT FOR 100 ns;
```

-- Reset iniziale del sistema

```
  reset_sig <= '1';
```

```
  WAIT FOR 20 ns;
```

```
  reset_sig <= '0';
```

-- ----- operazione numero 1:

-- 15*3=45 (002D)

-- Impostazione dei valori per la prima moltiplicazione: 15 * 3

```
  multiplicand <= "00001111"; -- 15 in binario
```

```
multiplier <= "00000011"; -- 3 in binario

-- start deve essere visto da clk_div: poichè sarà generato dal button debouncer si aggiungerà anche il
clk_div

-- al button debouncer e il segnale di start deve durare quanto il periodo del clk rallentato

WAIT FOR 40 ns;

-- Avvio della moltiplicazione
start_sig <= '1';

WAIT FOR 20 ns;

start_sig <= '0';

-- Attesa per il completamento dell'operazione
WAIT FOR 600 ns;

-- Reset del sistema per la prossima operazione
reset_sig <= '1';

WAIT FOR 20 ns;

reset_sig <= '0';

-- ----- operazione numero 2:

-- 15*(-3)=-45 (0053)

-- Impostazione dei valori per la seconda moltiplicazione: 15 * (-3)
multiplicand <= "00001111"; -- 15 in binario
multiplier <= "11111101"; -- -3 in complemento a 2
```

WAIT FOR 40 ns;

-- Avvio della moltiplicazione

start_sig <= '1';

WAIT FOR 20 ns;

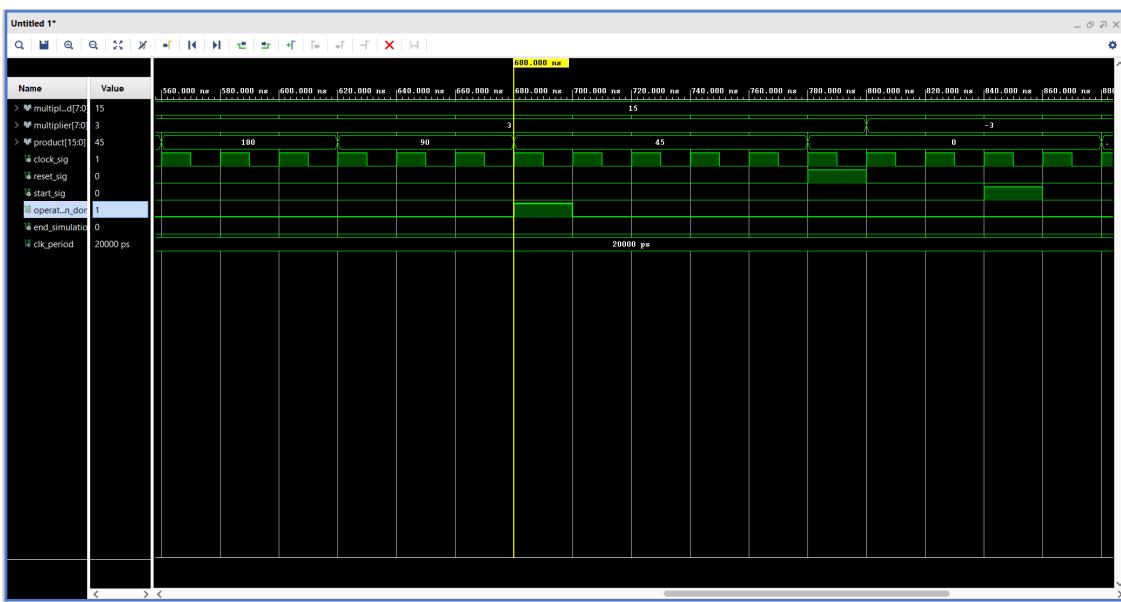
start_sig <= '0';

WAIT;

END PROCESS;

END behavioural;

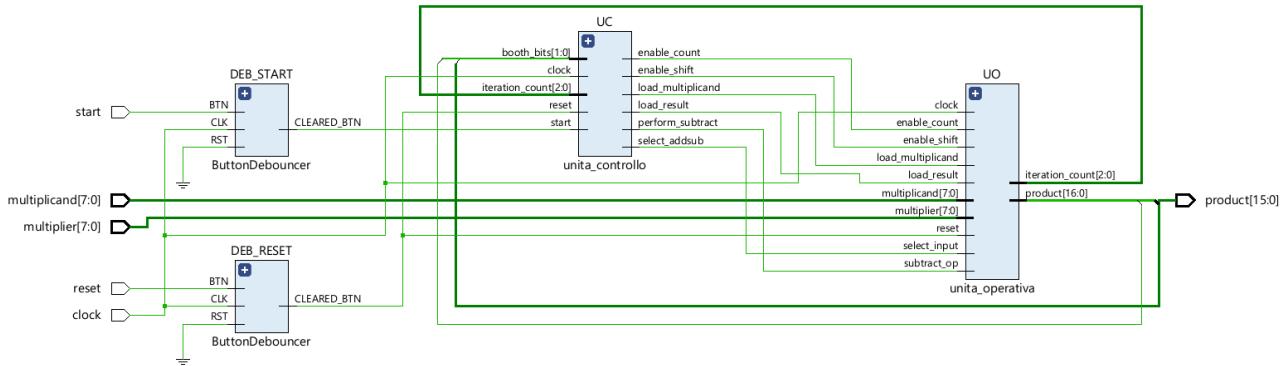
Di seguito l'immagine dei segnali prodotta dalla simulazione.



Esercizio 7.2: Implementazione su board moltiplicatore di Booth

Sintetizzare il moltiplicatore implementato al punto 7.1 su FPGA e testarlo mediante l'utilizzo dei dispositivi di input/output (switch, bottoni, led, display) presenti sulla board di sviluppo in dotazione. La modalità di utilizzo degli stessi è a completa discrezione degli studenti.

Implementazione



Per l'implementazione del moltiplicatore su board rispetto all'esercizio precedente abbiamo:

- Aggiunto i DEBOUNCER per i bottoni di start e reset
- Aggiunto 16 switch per il caricamento degli operandi (8 switch per operando)
- Attivato i 16 led della board per mostrare il risultato della moltiplicazione.

```
ENTITY molt_booth_on_board IS
PORT (
    clock, reset, start : IN STD_LOGIC;          -- Segnali di controllo
    multiplicand, multiplier : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- Operandi di ingresso
    --stop: out std_logic; --a che serve?
    product : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)      -- Risultato della moltiplicazione
    --operation_complete : OUT STD_LOGIC           -- Segnale di completamento operazione
);
END molt_booth_on_board;
```

ARCHITECTURE structural OF molt_booth_on_board IS

```
COMPONENT unita_controllo IS
PORT (
    booth_bits : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    clock, reset, start : IN STD_LOGIC;
    iteration_count : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
    load_multiplicand, enable_count, load_result, enable_shift : OUT STD_LOGIC;
);
```

```

    select_addsub, perform_subtract, operation_done : OUT STD_LOGIC
);

END COMPONENT;

COMPONENT unita_operativa IS
    PORT (
        multiplicand, multiplier : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        clock, reset : IN STD_LOGIC;
        load_result, enable_shift, load_multiplicand, subtract_op, select_input, enable_count : IN STD_LOGIC;
        iteration_count : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
        product : OUT STD_LOGIC_VECTOR(16 DOWNTO 0)
    );
END COMPONENT;

COMPONENT ButtonDebouncer IS
    GENERIC (
        CLK_period: integer := 10; -- periodo del clock (della board) in nanosecondi
        btn_noise_time: integer := 10000000 -- durata stimata dell'oscillazione del bottone in nanosecondi
            -- il valore di default è 10 millisecondi
    );
    PORT ( RST : in STD_LOGIC;
        CLK : in STD_LOGIC;
        BTN : in STD_LOGIC;
        CLEARED_BTN : out STD_LOGIC);
END COMPONENT;

-- Segnali di interconnessione tra unità di controllo e unità operativa
SIGNAL lsb_bits : STD_LOGIC_VECTOR(1 DOWNTO 0);      -- Bit meno significativi per l'algoritmo di Booth
SIGNAL ctrl_select_aq, ctrl_subtract, ctrl_load_aq : STD_LOGIC;

```

```

SIGNAL ctrl_clock : STD_LOGIC;

SIGNAL counter_value : STD_LOGIC_VECTOR(2 DOWNTO 0); -- Contatore per tracciare le iterazioni
SIGNAL internal_product : STD_LOGIC_VECTOR(16 DOWNTO 0); -- Risultato interno (include bit extra)
SIGNAL enable_count, load_adder : STD_LOGIC;
SIGNAL count_complete : STD_LOGIC;
SIGNAL enable_shift : STD_LOGIC;
SIGNAL load_multiplicand : STD_LOGIC;
-- SIGNAL process_done : STD_LOGIC; -- Segnale di completamento interno
SIGNAL temp_reset: std_logic; -- segnale di reset in ingresso
SIGNAL temp_start : std_logic;

SIGNAL reset_to_uo : STD_LOGIC; -- Segnale di reset per l'unità operativa

BEGIN
    -- Istanziazione dell'unità di controllo
    -- Gestisce la sequenza delle operazioni dell'algoritmo di Booth
    UC : unita_controllo PORT MAP
    (
        lsb_bits, clock, temp_reset, temp_start,
        counter_value,
        load_multiplicand, enable_count, ctrl_load_aq, enable_shift,
        ctrl_select_aq, ctrl_subtract
    );
    -- Istanziazione dell'unità operativa
    -- Esegue le operazioni aritmetiche e di shift richieste dall'algoritmo
    UO : unita_operativa PORT MAP
    (
        multiplicand, multiplier, clock, temp_reset, ctrl_load_aq, enable_shift, load_multiplicand,
        ctrl_subtract, ctrl_select_aq, enable_count, counter_value, internal_product
    );

```

```
DEB_START : ButtonDebouncer port map
```

```
('0', clock, start, temp_start);
```

```
DEB_RESET : ButtonDebouncer port map
```

```
('0', clock, reset, temp_reset);
```

```
-- Connessione dei segnali tra le unità
```

```
lsb_bits <= internal_product(1 DOWNTO 0); -- Invio all'unità di controllo i due bit meno significativi del  
registro A.Q
```

```
product <= internal_product(16 DOWNTO 1); -- Estrazione del risultato finale
```

```
-- la UO viene resettata sia se arriva un reset dall'esterno sia se l'operazione di moltiplicazione termina
```

```
-- reset_to_uo <= reset or process_done;
```

```
-- operation_complete <= process_done;
```

```
END structural;
```

I constraint attivi sulla board sono i seguenti:

```
## Clock signal
```

```
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { clock }];  
#IO_L12P_T1_MRCC_35 Sch=clk100mhz
```

```
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clock}];
```

```
##Switches
```

```
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 } [get_ports { multiplier[0] }];  
#IO_L24N_T3_RS0_15 Sch=sw[0]
```

```
set_property -dict { PACKAGE_PIN L16 IOSTANDARD LVCMOS33 } [get_ports { multiplier[1] }];  
#IO_L3N_T0_DQS_EMCCCLK_14 Sch=sw[1]
```

```
set_property -dict { PACKAGE_PIN M13 IOSTANDARD LVCMOS33 } [get_ports { multiplier[2] }];  
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]
```

```

set_property -dict { PACKAGE_PIN R15  IOSTANDARD LVCMOS33 } [get_ports { multiplier[3] }];
#IO_L13N_T2_MRCC_14 Sch=sw[3]

set_property -dict { PACKAGE_PIN R17  IOSTANDARD LVCMOS33 } [get_ports { multiplier[4] }];
#IO_L12N_T1_MRCC_14 Sch=sw[4]

set_property -dict { PACKAGE_PIN T18  IOSTANDARD LVCMOS33 } [get_ports { multiplier[5] }];
#IO_L7N_T1_D10_14 Sch=sw[5]

set_property -dict { PACKAGE_PIN U18  IOSTANDARD LVCMOS33 } [get_ports { multiplier[6] }];
#IO_L17N_T2_A13_D29_14 Sch=sw[6]

set_property -dict { PACKAGE_PIN R13  IOSTANDARD LVCMOS33 } [get_ports { multiplier[7] }];
#IO_L5N_T0_D07_14 Sch=sw[7]

set_property -dict { PACKAGE_PIN T8   IOSTANDARD LVCMOS18 } [get_ports { multiplicand[0] }];
#IO_L24N_T3_34 Sch=sw[8]

set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS18 } [get_ports { multiplicand[1] }]; #IO_25_34
Sch=sw[9]

set_property -dict { PACKAGE_PIN R16  IOSTANDARD LVCMOS33 } [get_ports { multiplicand[2] }];
#IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]

set_property -dict { PACKAGE_PIN T13  IOSTANDARD LVCMOS33 } [get_ports { multiplicand[3] }];
#IO_L23P_T3_A03_D19_14 Sch=sw[11]

set_property -dict { PACKAGE_PIN H6   IOSTANDARD LVCMOS33 } [get_ports { multiplicand[4] }];
#IO_L24P_T3_35 Sch=sw[12]

set_property -dict { PACKAGE_PIN U12  IOSTANDARD LVCMOS33 } [get_ports { multiplicand[5] }];
#IO_L20P_T3_A08_D24_14 Sch=sw[13]

set_property -dict { PACKAGE_PIN U11  IOSTANDARD LVCMOS33 } [get_ports { multiplicand[6] }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]

set_property -dict { PACKAGE_PIN V10  IOSTANDARD LVCMOS33 } [get_ports { multiplicand[7] }];
#IO_L21P_T3_DQS_14 Sch=sw[15]

```

LEDs

```

set_property -dict { PACKAGE_PIN H17  IOSTANDARD LVCMOS33 } [get_ports { product[0] }];
#IO_L18P_T2_A24_15 Sch=led[0]

set_property -dict { PACKAGE_PIN K15  IOSTANDARD LVCMOS33 } [get_ports { product[1] }];
#IO_L24P_T3_RS1_15 Sch=led[1]

set_property -dict { PACKAGE_PIN J13  IOSTANDARD LVCMOS33 } [get_ports { product[2] }];
#IO_L17N_T2_A25_15 Sch=led[2]

set_property -dict { PACKAGE_PIN N14  IOSTANDARD LVCMOS33 } [get_ports { product[3] }];
#IO_L8P_T1_D11_14 Sch=led[3]

set_property -dict { PACKAGE_PIN R18  IOSTANDARD LVCMOS33 } [get_ports { product[4] }];
#IO_L7P_T1_D09_14 Sch=led[4]

```

```

set_property -dict { PACKAGE_PIN V17 IOSTANDARD LVCMOS33 } [get_ports { product[5] }];
#IO_L18N_T2_A11_D27_14 Sch=led[5]

set_property -dict { PACKAGE_PIN U17 IOSTANDARD LVCMOS33 } [get_ports { product[6] }];
#IO_L17P_T2_A14_D30_14 Sch=led[6]

set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports { product[7] }];
#IO_L18P_T2_A12_D28_14 Sch=led[7]

set_property -dict { PACKAGE_PIN V16 IOSTANDARD LVCMOS33 } [get_ports { product[8] }];
#IO_L16N_T2_A15_D31_14 Sch=led[8]

set_property -dict { PACKAGE_PIN T15 IOSTANDARD LVCMOS33 } [get_ports { product[9] }];
#IO_L14N_T2_SRCC_14 Sch=led[9]

set_property -dict { PACKAGE_PIN U14 IOSTANDARD LVCMOS33 } [get_ports { product[10] }];
#IO_L22P_T3_A05_D21_14 Sch=led[10]

set_property -dict { PACKAGE_PIN T16 IOSTANDARD LVCMOS33 } [get_ports { product[11] }];
#IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]

set_property -dict { PACKAGE_PIN V15 IOSTANDARD LVCMOS33 } [get_ports { product[12] }];
#IO_L16P_T2_CSI_B_14 Sch=led[12]

set_property -dict { PACKAGE_PIN V14 IOSTANDARD LVCMOS33 } [get_ports { product[13] }];
#IO_L22N_T3_A04_D20_14 Sch=led[13]

set_property -dict { PACKAGE_PIN V12 IOSTANDARD LVCMOS33 } [get_ports { product[14] }];
#IO_L20N_T3_A07_D23_14 Sch=led[14]

set_property -dict { PACKAGE_PIN V11 IOSTANDARD LVCMOS33 } [get_ports { product[15] }];
#IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

##Buttons

set_property -dict { PACKAGE_PIN P17 IOSTANDARD LVCMOS33 } [get_ports { start }];
#IO_L12P_T1_MRCC_14 Sch=btnl

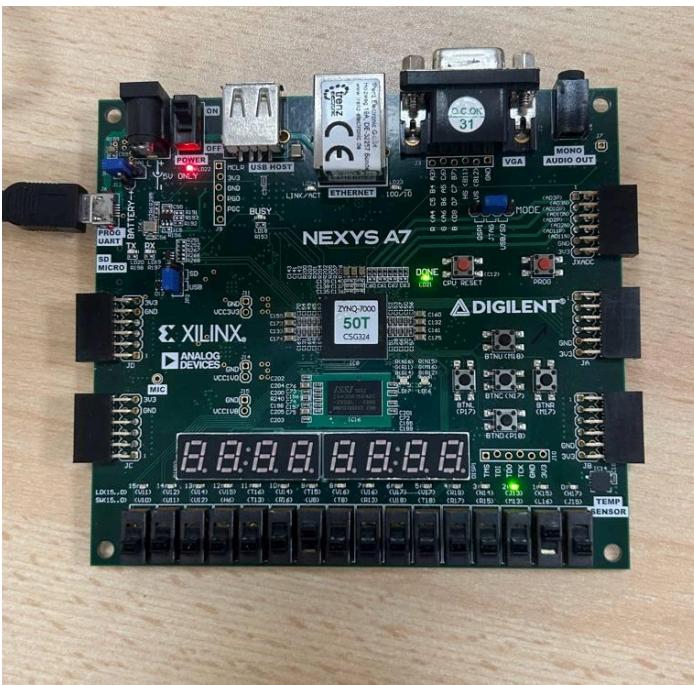
set_property -dict { PACKAGE_PIN M17 IOSTANDARD LVCMOS33 } [get_ports { reset }];
#IO_L10N_T1_D15_14 Sch=btnr

```

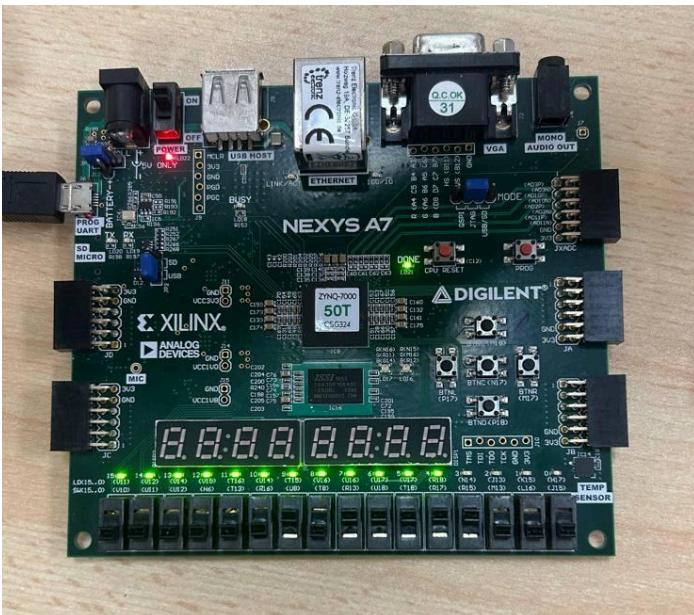
Sintesi su board di sviluppo

Sono state realizzate due moltiplicazioni:

- $2 \times 2 = 4$



- $-8 \times 2 = -16$



Capitolo 4: Comunicazione con handshaking

Esercizio 8 – Comunicazione con handshaking

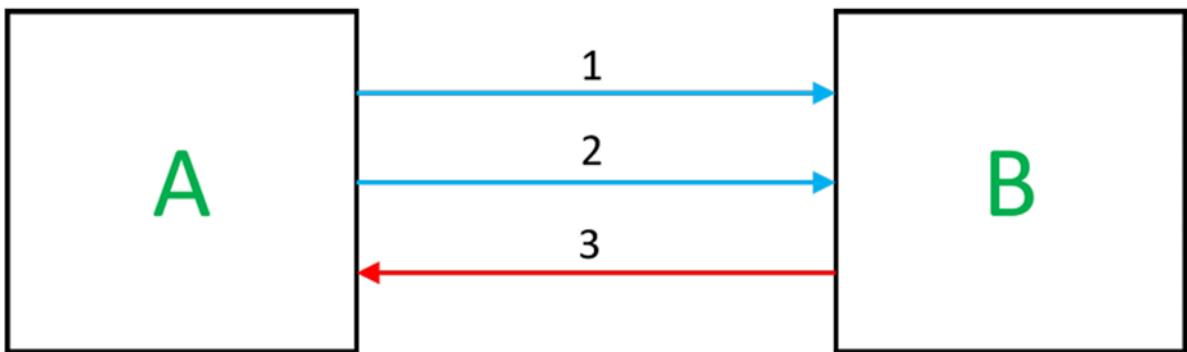
Progettare, implementare in VHDL e testare mediante simulazione un sistema composto da 2 nodi, A e B, che comunicano mediante un protocollo di handshaking. Il nodo A e il nodo B possiedono entrambi una memoria interna in cui sono memorizzate N stringhe di M bit, denominate X(i) e Y(i) rispettivamente

($i=0,..,N-1$). Il nodo A trasmette a B ciascuna stringa $X(i)$ utilizzando un protocollo di handshaking; B, ricevuta la stringa $X(i)$, calcola $S(i)=X(i)+Y(i)$ e immagazzina la somma in opportune locazioni della propria memoria interna.

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

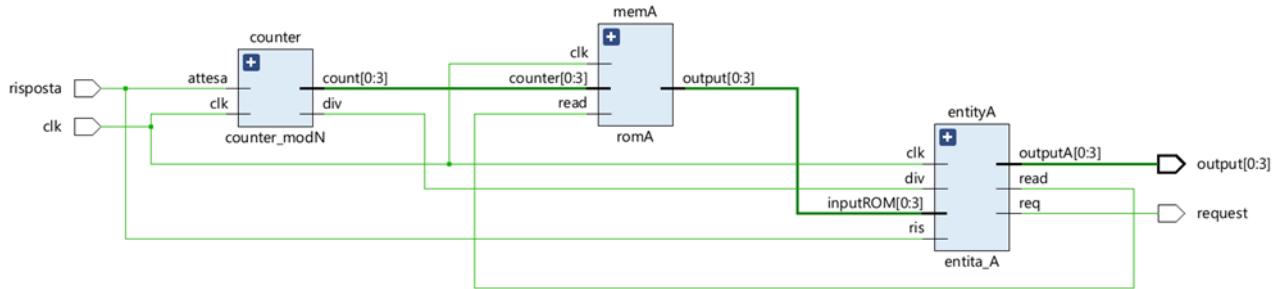
Progetto e architettura

L'obiettivo di questo esercizio è far comunicare due macchine, questo è possibile solo tramite un protocollo. In questo caso è stato applicato un protocollo handshaking (a stretta di mano semplice), consiste nello scambio di 3 segnali, prima di inviare la comunicazione A prepara il dato, successivamente alza la request, B riceve la request e si prepara a ricevere il dato, nel frattempo B alza ACK, A termina la comunicazione e abbassa REQ. Ora B continua con le sue operazioni e solo al termine abbassa ACK. E' stato optato questa scelta in quanto B deve effettuare una serie di operazioni "lunghe", dunque, onde evitare che A trasmettesse e B potesse perdere qualche messaggio si è preferito bloccare A e consentire a B di terminare con le operazioni.



Sistema A

Il sistema A è composto da 3 componenti, il contatore per effettuare il conteggio è utile ai fini dello scambio dei messaggi, se ho inviato meno di 4 messaggi ritorno allo stato iniziale altrimenti ritorno nello stato in cui sono pronto per iniziare la nuova comunicazione. La memoria in cui sono memorizzati gli elementi da trasmettere all'entità B. Infine un'entità A che si occupa di trasmettere il dato e di creare il protocollo handshaking con l'entità B.



Componenti Sistema A

Counter

È stato inserito il contatore perché si differisce leggermente dal contatore inserito nell'appendice.

Il contatore è composto da due stati :

- S0: in questo stato iniziale, si controlla se il segnale di attesa sia alto o basso. Il segnale di attesa non è altro che il segnale di risposta che proviene dal sistema B, se è alto, si procede nello stato s1 altrimenti si permane in S0.
- S1: si controlla se il segnale di attesa è alto, se è alto significa che B sta effettuando operazioni e quindi il conteggio deve fermarsi finché non si abbassa il segnale che notifica la fine dell'operazione e, dunque, la disponibilità a ricevere una nuova stringa.

```

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

entity counter_modN is

    generic(
        N:integer := 8 -- N
    );
    Port ( clk : in STD_LOGIC;
        attesa: in std_logic;
        count: out std_logic_vector(0 to 3):=(others => '0');
        div: out STD_LOGIC:='0'
    );

```

```

end counter_modN;

architecture Behavioral of counter_modN is

type state is (s0,s1);

signal curr: state:= s0;

signal count_tmp: std_logic_vector(0 to 3):=(others => '0');

begin

counter_process: process(clk)
begin
  if(clk' event and clk ='1') then
    case curr is
      when s0 => if(attesa = '0') then
        curr <= s0;
      elsif(attesa = '1') then
        curr <= s1;
      if(count_tmp = 3) then
        div <= '1';
        --count_tmp <= (others => '0');
      else
        div <= '0';
        count_tmp <= count_tmp + 1;
      end if;
    end if;
    when s1 => if(attesa = '1') then
      curr <= s1;
    else

```

```

curr <= s0;

end if;

when others => curr <= s0;

end case;

end if;

end process;

count <= count_tmp;

end Behavioral;

```

MemA

È analoga al componente presente in appendice, l'unica differenza è che sono presenti 4 elementi :

```

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use ieee.numeric_std.all;

entity romA is

  Generic (
    N: integer := 4
  );

  Port (
    read, clk: in STD_LOGIC;
    counter: in STD_LOGIC_VECTOR (0 to 3);
    output : out STD_LOGIC_VECTOR (0 to 3):=(others => '0')
  );
end romA;

architecture Behavioral of romA is

type rom_type is array (0 to N-1) of std_logic_vector(0 to 3);

signal romA: rom_type:={"0000",
                      "0001",

```

```

    "0010",
    "0011");

signal temp: std_logic_vector(0 to 3);

begin

process(clk)

begin

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

    if(read ='1') then

        temp <= romA(TO_INTEGER(unsigned(counter(0 to 3))));

    end if;

end if;

end process;

output <= temp;

end Behavioral;

```

Entità A

L'entità A è il fulcro di questo sistema, in quanto instaura il protocollo Handshaking e consente di trasmettere i dati a B.

E' stato utilizzato un approccio behavioral in cui tramite un automa descriviamo il comportamento della macchina:

- S0: è lo stato iniziale, lo start sarà il div del contatore. Finché div è alto si resta in S0, appena div si abbassa inizia il protocollo; quindi, si transita nello stato S1 e si alza il segnale read per leggere il valore dalla memoria.
- S1: è uno stato di “transizione”, serve per abbassare il segnale di read cosicché non venga letto un nuovo valore al prossimo colpo di clock.
- S2: in questo stato viene salvato il valore letto dalla rom in una variabile temporanea, si alza la richiesta REQ, si transita nello stato S3.
- S3: in questo stato si attende la risposta da B che ha letto il dato e ha terminato le operazioni.
- S4: è un altro stato di transizione in cui si abbassa REQ.
- S5: in questo stato si termina o meno la trasmissione in base al valore di div, se è alto significa che sono stati letti tutti i dati e sono stati trasmessi e si transita nello stato di idle, in caso contrario si transita nello stato S6 e alza read.

- S6: in questo stato si abbassa read, e si controlla se B sta ancora operando, e permane in questo stato attendendo che ACK si abbassi, non appena lo sarà si ritorna in s2 che è lo stato in cui alzo la richiesta per indicare che A è pronto per comunicare.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity entita_A is

    Port (
        ris,div: in std_logic;
        clk:in std_logic;
        inputROM: in std_logic_vector(0 to 3):= (others => '0');
        req: out std_logic;
        read: out std_logic:='0';
        outputA: out std_logic_vector(0 to 3):= (others => '0')
    );
end entita_A;

architecture Behavioral of entita_A is

type state is (s0,s1,s2,s3,s4,s5,s6);

signal curr:state:= s0;
signal reqtmp:std_logic:='0';
signal input_tmp:std_logic_vector(0 to 3):= (others => '0'); -- vettore per salvare dato corrente
begin

process(clk)
begin
    if(clk'event and clk ='1') then
        case curr is
            when s0 =>
                -- stato iniziale attende che div si abbassa per iniziare la trasmissione

```

```

if(div ='1') then

    curr <= s0;

else

    curr <= s1;

    read <='1'; -- attivo il segnale di lettura dalla rom

end if;

when s1 => -- STATO DI TRANSIZIONE

read <= '0';

curr <= s2;

when s2 => -- viene salvato il dato da inviare a B in un vettore temporaneo

input_tmp <= inputROM;

reqtmp <= '1'; -- il dato è pronto ora tengo alto req_tmp fino all'arrivo della risposta.

curr <= s3; -- invio il segnale di richiesta.

when s3 =>

if( ris = '0') then

    curr <= s3; -- rimane in attesa

else -- risposta alta, significa che B ha risposto

    curr <= s4; -- posso andare avanti e abbasso req( lo faccio nello stato successivo)

end if;

when s4 => -- STATO DI TRANSIZIONE

reqtmp <='0';

curr<= s5;

when s5 =>

if( div = '1') then -- div alto : il contatore ha scandito tutta la rom quindi

    curr <= s0;-- torniamo nello stato iniziale

```

```

else

    read <= '1'; -- se ha finito di mandare il messaggio i-esimo

    curr <= s6; -- ma div non è alto vado nello stato successivo e la macchina

end if; -- si appresta a inviare il dato successivo


when s6 =>

    read <='0';

    if(ris='0') then -- Se ris è = 0 read è ancora alto quindi posso tornare indietro, perchè

        curr<=s2;-- B ha risposto e posso andare avanti

    else -- B è alto, sta ancora eseguendo e devo attendere.

        curr<=s6;

    end if;

when others => curr <= s0;

end case;

end if;

end process;

outputA <= input_tmp;

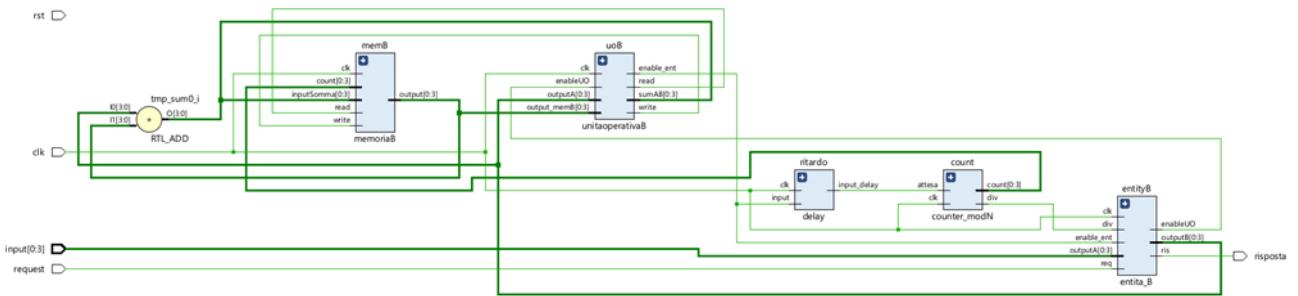
req <= reqtmp;

end Behavioral;

```

Sistema B

Il sistema B si differenzia dal sistema precedente in quanto deve fare anche altre operazioni. Infatti, si occupa di ricevere le stringhe da A e di sommarle con un valore prelevato dalla propria memoria, dunque è stata inserita un'unità operativa che svolge questo compito. Gli altri componenti presenti sono la memoria per prelevare i dati e una volta effettuata la somma memorizzarli all'interno di essa, un componente delay per evitare problemi relativi alla temporizzazione, questo prende in ingresso il segnale di abilitazione per il contatore e in uscita emette un segnale ritardato che andrà in ingresso al contatore.



Componenti Sistema B

Memoria B

Nella memoria di B sono presenti due segnali, **read** per leggere i valori da sommare con i valori trasmessi da A, e **write** per scrivere all'interno della memoria la somma effettuata. Un vettore **InputSomma** che preleva la somma e la salva nella posizione $N/2+1$, poiché i primi $N/2$ valori contengono le stringhe da sommare con A. Un segnale di **count**, che viene gestito grazie al contatore di cui non viene riportata l'implementazione poiché identica a quella utilizzata per il sistema A, la differenza tra i due sta nel segnale che, in A è quello di attesa che permette al contatore di fermare il conteggio aspettando che B abbassi la risposta, mentre attesa, mentre in B il segnale di attesa è stato assegnato come segnale di abilitazione al contatore che l'unità di controllo B dà al contatore.

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

```
entity memoriaB is
generic (
    N:integer := 16
);
Port (
    clk: in std_logic;
    read,write: in std_logic;
    inputSomma: in std_logic_vector(0 to 3);
    count: in std_logic_vector(0 to 3);
```

```

        output: out std_logic_vector(0 to 3)

    );

end memoriaB;

architecture Behavioral of memoriaB is

type mem is array (0 to N-1) of std_logic_vector(0 to 3);

signal mem_tmp: mem:= (
-- PRIMI 8 DA LEGGERE

"0000",
"0001",
"0010",
"0011",
"0000",
"0000",
"0000",
"0000",
others => "0000"      -- QUELLI CHE SARANNO "SOVRASCRITTI"
);

signal valore:std_logic_vector(0 to 3);

begin

process(clk)
begin
if(clk'event and clk = '1') then
    if(read = '1') then
        valore <= mem_tmp(TO_INTEGER(unsigned(count(0 to 3))));
    end if;
end if;
end process;
end;

```

```

elsif(write = '1') then

    mem_tmp((N/2)+TO_INTEGER(unsigned(count(0 to 3)))) <= inputSomma;

end if;

end if;

end process;

output <= valore;

end Behavioral;

```

Unità Operativa

Prima di vedere l'unità di controllo(entity B) è necessario descrivere l'unità operativa solo dopo ha senso creare/vedere l'unità di controllo. L'unità operativa presenta 6 stati :

- S0: Stato iniziale in cui l'unità operativa attende che il segnale di abilitazione per operare sia alzato dall'unità di controllo, non appena si alza si passa allo stato successivo disabilitando l'unità di controllo di B, questo per far sì che né A, né B possano procedere con la comunicazione e lo scambio di messaggi fintantoché l'operazione di somma non si conclude.
- S1: in questo stato B inizia ad “effettuare le operazioni”, infatti si abilita la lettura della memoria di B per prelevare un nuovo valore che verrà sommato.
- S2: stato di transizione per abbassare il segnale di lettura.
- S3: in questo stato ho pronto i due dati per effettuare la somma; quindi, posso abilitare l' unità di controllo e continuare con le operazioni.
- S4: in questo stato effettuo la somma, abilito la scrittura in memoria e disabilito nuovamente l'unità di controllo onde evitare conflitti. Quindi in questo momento si ferma l'unità di controllo in un particolare stato.
- S5: In questo stato ho terminato, ho scritto in memoria, attendo un nuovo segnale di abilitazione da parte dell'unità di controllo, se è basso resto in attesa nello stato s5, altrimenti si prepara una nuova somma transitando nello stato s1 (Non si torna mai in S0).

```

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

entity unitaoperativaB is

    Port (

```

```

clk:in std_logic;
outputA: in std_logic_vector(0 to 3):= (others => '0');
enableUO: in std_logic:='0';
enable_ent: out std_logic:='0';
write,read: out std_logic:='0';
output_memB: in std_logic_vector(0 to 3):= (others => '0');
sumAB: out std_logic_vector(0 to 3)

);

end unitaoperativaB;

```

architecture Behavioral of unitaoperativaB is

type state is (s0,s1,s2,s3,s4,s5);

signal curr:state:=s0;

signal tmp_read,tmp_write: std_logic:='0';

signal tmp_sum: unsigned(0 to 3):= (others => '0');

signal a,b: unsigned(0 to 3);

begin

process(clk)

begin

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

case curr is

when s0 =>

if(enableUO = '0') then -- Finchè l'enable è = 0 resto nello stato iniziale

curr<=s0; -- (FINCHÈ NON ARRIVA REQ IN RETE DI CONTROLLO B)

else

```
curr <= s1;

enable_ent <= '0'; -- DISABILITI L UNITÀ DI CONTROLLO IN MANIERA CHE
-- A E B NON POSSANO COMUNICARE COSICCHÈ HO TEMPO PER ESEGUIRE LA SOMMA.

end if;
```

when s1 => -- STATO DI TRANSIZIONE

```
tmp_read <= '1'; -- abilito la lettura della memoria

curr <= s2;
```

when s2 =>

```
tmp_read <= '0'; -- ho letto il dato abbasso il segnae

curr <= s3;
```

when s3 =>

-- MI PREPARO PER EFFETTUARE LA SOMMA

```
a <= unsigned(outputA);

b <= unsigned(output_memB);

curr <= s4;

enable_ent <= '1'; -- ABILITÒ L ENTITÀ di controllo B per gestire nuove richieste
```

when s4 =>

-- Effettuò la somma e scrivo in memoria

```
tmp_sum <= a+b;

tmp_write <= '1'; -- SALVO IL VALORE

enable_ent <= '0'; -- Disabilita nuovamente l'unità di controllo per evitare conflitti

curr <= s5;
```

```

when s5 =>

    tmp_write <= '0'; -- HO LETTO QUINDI SPENGO IL SEGNALE

    if(enableUO='0')then -- SE L'UNITÀ OPERATIVA è DISABILITÀ

        curr <= s5;

    else

        curr <= s1;

    end if;

when others => curr <= s0;

end case;

end if;

end process;

read <= tmp_read;

write <= tmp_write;

sumAB <= std_logic_vector(tmp_sum(0 to 3));

end Behavioral;

```

Entity B

L'unità di controllo, presenta un segnale REQ che riceve dal sistema A, in uscita presenta un segnale RIS, ovvero di risposta (ACK) per confermare l'inizio della comunicazione tra le 2 entità. Oltre i segnali sopra citati, vi è un segnale di div che riceve dal proprio contatore, la stringa in ingresso che proviene dalla ROM A mediante handshaking, la stringa di B che preleva direttamente dalla memoria e un segnale di abilitazione per l'UO al fine di effettuare la somma. Per avere una chiara interpretazione di come i segnali vengono gestiti, anche qui viene definito un automa:

- S0: stato iniziale in cui si attende la richiesta dal sistema A. Non appena arriva la richiesta sarà abilitata la UO e si alza la risposta per far sì che A si fermi aspettando che B abbia finito.
- S1: stato di controllo sulla UC stessa, se basso si rimane in s1 attendendo che la UO la abiliti, altrimenti si va in s2 disabilitando la UO.
- S2: in questo stato l'UO avrà effettuato la somma, si verifica se il segnale di REQ mandato da A è ancora alto, nel caso sia alto si attende in S2, altrimenti si avanza in S3.

- S3: stato di controllo del div per verificare se si è raggiunti il numero massimo di conteggio. Si prosegue verso lo stato successivo se non si è raggiunto il numero di stringhe da inviare.
- S4: Analogo allo stato s0. Esso è uno stato “aggiuntivo” che è stato introdotto poiché, dopo aver controllato il segnale di div, il dato viene elaborato e si passa nello stato di s1 per effettuare una nuova somma. Quindi se si arriva nello stato di s4 non si tornerà più in s0, al più si potrà permanere nello stato corrente fintantoché A non sarà pronto ad una nuova trasmissione. Quindi una volta trasmesso il primo dato, attraverso questo tipo di automa si intende che non si tornerà mai più nello stato di s0 a meno di un segnale di div alto.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity entita_B is
Port (
    req,div,clk: in std_logic;
    enable_ent: in std_logic:='0';
    outputA: in std_logic_vector(0 to 3):= (others =>'0');
    outputB: out std_logic_vector(0 to 3):= (others =>'0');
    enableUO: out std_logic:='0';
    ris: out std_logic
);
end entita_B;

```

```

architecture Behavioral of entita_B is
type state is(s0,s1,s2,s3,s4);
signal curr:state:=s0;
signal tmpinput:std_logic_vector(0 to 3):= (others =>'0');
signal enableUO_tmp,ris_tmp: std_logic:='0';

begin

```

```

process(clk)
begin
  if(clk'event and clk ='1') then
    case curr is
      when s0 =>
        -- Attendo il segnale di inizio comunicazione
        if(req='0') then
          curr <= s0;
        else
          curr <= s1; -- INIZIO LA COMUNICAZIONE E ABILITò I unità operativa
          enableUO_tmp <= '1';
          ris_tmp<= '1'; -- AVVISO A LA RICEZIONE DEL MESSAGIO
        end if;
      when s1 =>
        if(enable_ent='0') then -- ora I unità operativa sta effettuando i calcoli e mi tiene bloccato
          curr <= s1;
        else
          enableUO_tmp <= '0'; -- I unità operativa ha finito e posso andare avanti
          curr <= s2;
        end if;
      when s2 =>
        -- verifica se la richiesta di A è ancora attiva
        if (req='0') then

```

```

    ris_tmp <= '0'; -- disabilita la risposta ( il dato è stato elaborato)

    curr <= s3;

    else -- altrimenti attendo

    curr <= s2;

    end if;

when s3 =>

    -- verifica se il ciclo di trasmissione è terminato.

    if(div='0') then

        curr <= s4; -- Se non è finito, passa allo stato s4 per elaborare il prossimo dato

        else

            curr <= s0; -- Se il ciclo è completato, torna allo stato iniziale

        end if;

when s4 =>

    -- Attende una nuova richiesta di A per riprendere la comunicazione

    if(req='0') then

        curr <= s4; -- Rimane in attesa

        else

            curr <= s1; -- Riparte la comunicazione

            enableUO_tmp<= '1'; -- Riattiva l'unità operativa

            ris_tmp<= '1';-- Riattiva il segnale di risposta

        end if;

when others => curr <= s0;

end case;

end if;

end process;

ris <= ris_tmp;

enableUO <= enableUO_tmp;

```

```
outputB <= outputA;
```

```
end Behavioral;
```

Simulazione

Per effettuare il testbench è stato necessario lasciare che la macchina evolvesse, avendo inserito nell'unità di controllo di A che “lo start” è rappresentato dalla presenza del div alto o meno, non necessitiamo di alzare nessun bit.

```
library IEEE;  
  
use IEEE.STD_LOGIC_1164.ALL;  
  
use work.all;  
  
--use IEEE.NUMERIC_STD.ALL;  
  
entity B_tb is  
  
-- Port ( );  
  
end B_tb;
```

```
architecture Behavioral of B_tb is
```

```
component A  
  
Port (  
  
clk: in std_logic;  
  
risposta: in std_logic;  
  
request: out std_logic;  
  
output: out std_logic_vector(0 to 3)  
  
);
```

```
end component;
```

```
component B
```

```
Port ( clk,rst: in std_logic;  
  
request: in std_logic;
```

```
    input: in std_logic_vector(0 to 3);  
  
    risposta: out std_logic  
  
 );  
  
end component;
```

```
constant CLK_periodA : time := 20ns;
```

```
constant CLK_periodB : time := 10ns;
```

--ricorda: il limite di frequenza sta nel contatore, ovvero il segnale di ris deve essere alto per almeno un colpo di clock del sistema A, altrimenti il contatore non lo vede e non si incrementa

```
signal rst,r,clkA,clkB,ris : STD_LOGIC := '0';
```

```
signal data : STD_LOGIC_VECTOR(0 to 3);
```

```
begin
```

```
    entita1: A port map(clk => clkA, request => r, risposta => ris, output => data);
```

```
    entita2: B port map(clk => clkB, rst => rst, request => r, risposta => ris, input => data);
```

```
clkA_proc: process
```

```
begin
```

```
    clkA <= '0';
```

```
    wait for CLK_periodA/2;
```

```
    clkA <= '1';
```

```
    wait for CLK_periodA/2;
```

```
end process;
```

```
clkB_proc: process
```

```
begin
```

```

clkB <= '0';

wait for CLK_periodB/2;

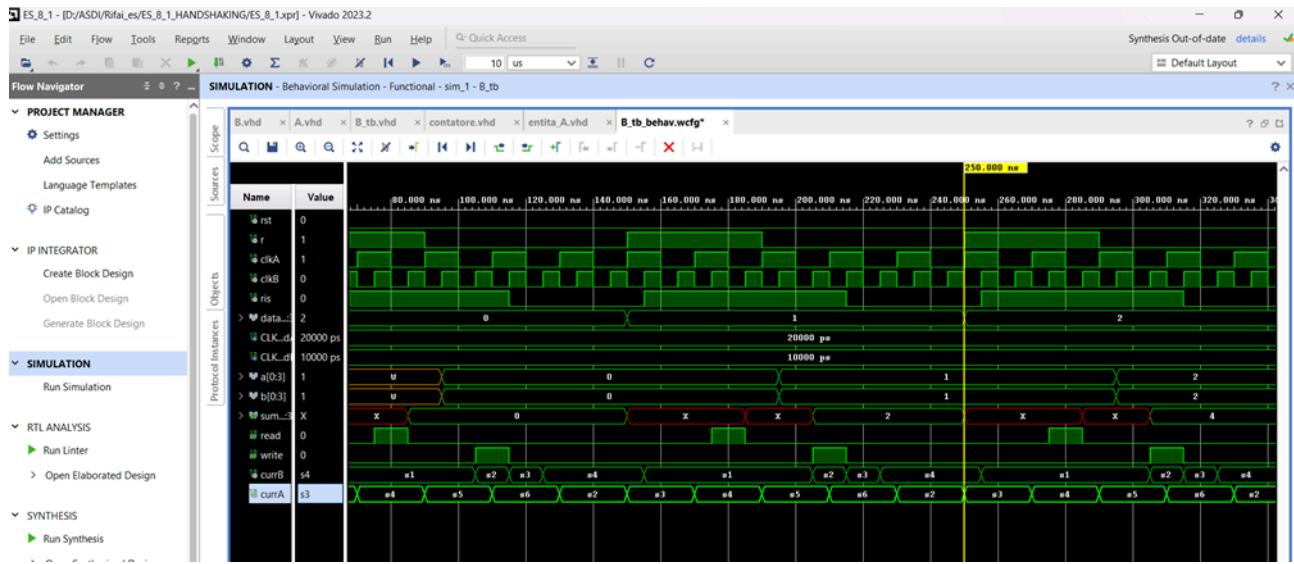
clkB <= '1';

wait for CLK_periodB/2;

end process;

```

end Behavioral;



Capitolo 5: Processore

Esercizio 9.1

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

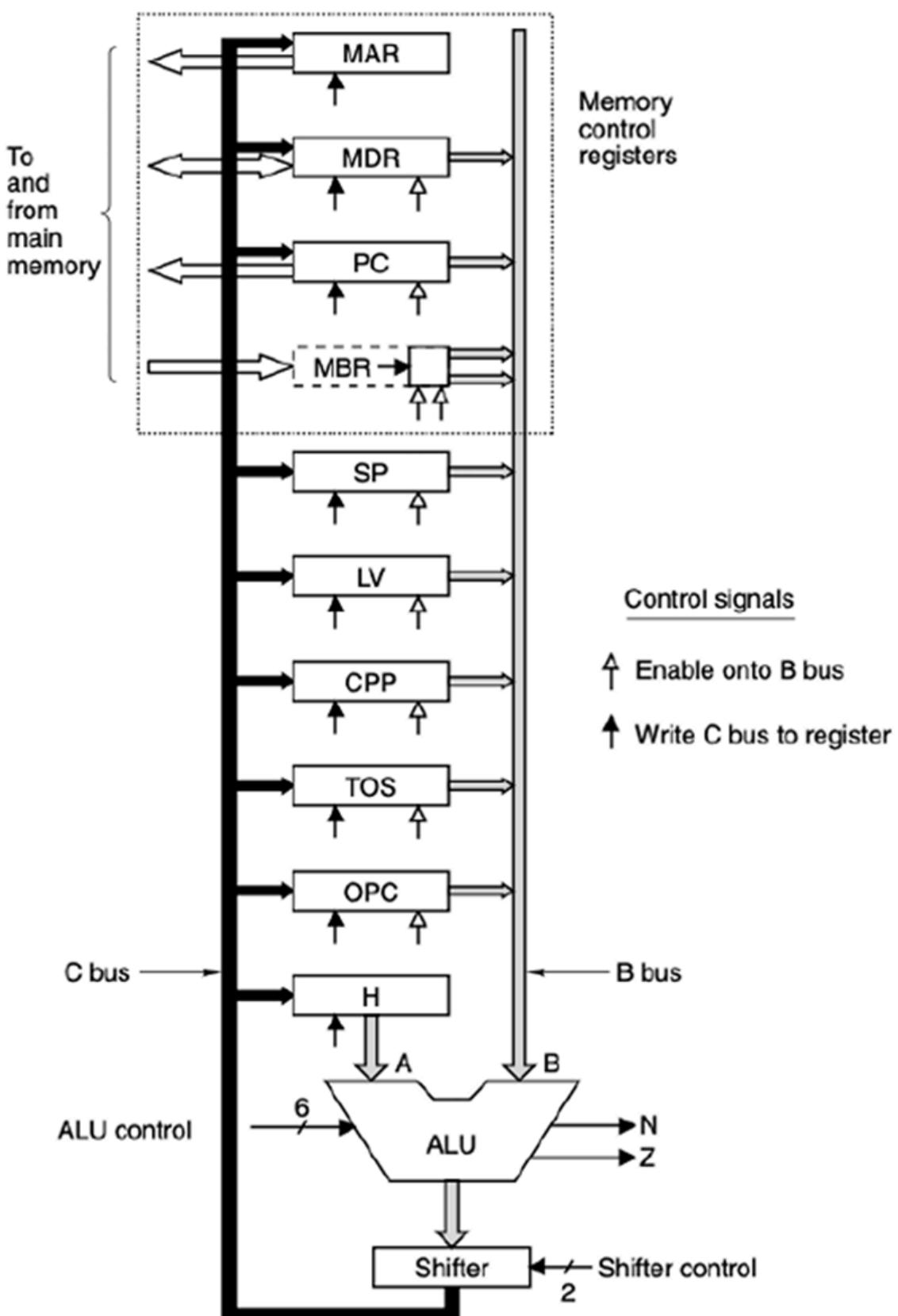
- a) si proceda all'analisi dell'architettura mediante simulazione e si approfondisca lo studio del suo funzionamento per due istruzioni a scelta,
- b) si modifichi un codice operativo a scelta, documentando tutte le modifiche effettuate

Processore MIC-1

I processore MIC-1 è implementato in logica microprogrammata, ossia ciascuna istruzione in linguaggio IJVM è composta da una sequenza di microistruzioni che compongono quindi il microprogramma. Quest'ultimo è memorizzato a sua volta in una ROM che si trova all'interno del processore. Il processore in questione è strutturato nel seguente modo:

- Unità operativa
- Unità di controllo

Unità operativa:



La parte operativa, come mostrato in figura, contiene registri (32 bit), bus B e C, ALU e shift register. Ogni registro ha una coppia di segnali di controllo che permettono di raccogliere e memorizzare ciò che gli arriva dal bus C oppure di mettere sul bus B il loro contenuto.

Tra i registri troviamo:

- Registri dell'interfaccia con la memoria:
 - o MAR - memory address register;
 - o MDR - memory data register;
 - o PC - program counter;
 - o MBR - memory byte register;
- Registro che mantiene il primo operando dell'ALU:
 - o H - holding.
- Altri registri utili:
 - o SP - stack pointer;
 - o LV - local variables;
 - o CPP - constant pool pointer;
 - o TOS - top of stack;
 - o OPC - scratch register.

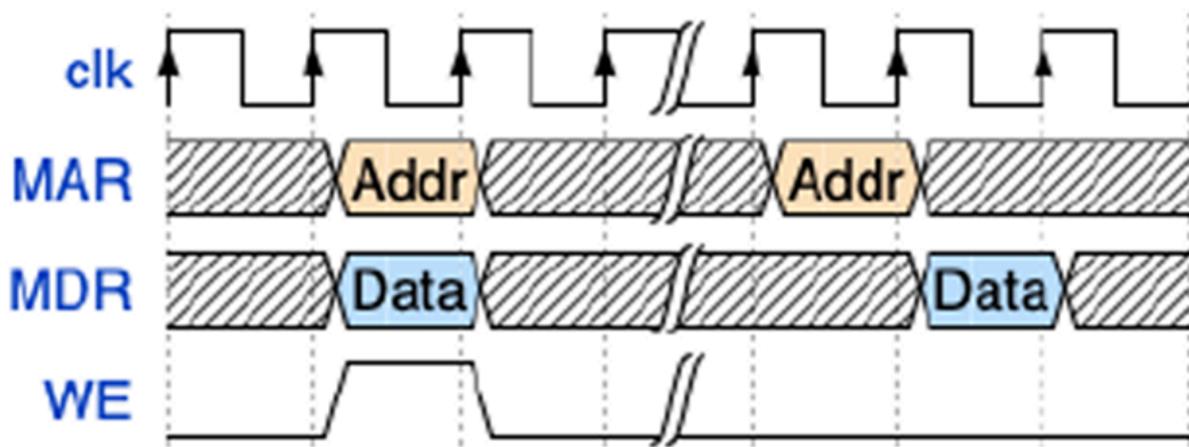
L'ALU del processore invece si occupa di compiere le operazioni aritmetiche richieste. Essa ha 2 ingressi A e B. Il primo è collegato al registro H e il secondo al bus B. Per conoscere quale informazione deve compiere, la ALU fa riferimento a un segnale su 6 bit che fa da selezione. A ogni operazione, quindi, sarà associata la propria stringa di 6 bit.

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

Per interfacciarsi con la memoria, il processore utilizza due interfacce:

- MAR e MDR: controllano un'interfaccia a 32 bit in lettura e scrittura. Questa interfaccia è utilizzata per accedere ai dati su cui operano le istruzioni IJVM (Constant Pool, Local Variable, Stack degli operandi). In questo modo specifico da quale indirizzo di memoria di MAR partire per leggere o scrivere e mettere il risultato in MDR.
- PC e MBR: controllano un'interfaccia a 8 bit in sola lettura. Questa interfaccia è utilizzata per prelevare le istruzioni IJVM dalla memoria (Method Area). In questo modo specifico da quale indirizzo di memoria di PC partire per leggere e mettere il risultato in MBR.

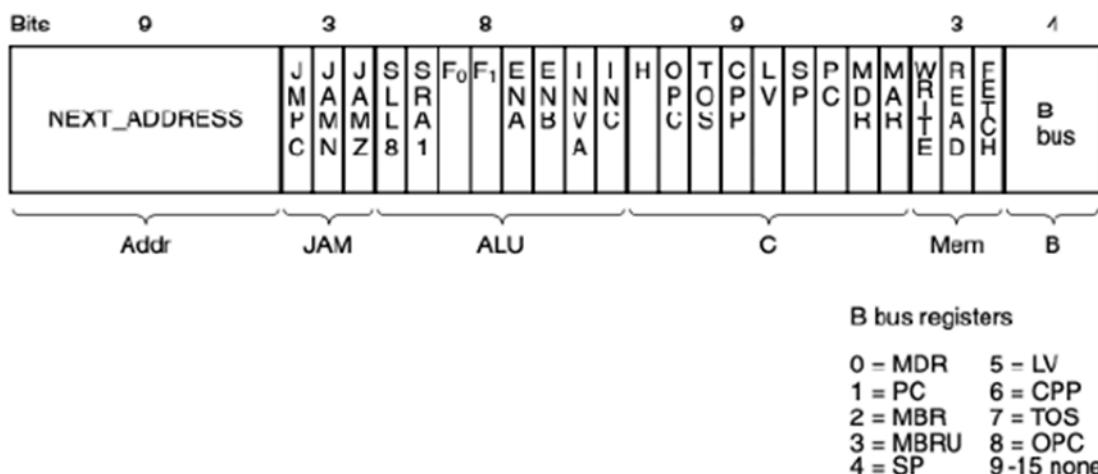
In fase di scrittura l'interfaccia MAR-MDR, grazie a un segnale di Write Enable (WE), riesce sullo stesso fronte di salita del clock sia ad acquisire il dato che a scriverlo in memoria. In fase di lettura invece le interfacce hanno un funzionamento simile, ossia viene caricato nel registro MAR l'indirizzo da cui si vuole leggere (ossia quello contenuto nel PC). Al fronte successivo del clock viene inserito quindi il dato nel registro MDR.



Le microistruzioni:

Le microistruzioni sono rappresentate su 36 bit. Ognuna di esse quindi comprende vari campi, ognuno con dei segnali per abilitare vari registri:

- Addr: indirizzo di una potenziale prossima microistruzione;
- JAM: determina come è selezionata la prossima microistruzione;
- ALU: controllo dell'ALU e dello shifter;
- C: controlla quali registri vengono scritti dal bus C;
- Mem: controlla le operazioni di memoria;
- B: seleziona a seconda del suo valore il registro connesso al bus B



Si nota come in ogni microistruzione è contenuto il campo NEXT ADDRESS tramite il quale ogni microistruzione si concatena alla successiva. In questo modo, anche se in memoria le microistruzioni non

sono memorizzate in celle di memoria contigue, si riesce comunque a trovare la prossima microistruzione da eseguire. A tal proposito sono utili i bit di salto:

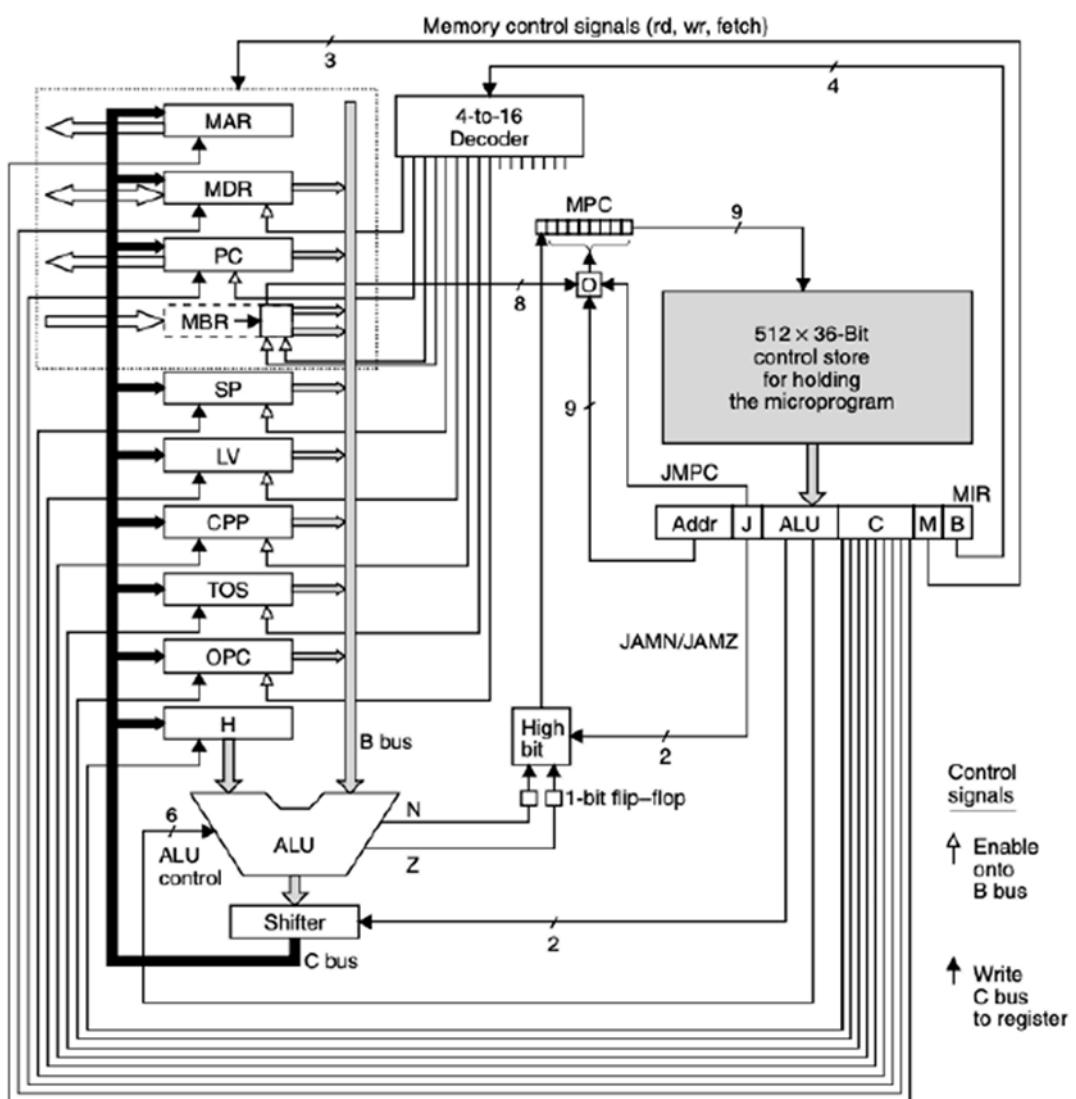
- JAMN = 1: il bit più significativo di un microPC (contenuto nell'unità di controllo) viene messo in OR con il flag N dell'ALU (risultato negativo in uscita).
- JAMZ = 1: il bit più significativo viene messo in OR con il flag Z dell'ALU (risultato nullo in uscita).
- JAMC = 1: gli 8 bit meno significativi di NEXT ADDRESS sono messi in OR con MBR. In questo modo ottengo un salto di un valore pari a quello contenuto in MBR.

Se invece i 3 bit di salto JAM sono tutti 0, MPC sarà uguale semplicemente a NEXT ADDRESS.

Importante è però assicurarsi che la temporizzazione tra le istruzioni sia adeguata, in tal senso bisogna assicurarsi che prima di caricare la prossima microistruzione sia disponibile un nuovo microPC e che i ritardi dovuti ai vari componenti che costituiscono il processore non influiscano sul funzionamento. Scrivere a mano le microistruzioni è possibile, ma sarebbe difficile e potrebbe portare a degli errori, oltre a rendere il programma difficilmente modificabile. Per questo si utilizza il linguaggio detto MAL (MicroAssembly Language) per semplificare la scrittura del microprogramma.

Unità di Controllo

L'unità di controllo è costituita dal Control Store, ossia una memoria di sola lettura che contiene le microistruzioni del microprogramma. L'accesso a questa memoria è gestito da un'interfaccia tra i registri MPC (MicroPC) e MIR (MicroInstruction Register). Operando in questo modo, la control unit genera a ogni colpo di clock lo stato di tutti i segnali di controllo e l'indirizzo della prossima microistruzione da eseguire (indicato dal mPC). Ricapitolando, il flusso di esecuzione di un microprogramma è il seguente: inizialmente si effettua la traduzione da assembler a microassembler e in seguito da microassembler a MAL. In questo modo si riesce ad arrivare a delle parole di controllo, note al MIC-1. Dopo ciò, le microistruzioni vengono inserite nella Control Store e sono pronte ad essere eseguite.



Esercizio IADD e POP (Analisi e simulazione)

IADD

L'istruzione IADD

iadd = 0x65:

MAR = SP = SP - 1; rd

H = TOS

MDR = TOS = MDR + H; wr; goto main

L'istruzione è eseguita in 3 cicli di clock:

- MAR = SP = SP-1; rd : si occupa di decrementare il valore dello stack pointer e memorizzare l'indirizzo nel MAR.
- H=TOS : si occupa di salvare il valore del TOS all'interno del registro di appoggio.
- MDR = TOS = MDR+H : effettua la somma dell'elemento che è stato caricato nel MDR(l'elemento puntato dal MAR) con l'elemento memorizzato in H, infine salva il valore sia su TOS che sul registro MDR.
- Goto main : alla fine di questa micro-istruzione, il processore salta alla parte principale del programma, indicando che l'operazione è completata e che il processore può passare a eseguire la successiva istruzione del programma.

Analizzando il codice mal notiamo che il codice si trova all'indirizzo 0x65 codificato in decimale 101, andando a controllare le istruzioni presenti nel control store possiamo notare che alle seguenti micro-istruzioni corrispondono le seguenti control-word.

- . Microistruzione 1: MAR = SP = SP - 1; rd
 - next address: 001100110
 - ○ jmp: 000
 - ○ alu: 00110110
 - ○ bus C: 000001001
 - ○ mem: 010
 - ○ bus B: 0100
- Microistruzione 2: H = TOS
 - next address: 001100111
 - jmp: 000
 - alu: 00010100
 - bus C: 100000000
 - mem: 000
 - bus B: 0111
- Microistruzione 3: MDR = TOS = MDR + H; wr; goto main
 - next address: 000000110
 - jmp: 000
 - alu: 00111100
 - bus C: 001000010
 - mem: 100
 - bus B: 0000

Simulazione IADD

Programma :

.main

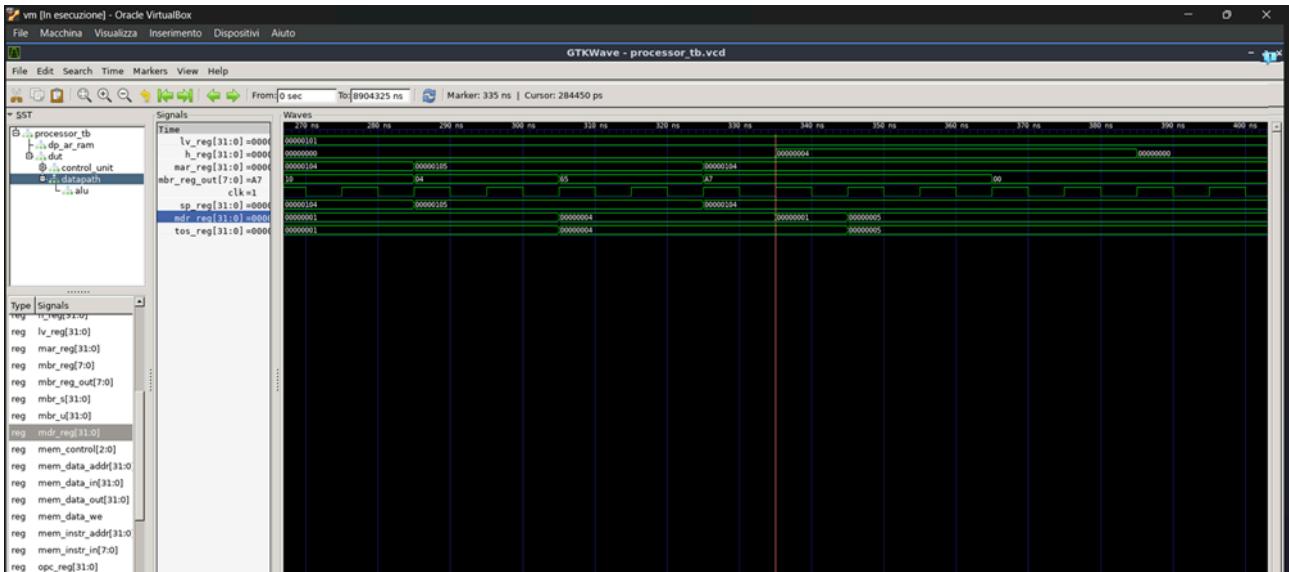
BIPUSH 0x01

BIPUSH 0x04

IADD

HALT

.endmethod



Come si può notare dalla figura, quando viene eseguita l'istruzione IADD (0x65) viene estratto 0x1 che va nel Registro TOS e 0x4 che va nel registro H. A questo punto viene effettuata la somma tra H e TOS e il risultato è 0x05.

POP

La seconda istruzione analizzata è la POP, la quale permette di rimuovere l'ultimo elemento inserito nello stack. Di seguito è riportato il codice dell'istruzione:

pop = 0x59:

MAR = SP = SP-1; rd

empty

TOS = MDR; goto main

- Il MAR (MemoryAddressRegister) viene impostato uguale allo stack pointer (SP) e poi lo SP viene decrementato di 1. Viene eseguita un'operazione di lettura dalla memoria (rd), che porta il valore dalla posizione dello stack pointer (SP) al registro MDR (Memory Data Register);
- Questa riga indica che lo stack è vuoto (empty), il che potrebbe essere verificato se non ci sono dati da estrarre dallo stack. Nel contesto di un'operazione di pop, questo potrebbe indicare che non c'è nessun dato da estrarre dallo stack;
- Il valore letto dalla memoria (che è stato precedentemente nel lo stack) viene memorizzato nel registro TOS (Top of Stack). Quindi, il valore estratto dallo stack (il vecchio TOS) viene memorizzato in TOS per un utilizzo successivo.

Simulazione POP

Per simulare l'istruzione abbiamo utilizzato il seguente programma:

. main

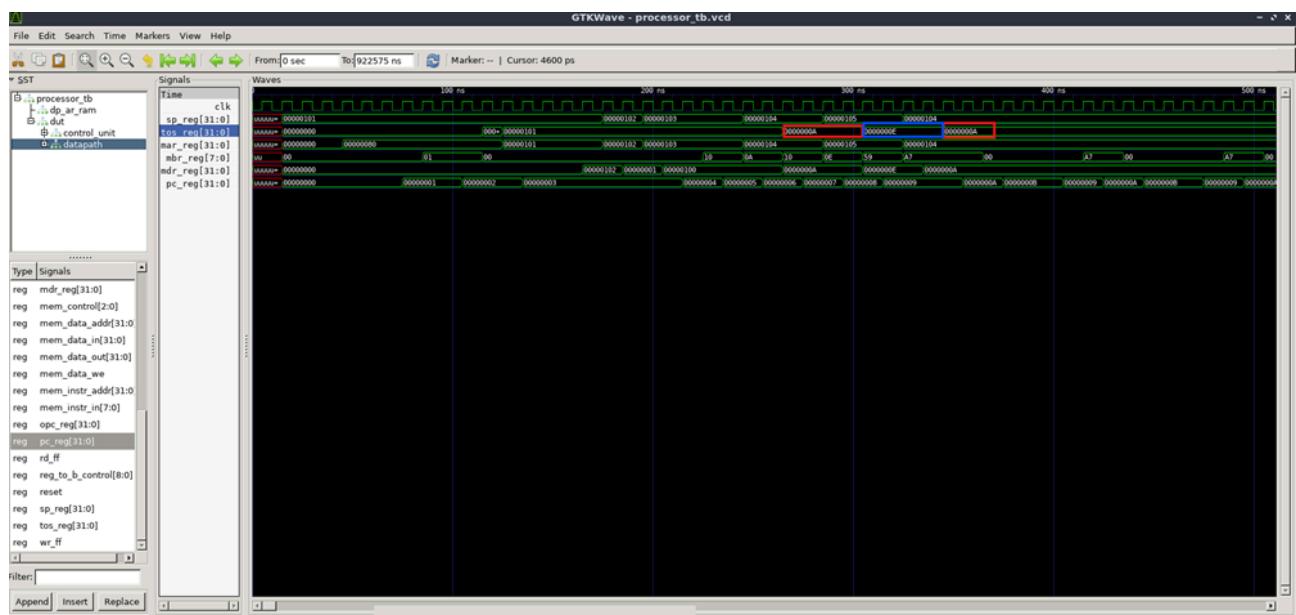
BIPUSH 0xA

BIPUSH 0xE

POP

HALT

.endmethod



Come si evince dal grafico della simulazione, in un primo momento vengono caricati (attraverso BIPUSH) sullo stack i due valori, rispettivamente, 0xA e 0xE. Prima dell'esecuzione di POP si nota che in TOS è presente il valore 0xE poiché è l'ultimo ad essere stato caricato sullo stack mentre dopo che la POP viene eseguita in TOS compare il valore 0xA.

IADD modificata

Il codice operativo che è stato scelto per la modifica è IADD. In particolare, come riportato nel codice di seguito, è stata effettuata una modifica che coinvolge il registro H, il quale viene incrementato ulteriormente di uno durante lo svolgimento dell'istruzione.

ajvm.mal

processor_tb.vhd

```
# MIC-1 Microprogram
# Copyright (C) 2019 Alberto Moriconi

        goto mic1_entry

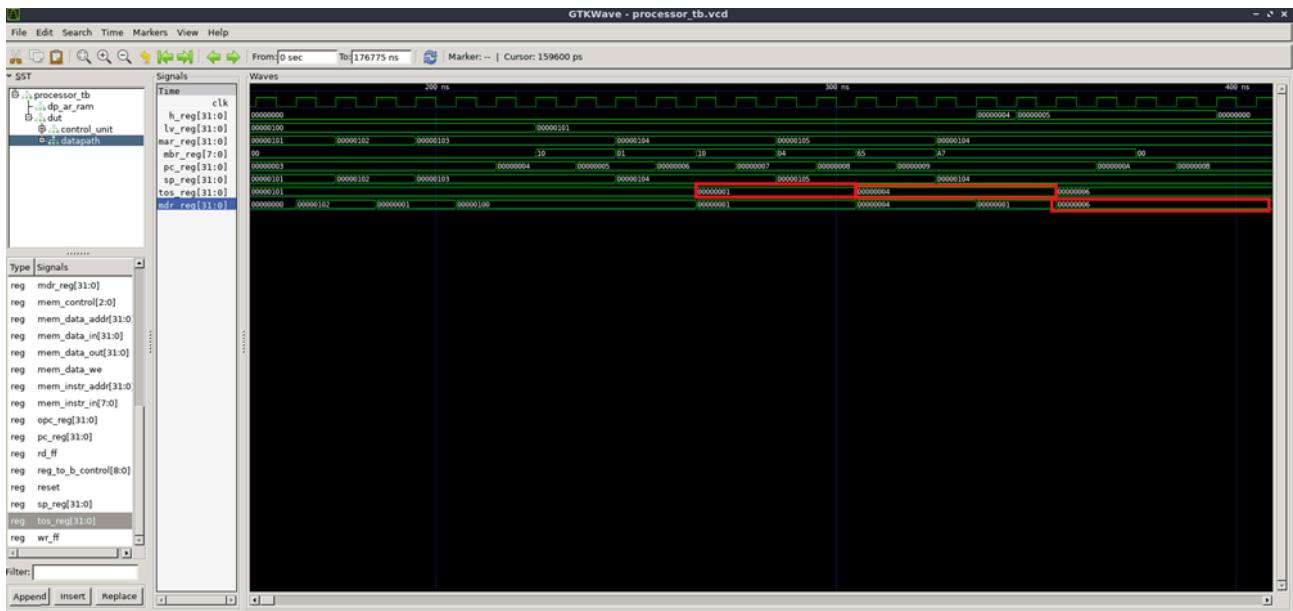
main:
        PC = PC + 1; fetch; goto (MBR)

nop = 0x00:
        goto main

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

Per simulare la IADD da noi modificata è stato usato il seguente programma:

```
.main
    BIPUSH 0x1
    BIPUSH 0x4
    IADD
    HALT
.endmethod;
```



Come si evince dal grafico della simulazione, prima dell'operazione IADD vengono caricati sullo stack i valori 0x1 e 0x4, cioè 1 e 4 in esadecimale. Dopo l'esecuzione dell'operazione in TOS non compare il valore 5 ma il valore 6 a causa dell'incremento del registro H citato precedentemente.

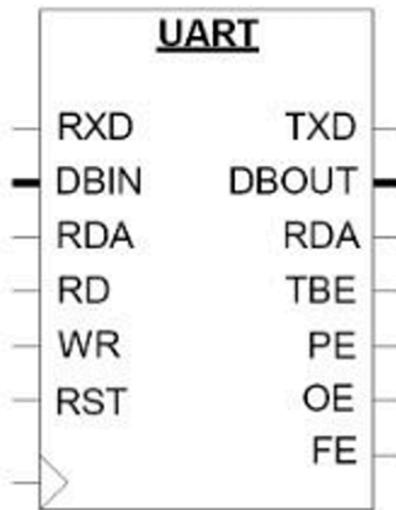
Capitolo 6: Interfaccia Seriale

Esercizio 10.1

Partendo dall'implementazione fornita dalla Digilent di un dispositivo UART-RS232 (componente RS232RefComp.vhd), progettare, implementare e simulare in VHDL un sistema composto da 2 unità A e B che condividono lo stesso segnale di clock e comunicano tra loro mediante interfaccia seriale. Il sistema A contiene una ROM di 8 locazioni da 1 byte ciascuno, un contatore CONT_A per scandire le locazioni della ROM e una UART_A, mentre il sistema B contiene una memoria MEM di 8 locazioni da 1 byte ciascuno, un contatore CONT_B per scandire le locazioni della MEM e una UART_B. Quando un segnale WR viene asserito nell'unità A, viene prelevato un byte dalla ROM e inviato all'unità B, che dovrà riceverlo e salvarlo in MEM.

Progetto e architettura

L'UART (Universal Asynchronous Receiver-Transmitter) è un componente capace di convertire dei flussi di bit da serie a parallelo o viceversa. Per fare ciò, al suo interno è presente uno shift register, capace di effettuare tale conversione.



L'UART presenta in ingresso, dal lato trasmettitore, DBIN e WR. Il primo ingresso ospita i dati da trasmettere in parallelo, mentre il secondo è un segnale di write che dice quando si può iniziare a trasmettere. Come uscite invece il componente presenta TXD e TBE. TXD è l'uscita che presenta i dati in modo seriale, TBE invece serve ad indicare quando il buffer di trasmissione è libero (1) o occupato (0). Lato ricevitore invece si trovano gli ingressi RXD, RD e RDA. RXD ospita i dati serie in ingresso, RD è un segnale di read che avvia la lettura del dato presente in RXD, mentre RDA è un altro segnale che indica la disponibilità di un nuovo dato in ingresso (questo segnale è presente anche in uscita). Per quanto riguarda le uscite, oltre a RDA, l'UART presenta DBOUT e dei bit di errore. La prima uscita presenta i dati ricevuti in formato parallelo mentre i bit di errore si dividono in 3 tipologie:

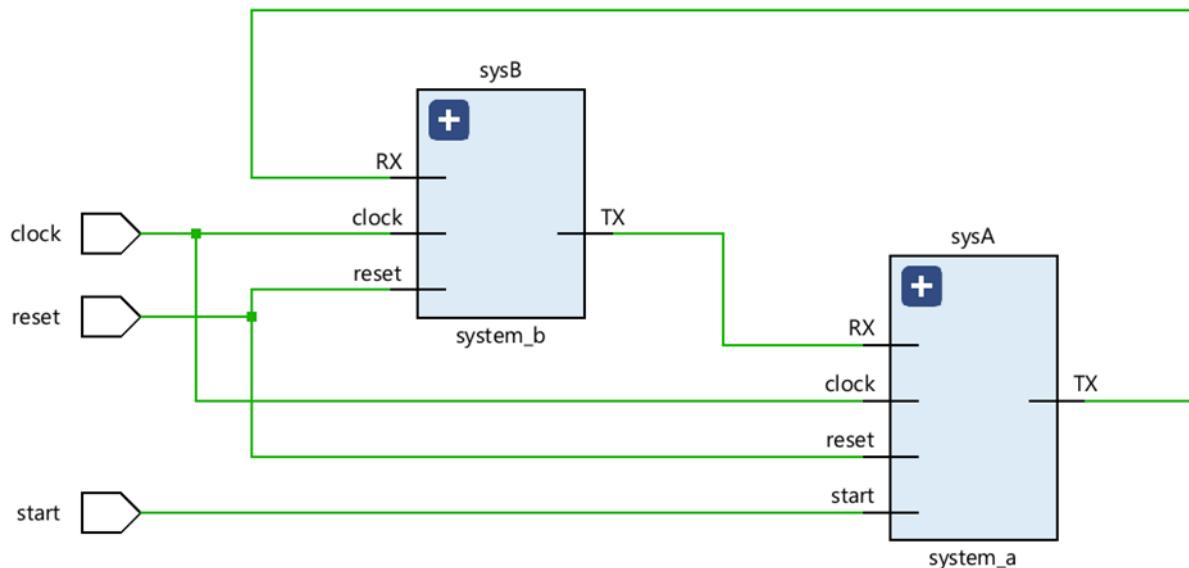
- PE: Parity error, si presenta quando il bit di parità (che viene trasmesso assieme agli altri) rileva un errore nella trasmissione.
- OE: Overrun error: si verifica quando il trasmettitore ha già inviato un nuovo dato ma in ricevitore non ha ancora terminato l'elaborazione del precedente. In questo modo il nuovo dato viene perso.

- FE: Farming error: si presenta quando non viene rilevato il bit di stop e quindi è notifica che c'è stato un errore precedentemente nel campionamento del segnale.

Per effettuare la trasmissione, l'UART trasmette tramite un protocollo che prevede un bit di start, 8 di dato, un bit di controllo (parità) e uno di stop. Quando non si trasmette il segnale RDX è a riposo (1), appena si vuole iniziare la trasmissione il segnale si abbassa e poi si inizia. Quando si vorrà terminare la trasmissione si alza quindi il segnale. Bisogna però assicurarsi che il campionamento dei segnali trasmessi sia effettuato a dovere dal ricevitore, per questo esso lavora con un clock a frequenza 16 volte maggiore rispetto al trasmettitore. In questo modo si riesce a campionare il valore al centro e si è sicuri di prendere il valore giusto e di conseguenza tutti i valori disponibili.

Implementazione

Sistema complessivo



```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity system is
    Port (
        start : in STD_LOGIC;
        reset : in STD_LOGIC;
        clock : in STD_LOGIC);
end system;

```

architecture Structural of system is

component system_a is

```
Port ( start : in STD_LOGIC; -- Start esterno  
       reset : in STD_LOGIC; -- Reset esterno  
       clock : in STD_LOGIC; -- Clock  
       RX : in std_logic; -- Segnale di ricezione, ricevuto da system_b  
       TX : out std_logic); -- Segnale di invio, da mandare a system_b  
end component;
```

component system_b is

```
Port ( clock : in STD_LOGIC; -- Clock di sistema  
       reset : in STD_LOGIC; -- Reset esterno  
       RX : in STD_LOGIC; -- Segnale di ricezione, da mandare a system_a  
       TX : out STD_LOGIC; -- Segnale di invio, da ricevere da system_a  
       error : out STD_LOGIC; -- Segnale che avvisa della presenza di errori  
       data_out : out STD_LOGIC_VECTOR (7 downto 0)); -- Dato salvato in memoria  
end component;
```

signal sig_A : STD_LOGIC;

signal sig_B : STD_LOGIC;

signal sig_ERR : STD_LOGIC;

begin

sysA : system_a port map(

```

start => start,
reset => reset,
clock => clock,
RX => SIG_A,
TX => SIG_B);

sysB : system_b port map(
    clock => clock,
    reset => reset,
    RX => SIG_B,
    TX => SIG_A,
    error => SIG_ERR,
    data_out => open);

end Structural;

```

Sistema A

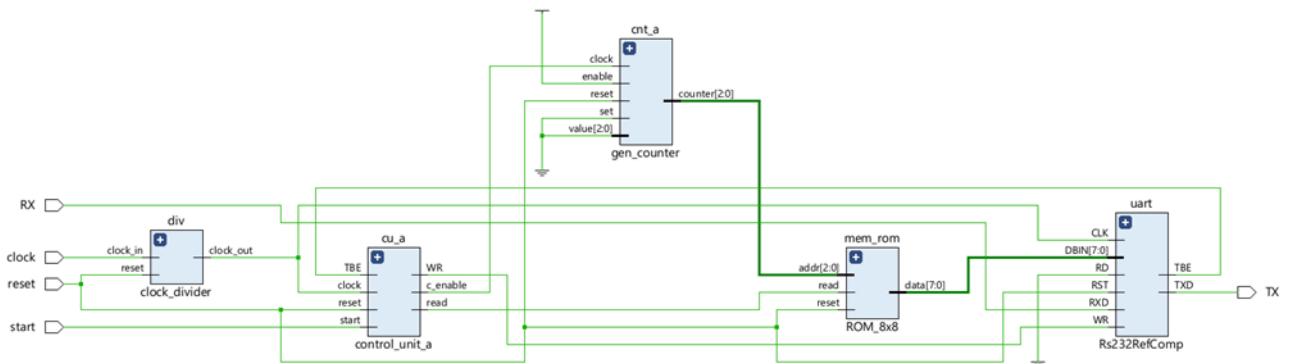
Il sistema A è un'architettura strutturale costituita da diversi componenti:

- **Rom:** una memoria con 8 locazioni da 8 bit ciascuna in cui sono memorizzati i dati da inviare.
- **Contatore** (generic): contatore modulo otto che permette di scandire gli indirizzi della rom; viene incrementato quando il segnale *c_enable* è alto.
- **Clock Divider:** riduce la frequenza del clock di sistema da 100 MHz ad una più bassa di 50 MHz, compatibile con i tempi richiesti dagli altri componenti.
- **Control Unit:** unità di controllo che prende in ingresso il segnale di *start*, *clock* (rallentato), il *reset*, e lo stato del segnale TBE (Transfer Buffer Empty). In base a questi segnali fornisce le abilitazioni per la lettura della rom (READ), per l'incremento del contatore e per l'invio dei dati (WR).
- **Modulo UART (Rs232RefComp):** gestisce la comunicazione seriale. In particolare, trasmette i dati ricevuti dalla rom quando il segnale WR è alto.

Di seguito viene riportato il funzionamento dell'unità di controllo dell'entità A, in particolare essa funziona come una macchina a stati finiti composta da cinque stati.

- **S0 (idle)**: stato iniziale in cui si attende l'avvio del sistema A tramite il segnale start. Non appena start è alto, si passa allo stato successivo per iniziare la sequenza di invio.
- **S1 (wait_request)**: stato di attesa della disponibilità del trasmettitore. In questo stato il sistema abilita la lettura dalla rom (read alto) e attende che TBE si alzi, segnalando che il trasmettitore è pronto. Fino a quando TBE rimane basso si permane in S1.
- **S2 (sending)**: stato in cui avviene l'invio effettivo del dato. Il segnale WR viene alzato e mantenuto tale fin quando TBE non si abbassa, segnalando che il dato è stato preso in carico dal trasmettitore. Solo in questo caso si passa allo stato successivo.
- **S3 (wait_confirm)**: stato di conferma dell'avvenuto invio. Qui il sistema attende che TBE torni di nuovo alto, confermando che il trasmettitore è pronto per ricevere un nuovo dato. Finché TBE è basso, si permane in S3.
- **S4 (cnt_update)**: stato di aggiornamento del contatore. In questo stato viene attivato il segnale c_enable per incrementare il contatore di indirizzamento della ROM. Dopo questo aggiornamento, si ritorna allo stato iniziale S0 per iniziare un nuovo ciclo di invio dati.

System A:



library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity system_a is

```

Port ( start : in STD_LOGIC; -- Start esterno
      reset : in STD_LOGIC; -- Reset esterno
      clock : in STD_LOGIC; -- Clock
      RX : in std_logic; -- Segnale di ricezione, ricevuto da system_b
      TX : out std_logic); -- Segnale di invio, da mandare a system_b
  
```

```
end system_a;
```

```
architecture structural of system_a is
```

```
component control_unit_a is
```

```
Port ( TBE : in STD_LOGIC; -- Richiesta di dati da inviare
```

```
start : in STD_LOGIC; -- Start esterno
```

```
clock : in STD_LOGIC; -- Clock di sistema
```

```
reset : in STD_LOGIC; -- Reset di sistema
```

```
WR : out STD_LOGIC; -- Abilitazione all'invio dati
```

```
read : out STD_LOGIC; -- Abilitazione alla lettura su ROM
```

```
c_enable : out STD_LOGIC); -- Abilitazione del contatore
```

```
end component;
```

```
component clock_divider is
```

```
generic(
```

```
clock_frequency_in : integer := 100000000; -- Frequenza di Clock in ingresso: di default è quello della  
board (100MHz)
```

```
clock_frequency_out : integer := 500); -- Frequenza di Clock in uscita: di default è 500Hz
```

```
Port ( clock_in : in STD_LOGIC; -- Clock in ingresso
```

```
reset : in STD_LOGIC; -- Segnale di reset
```

```
clock_out : out STD_LOGIC); -- Clock "rallentato" in uscita
```

```
end component;
```

```
component gen_counter is
```

```
generic(
```

```
size : natural; -- Numero di bit usati
```

```

module : natural -- Modulo (deve essere minore o uguale di 2^size)
);

port ( clock : in STD_LOGIC; -- Segnale di clock
      reset : in STD_LOGIC; -- Segnale di reset
      enable : in STD_LOGIC; -- Segnale di abilitazione
      set    : in STD_LOGIC; -- Permette di settare un valore stabilito
      value  : in STD_LOGIC_VECTOR(size-1 downto 0); -- Valore stabilito
      counter : out STD_LOGIC_VECTOR(size-1 downto 0)); -- Dato del counter in uscita
end component;

```

```

component ROM_8x8 is
  Port ( reset : in STD_LOGIC; -- Input di reset
         read : in STD_LOGIC; -- Abilitazione alla lettura
         addr : in STD_LOGIC_VECTOR (2 downto 0); -- Input di indirizzamento
         data : out STD_LOGIC_VECTOR (7 downto 0)); -- Dato letto in Output
end component;

```

```

component Rs232RefComp is
  Port (
    TXD    : out std_logic := '1';
    RXD    : in std_logic;
    CLK    : in std_logic;                                --Master Clock
    DBIN   : in std_logic_vector (7 downto 0);--Data Bus in
    DBOUT : out std_logic_vector (7 downto 0);      --Data Bus out
    RDA    : inout std_logic;                            --Read Data Available(1
    quando il dato Ã" disponibile nel registro rdReg)

    TBE    : inout std_logic := '1';                   --Transfer Bus Empty(1 quando il
    dato da inviare Ã" stato caricato nello shift register)

```

```

        RD      : in std_logic;          --Read Strobe(se 1 significa "leggi"
--> fa abbassare RDA)

        WR      : in std_logic;          --Write Strobe(se 1 significa "scrivi"
--> fa abbassare TBE)

        PE      : out std_logic;         --Parity Error Flag

        FE      : out std_logic;         --Frame Error Flag

        OE      : out std_logic;         --Overwrite Error Flag

        RST     : in std_logic := '0');   --Master Reset

end component;

```

```

signal tbe_sig : STD_LOGIC := '0'; -- Segnale TBE per la richiesta di dati da inviare

signal wr_sig : STD_LOGIC := '0'; -- Segnale di abilitazione all'invio dati

signal count_enable : STD_LOGIC := '0'; -- Segnale di abilitazione del contatore

signal read_sig : STD_LOGIC := '0'; -- Segnale che abilita la lettura nella ROM

signal sys_clock : STD_LOGIC := '0'; -- Clock "rallentato" dal divider

signal address : std_logic_vector(2 downto 0); -- Indirizzo della locazione di memoria

signal data_sig : std_logic_vector(7 downto 0); -- Dato da trasferire

```

```
begin
```

```

-- ROM del Sistema A

mem_rom : ROM_8x8 port map (
    reset => reset,
    read => read_sig,
    addr => address,
    data => data_sig);

```

```
-- Control Unit del sistema A
```

```
cu_a : control_unit_a Port map (
    TBE => tbe_sig,
    start => start,
    clock => sys_clock,
    reset => reset,
    WR => wr_sig,
    read => read_sig,
    c_enable => count_enable);
```

-- Contatore degli indirizzi nella ROM

```
cnt_a : gen_counter generic map(
```

```
    size => 3,
```

```
    module => 8)
```

```
    Port map (
```

```
        clock => count_enable,
```

```
        reset => reset,
```

```
        enable => '1',
```

```
        set => '0',
```

```
        value => (others => '0'),
```

```
        counter => address);
```

-- Clock Divider: le specifiche Diligent raccomandano 50MHz (il clock della macchina è 100MHz)

```
div : clock_divider generic map(
```

```
    clock_frequency_in => 100000000,
```

```
    clock_frequency_out => 50000000)
```

```
    Port map (
```

```
        reset => reset,
```

```

clock_in => clock,
clock_out => sys_clock);

-- Dispositivo UART-RS232

uart : Rs232RefComp

port map (
    TXD      => TX,
    RXD      => RX,
    CLK      => sys_clock,
    DBIN     => data_sig,
    TBE      => tbe_sig,
    RD       => '-'; -- system_a non deve leggere, ma solo inviare dati
    WR      => wr_sig,
    RST     => reset);

```

end structural;

CONTROL UNIT A:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```

entity control_unit_a is

```

Port ( TBE : in STD_LOGIC; -- Richiesta di dati da inviare
      start : in STD_LOGIC; -- Start esterno
      clock : in STD_LOGIC; -- Clock di sistema
      reset : in STD_LOGIC; -- Reset di sistema
      WR : out STD_LOGIC; -- Abilitazione all'invio dati

```

```
read : out STD_LOGIC; -- Abilitazione alla lettura su ROM  
c_enable : out STD_LOGIC); -- Abilitazione del contatore  
end control_unit_a;
```

```
architecture behavioral of control_unit_a is
```

```
type state is (idle,wait_request,sending,wait_confirm,cnt_update);
```

```
signal current_state : state := idle;
```

```
signal next_state : state;
```

```
begin
```

```
state_trasition : process (clock, tbe, start, current_state)
```

```
begin
```

```
case current_state is
```

```
-- Fase di Idle: si parte solo dopo lo start
```

```
when idle =>
```

```
if(start = '1') then
```

```
next_state <= wait_request;
```

```
else
```

```
next_state <= idle;
```

```
end if;
```

```
-- Fase di attesa di una richiesta (fino a quando non si alza TBE)
```

```
when wait_request =>
```

```
if(TBE = '0') then
```

```

next_state <= wait_request;

else

next_state <= sending;

end if;

-- Fase di invio del dato (deve essere alto fino a quando non abbiamo finito)

when sending =>

if(TBE ='1') then

next_state <= sending;

else

next_state <= wait_confirm;

end if;

-- Fase di attesa per un secondo TBE, per confermare il corretto invio

when wait_confirm =>

if(TBE = '0') then

next_state <= wait_confirm;

else

next_state <= cnt_update;

end if;

-- Fase di aggiornamento del contatore che precede l'idle

when cnt_update =>

next_state <= idle;

end case;

end process;

```

```
stato_uscita : process (clock)
begin
    case current_state is
        -- Fase di Idle: si parte solo dopo lo start
        when idle =>
            WR <= '0';
            read <= '0';
            c_enable <= '0';

        -- Fase di attesa di una richiesta (fino a quando non si alza TBE)
        when wait_request =>
            WR <= '0';
            read <= '1';
            c_enable <= '0';

        -- Fase di invio del dato (deve essere alto fino a quando non abbiamo finito)
        when sending =>
            WR <= '1';
            read <= '1';
            c_enable <= '0';

        -- Fase di attesa per un secondo TBE, per confermare il corretto invio
        when wait_confirm =>
            WR <= '0';
            read <= '0';
```

```
c_enable <= '0';

-- Fase di aggiornamento del contatore che precede l'idle
when cnt_update =>

WR <= '0';
read <= '0';
c_enable <= '1';

end case;

end process;

-- Tempificatore degli stati
CLKP : process (clock,reset)
begin
if (rising_edge(clock)) then
if (reset = '0') then
current_state <= next_state;
else
current_state <= idle;
end if;
end if;
end process;

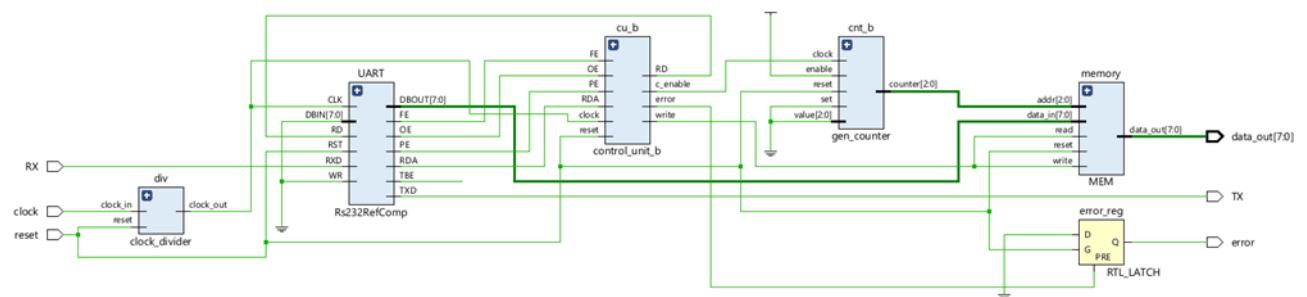
end behavioral;
```

Sistema B

Il sistema B rappresenta il ricevitore, che contiene, Uart, l'unità di controllo, due contatori, uno che funge da clock divider e uno che funge da contatore per scandire le locazioni di memoria. Dato che i contatori e l'unità di memoria sono già presente in appendice non verranno presentate. L'unità di controllo di B presenta quattro stati:

- S0(IDLE) : questo è lo stato iniziale in cui il ricevitore attende il segnale RDA(read data available) un segnale di ricevimento che indica la presenza di un dato, se è alto si avvia verso next state altrimenti resta in idle.
- S1(Verify) : questo è lo stato di verifica degli errori, si controlla se uno dei tre flag degli errori è alto o basso. Se uno dei 3 è alto si va nello stato error report altrimenti si transita nello stato wait ending. In questo stato si abilita l'incremento del contatore, la scrittura del dato in memoria e si alza RD per notificare all'UART che il dato è stato letto.
- S2(Error report) : questo è lo stato di gestione dell'errore. Se RDA rimane alto, significa che il dato errato è ancora presente nel registro di ricezione della UART, si attende che venga scartato, passando a Wait_ending. Se invece RDA è basso, si ritorna direttamente in idle.
- S3(Wait ending) : in questo stato si attende che RDA si abbassi, sta ad indicare che il dato è stato gestito e che la UART è pronta a ricevere un nuovo byte, quando il segnale si abbassa si ritorna in idle.

System B:



library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity system_b is

Port (clock : in STD_LOGIC; -- Clock di sistema

reset : in STD_LOGIC; -- Reset esterno

RX : in STD_LOGIC; -- Segnale di ricezione, da mandare a system_a

TX : out STD_LOGIC; -- Segnale di invio, da ricevere da system_a

```
error : out STD_LOGIC; -- Segnale che avvisa della presenza di errori  
data_out : out STD_LOGIC_VECTOR (7 downto 0)); -- Dato salvato in memoria  
end system_b;
```

architecture structural of system_b is

component control_unit_b is

Port (RDA: in STD_LOGIC; -- Segnale di lettura disponibile

OE: in STD_LOGIC; -- Errore di overrun

PE: in STD_LOGIC; -- Errore di parità

FE : in STD_LOGIC; -- Errore di framing

clock : in STD_LOGIC; -- Clock di sistema

reset : in STD_LOGIC; -- Reset di sistema

c_enable : out STD_LOGIC; -- Abilitatore del counter

RD : out STD_LOGIC; -- Segnale di conferma lettura del dato

write : out STD_LOGIC; -- Abilitazione al salvataggio del dato

error : out STD_LOGIC); -- Segnalazione degli errori

end component;

component clock_divider is

generic(

clock_frequency_in : integer := 100000000; -- Frequenza di Clock in ingresso: di default è
quello della board (100MHz)

clock_frequency_out : integer := 500); -- Frequenza di Clock in uscita: di default è 500Hz

Port (clock_in : in STD_LOGIC; -- Clock in ingresso

reset : in STD_LOGIC; -- Segnale di reset

clock_out : out STD_LOGIC); -- Clock "rallentato" in uscita

end component;

```

component gen_counter is
generic(
    size : natural; -- Numero di bit usati
    module : natural -- Modulo (deve essere minore o uguale di 2^size)
);
port (
    clock : in STD_LOGIC; -- Segnale di clock
    reset : in STD_LOGIC; -- Segnale di reset
    enable : in STD_LOGIC; -- Segnale di abilitazione
    set : in STD_LOGIC; -- Permette di settare un valore stabilito
    value : in STD_LOGIC_VECTOR(size-1 downto 0); -- Valore stabilito
    counter : out STD_LOGIC_VECTOR(size-1 downto 0)); -- Dato del counter in uscita
end component;

```

```

component MEM is
Generic (
    Mem_size : positive; -- Numero di locazioni di memoria
    Addr_size : positive; -- Numero di bit per l'indirizzo (Mem_size può essere al più
    2^Addr_size)
    Data_size : positive); -- Dimensione dei dati in memoria
Port (
    reset : in STD_LOGIC; -- Input di reset
    data_in : in STD_LOGIC_VECTOR (Data_size-1 downto 0); -- Dato fornito in ingresso
    addr : in STD_LOGIC_VECTOR (Addr_size-1 downto 0); -- Input di indirizzamento
    write : in STD_LOGIC; -- Abilitazione alla scrittura
    read : in STD_LOGIC; -- Abilitazione alla lettura
    data_out : out STD_LOGIC_VECTOR (Data_size-1 downto 0)); -- Dato letto in Output
end component;

```

```

component Rs232RefComp is

Port (
    TXD : out std_logic := '1';

    RXD : in std_logic;

    CLK  : in std_logic;                                --Master Clock

    DBIN           : in std_logic_vector (7 downto 0);--Data Bus in

    DBOUT : out std_logic_vector (7 downto 0); --Data Bus out

    RDA : inout std_logic;                            --Read Data
Available(1 quando il dato Ã“ disponibile nel registro rdReg)

    TBE : inout std_logic := '1';                     --Transfer Bus Empty(1
quando il dato da inviare Ã“ stato caricato nello shift register)

    RD      : in std_logic;                         --Read Strobe(se 1 significa
"leggi" --> fa abbassare RDA)

    WR      : in std_logic;                         --Write Strobe(se 1 significa
significa "scrivi" --> fa abbassare TBE)

    PE      : out std_logic;                        --Parity Error Flag

    FE      : out std_logic;                        --Frame Error Flag

    OE      : out std_logic;                        --Overwrite Error Flag

    RST     : in std_logic := '0';                  --Master Reset

end component;

signal RDA : STD_LOGIC; -- Segnale di ricevimento dati

signal OE, PE, FE : STD_LOGIC; -- Segnali di errore ricevimento dati (Overrun, Parity, Framing)

signal count_enable : STD_LOGIC; -- Segnale che abilita il contatore

signal RD : STD_LOGIC; -- Segnale di abilitazione alla lettura del dato ricevuto

signal write_sig : STD_LOGIC; -- Segnale di abilitazione alla scrittura su memoria

signal error_sig : STD_LOGIC; -- Segnale di conferma presenza errori

signal sys_clock : STD_LOGIC; -- Clock del sistema "rallentato" dal divider

signal address : STD_LOGIC_VECTOR(2 downto 0); -- Indirizzo di memoria su cui scrivere

```

```

signal data_sig : STD_LOGIC_VECTOR(7 downto 0); -- Dato da scrivere in memoria

begin

    error <= '1' when error_sig = '1' else '0' when reset = '1'; -- Impostiamo a 0 il segnale di errore nel
caso in cui il reset è alto

    -- Control Unit del sistema B

    cu_b : control_unit_b port map(
        RDA => RDA,
        OE => OE,
        PE => PE,
        FE => FE,
        clock => sys_clock,
        reset => reset,
        c_enable => count_enable,
        RD => RD,
        write => write_sig,
        error => error_sig);

    -- Contatore degli indirizzi nella ROM

    cnt_b : gen_counter generic map(
        size => 3,
        module => 8)

    Port map (
        clock => count_enable,
        reset => reset,

```

```
enable => '1',
set => '0',
value => (others => '0'),
counter => address);
```

-- Clock Divider: le specifiche Diligent raccomandano 50MHz (il clock della macchina è 100MHz)

```
div : clock_divider generic map(
clock_frequency_in => 100000000,
clock_frequency_out => 50000000)
```

Port map (

```
reset => reset,
clock_in => clock,
clock_out => sys_clock);
```

-- Memoria scrivibile del sistema

```
memory : MEM generic map(
```

```
Mem_size => 8,
Addr_size => 3,
Data_size => 8)
```

port map(

```
write => write_sig,
read => write_sig, -- Leggiamo assieme alla scrittura
reset => reset,
data_in => data_sig,
addr => address,
data_out => data_out);
```

```

-- Dispositivo UART-RS232

UART : Rs232RefComp port map(
    TXD => TX,
    RXD => RX,
    CLK => sys_clock,
    DBIN => (others=> '-'),
    DBOUT => data_sig,
    RDA => RDA,
    RD => RD,
    WR => '0',
    PE => PE,
    OE => OE,
    FE => FE,
    RST => reset);
end structural;

```

Control unit:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity control_unit_b is
    Port (
        RDA: in STD_LOGIC; -- Segnale di lettura disponibile
        OE: in STD_LOGIC; -- Errore di overrun
        PE: in STD_LOGIC; -- Errore di parità
        FE : in STD_LOGIC; -- Errore di framing
        clock : in STD_LOGIC; -- Clock di sistema
        reset : in STD_LOGIC; -- Reset di sistema
    );
end entity;

```

```
c_enable : out STD_LOGIC; -- Abilitatore del counter  
RD : out STD_LOGIC; -- Segnale di conferma lettura del dato  
write : out STD_LOGIC; -- Abilitazione al salvataggio del dato  
error : out STD_LOGIC); -- Segnalazione degli errori  
end control_unit_b;
```

```
architecture Behavioral of control_unit_b is
```

```
type state is (idle,verify,error_report,wait_ending);  
signal current_state: state := idle;  
signal next_state : state;
```

```
begin
```

```
transizione_stato : process (clock, RDA, OE, PE, FE, current_state)
```

```
begin
```

```
case current_state is
```

```
-- Fase di idle: la macchina si avvia quando riceve un segnale di ricevimento dati (ossia se  
RDA è alto)
```

```
when idle =>  
if (RDA = '1') then next_state <= verify;  
else next_state <= idle;  
end if;
```

```
-- Fase di verifica degli errori
```

```
when verify =>

if (OE = '1' or PE = '1' or FE = '1') then next_state <= error_report;

else next_state <= wait_ending;

end if;
```

-- Fase di eventuale output dell'errore

```
when error_report =>

if (RDA = '1') then

next_state <= wait_ending;

else

next_state <= idle;

end if;
```

-- Fase di attesa che RDA si abbassi e si ritorni all'idle

```
when wait_ending =>

if (RDA = '1') then

next_state <= wait_ending;

else

next_state <= idle;

end if;
```

end case;

end process;

stato_uscita : process (clock)

begin

```
case next_state is
    when idle =>
        c_enable <= '0';
        write <= '0';
        RD <= '0';
        error <= '0';

    when verify =>
        c_enable <= '1';
        write <= '1';
        RD <= '1';
        error <= '0';

    when error_report =>
        c_enable <= '0';
        write <= '0';
        RD <= '0';
        error <= '1';

    when wait_ending =>
        c_enable <= '0';
        write <= '0';
        RD <= '0';
        error <= '0';

    end case;
end process;
```

-- Tempificazione degli stati

```

CLKP : process (clock,reset)
begin
  if (rising_edge(clock)) then
    if (reset = '0') then
      current_state <= next_state;
    else
      current_state <= idle;
    end if;
  end if;
end process;

```

end Behavioral;

Simulazione

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```

```

entity system_ab_tb is
end system_ab_tb;

```

```

architecture system_ab of system_ab_tb is

```

component system is

Port (

start : in STD_LOGIC;

reset : in STD_LOGIC;

clock : in STD_LOGIC);

end component;

```
signal STRT, RST, CLK, SIG_A, SIG_B, SIG_ERR : std_logic := '0';
```

```
begin
```

```
    sys : system
```

```
    port map(
```

```
        start => STRT,
```

```
        reset => RST,
```

```
        clock => CLK);
```

```
CLKP : process
```

```
begin
```

```
    CLK <= '1';
```

```
    wait for 5 ns;
```

```
    CLK <= '0';
```

```
    wait for 5 ns;
```

```
end process;
```

```
test : process
```

```
begin
```

```
    RST <= '1';
```

```
    wait for 30 ns;
```

```
    RST <= '0';
```

```
    wait for 15 ns;
```

```
    STRT <= '1';
```

```
    wait for 200ns;
```

```

    wait;

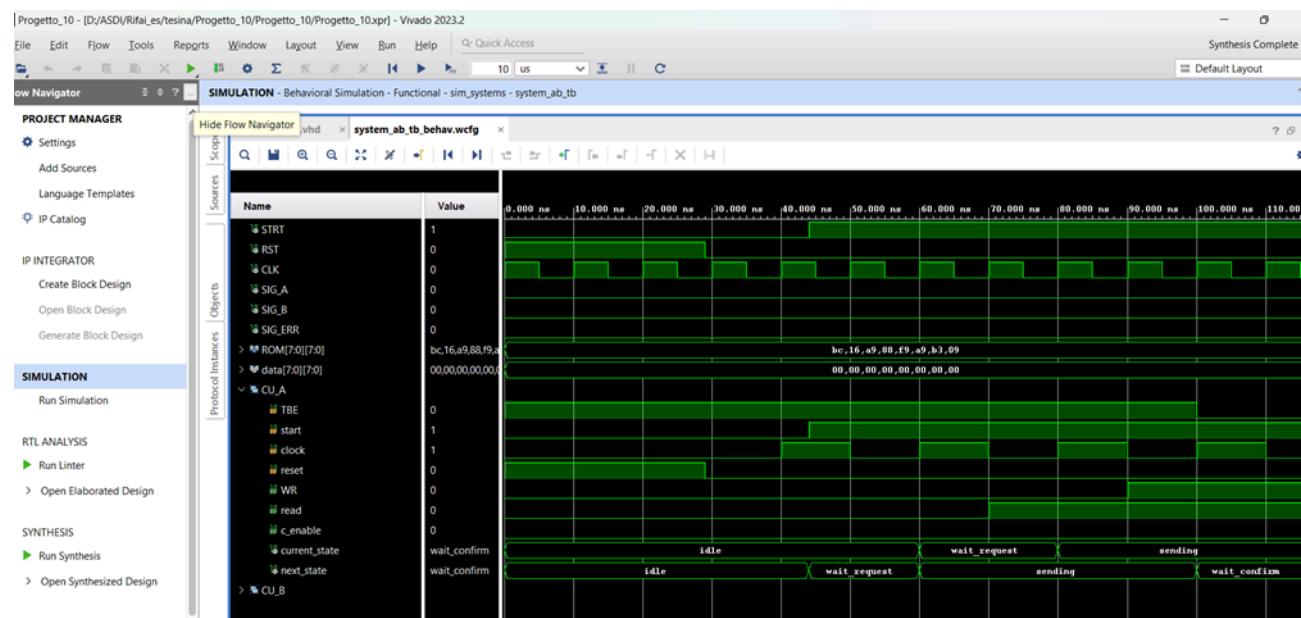
end process;

```

```

end system_ab;

```



Capitolo 7: Switch Multistadio

Esercizio 11.1

Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network. Lo switch deve consentire lo scambio di messaggi di 2 bit ciascuno da un nodo sorgente a un nodo destinazione in una rete con 4 nodi, implementando uno schema a priorità fissa fra i nodi (es. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).

Progetto e architettura

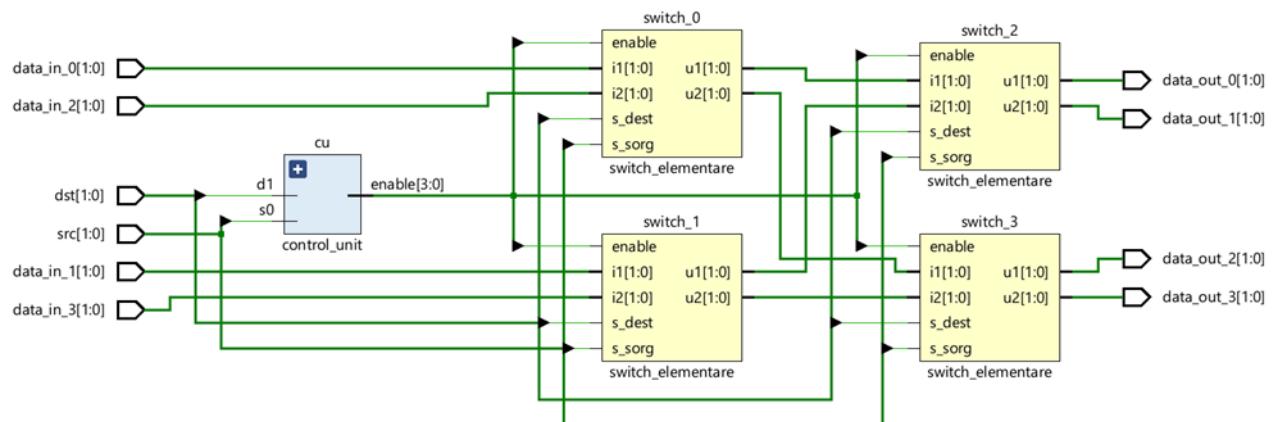
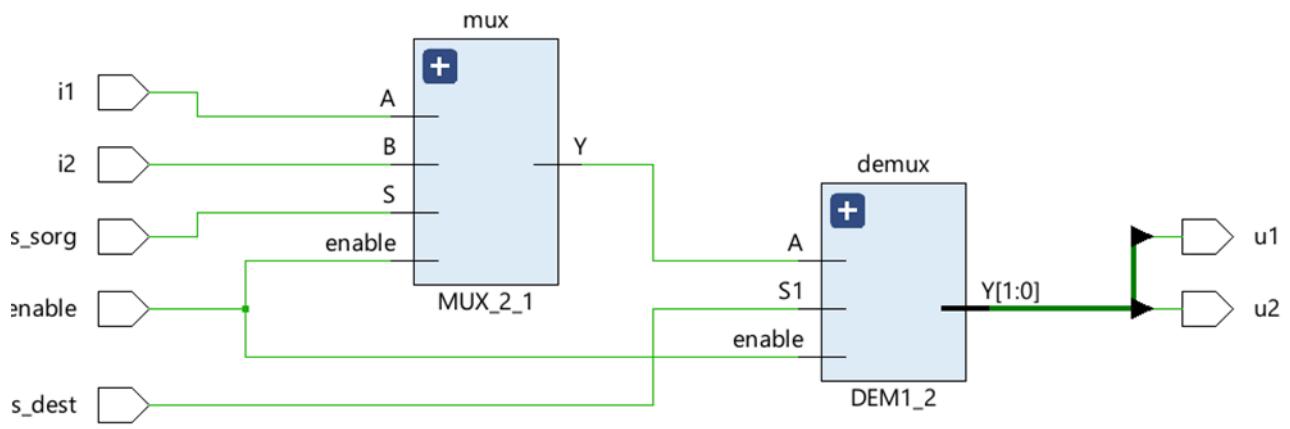
Per creare uno switch che connette due nodi è necessario disporre di due informazioni:

- L'indirizzo di nodo sorgente
- L'indirizzo di nodo destinazione

A tal fine scomponiamo il sistema in due sottosistemi, uno di ingresso e uno di uscita. Tale suddivisione può essere realizzata utilizzando un multiplexer per l'ingresso e un demultiplexer per l'uscita. I componenti elementari vengono collegati in accordo con la tecnica del **perfect shuffling**. Secondo questa tecnica, avendo M carte, dopo un numero di passaggi di rimescolamento perfetto pari a $\log_2(M)$ si riottiene l'ordinamento di partenza. In questo esercizio si hanno 4 nodi e quindi sono necessari 2 stadi ($\log_2(4)=2$) per garantire la connettività totale del sistema. L'indirizzo deve essere diviso in tante parti quanti sono gli stati in modo da comandare tutti i blocchi che fanno parte del sistema. Siccome i bit da inviare sono due per messaggio, avremo un pacchetto che sarà composto da due bit di dato e due bit di destinazione per un totale di 4 bit. In uscita allo switch saranno presenti solo i due bit di dato. Inoltre, per quanto concerne la gestione delle possibili collisioni, tenuto conto che lo switch deve essere in grado di gestire dei livelli di priorità, si è scelto di assegnare priorità fisse ai vari nodi (nodo 1 più prioritario, con priorità decrescente fino a nodo 4).

Per la realizzazione di questo esercizio si è utilizzato un approccio strutturale. In particolare, il sistema contiene vari blocchi denominati switch che si occuperanno dell'instradamento. Tali blocchi, come anticipato, sono costituiti da un multiplexer 2:1 e da un demux 1:2.

Per il multiplexer la selezione è costituita dall'indirizzo sorgente, mentre per il demux la selezione è costituita dall'indirizzo destinazione. Utilizzando nell'insieme questi componenti si va a comporre la rete di Omega Network.



Implementazione

Switch Elementare (il mux e il demux sono implementati in appendice):

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity switchElementare is
  Port (
    -- Ingressi: i1 e i2 sono i dati di input
    i1 : in std_logic_vector(1 downto 0);
    i2 : in std_logic_vector(1 downto 0);
    -- Selezione della sorgente: s_sorg controlla quale ingresso viene scelto
    s_sorg : in std_logic;
    enable : in std_logic;
    s_dest : out std_logic;
    u1 : out std_logic;
    u2 : out std_logic
  );
end entity;
  
```

```

s_sorg : in std_logic;
-- Selezione della destinazione: s_dest controlla quale uscita viene attivata
s_dest : in std_logic;
-- Segnale di abilitazione (ENABLE): attiva o disattiva il funzionamento del circuito
enable : in std_logic;
-- Uscite: u1 e u2 sono i dati di output
u1 : out std_logic_vector(1 downto 0);
u2 : out std_logic_vector(1 downto 0)
);

end switchElementare;

architecture Structural of switchElementare is

-- Componente MUX 2:1
component MUX_2_1 is

Port(
A : in STD_LOGIC_VECTOR(1 downto 0); -- Ingresso A
B : in STD_LOGIC_VECTOR(1 downto 0); -- Ingresso B
enable : in STD_LOGIC; -- Segnale di abilitazione
S : in STD_LOGIC; -- Selezione
Y : out STD_LOGIC_VECTOR(1 downto 0)-- Uscita
);

end component;

-- Componente DEMUX 1:2
component DEM1_2 is

Port(
A : in STD_LOGIC_VECTOR(1 downto 0); -- Ingresso
S1 : in STD_LOGIC; -- Selezione
enable : in STD_LOGIC; -- Segnale di abilitazione

```

```
Y0: out STD_LOGIC_VECTOR(1 downto 0); -- Uscita 0  
Y1: out STD_LOGIC_VECTOR(1 downto 0)          -- Uscita 1  
);  
end component;
```

-- Segnale intermedio per connettere MUX e DEMUX

```
Signal mid_sig : STD_LOGIC_VECTOR(1 downto 0) := (others => '0');
```

```
begin
```

-- Istanziazione del MUX 2:1

```
mux : MUX_2_1
```

```
Port map (
```

```
    A => i1,
```

```
    B => i2,
```

```
    enable => enable,
```

```
    S => s_sorg,
```

```
    Y => mid_sig
```

```
);
```

-- Istanziazione del DEMUX 1:2

```
demux : DEM1_2
```

```
Port map(
```

```
    A => mid_sig,           -- Segnale intermedio proveniente dal MUX
```

```
    S1 => s_dest,          -- Selezione della destinazione
```

```
    enable => enable,       -- Abilitazione
```

```
    Y0 => u1,               -- Uscita 1
```

```
    Y1 => u2               -- Uscita 2
```

```

    );
}

end Structural;

Control Unit:

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity control_unit is
  Port (
    s0 : in std_logic; -- bit meno significativo nel segnale di sorgente
    d1 : in std_logic; -- bit più significativo nel segnale di destinazione
    enable : out std_logic_vector( 3 downto 0)
  );
end control_unit;

architecture Behavioral of control_unit is
begin

  cu_process : process (s0, d1)
  begin

    case s0 is
      when '0' =>
        if(d1='0') then -- 00
          enable <= "0101";
        elsif(d1='1') then -- 01

```

```

enable <= "1001";
else
enable <= (others=>'0');
end if;

when '1' =>
if(d1='0') then --10
enable<= "0110" ;
elsif (d1 = '1') then --11
enable <= "1010" ;
else
enable <= (others =>'0');
end if;

when others =>
enable <= (others =>'0');

end case;
end process;

end Behavioral;

```

Omega Network :

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity omega_network is
Port (
-- input

```

```

data_in_0 : in STD_LOGIC_VECTOR(1 downto 0); -- dato in ingresso nella prima posizione sorgente
data_in_1 : in STD_LOGIC_VECTOR(1 downto 0); -- seconda
data_in_2 : in STD_LOGIC_VECTOR(1 downto 0); -- terza
data_in_3 : in STD_LOGIC_VECTOR(1 downto 0); -- quarta
-- selezione
src : in STD_LOGIC_VECTOR(1 downto 0); -- sorgente in ingresso allo switch
dst : in STD_LOGIC_VECTOR(1 downto 0);-- destinazione di uscita
-- output
data_out_0 : out STD_LOGIC_VECTOR(1 downto 0);
data_out_1 : out STD_LOGIC_VECTOR(1 downto 0);
data_out_2 : out STD_LOGIC_VECTOR(1 downto 0);
data_out_3 : out STD_LOGIC_VECTOR(1 downto 0)

);

end omega_network;

```

architecture Structural of omega_network is

```

component control_unit
port (
s0 : in std_logic; -- bit meno significativo nel segnale di sorgente
d1 : in std_logic; -- bit più significativo nel segnale di destinazione
enable : out std_logic_vector( 3 downto 0)
);


```

```

end component;

component switchElementare
port(
i1 : in std_logic_vector(1 downto 0);
i2 : in std_logic_vector(1 downto 0);
s_sorg : in std_logic;
s_dest : in std_logic;
enable : in std_logic;
u1 : out std_logic_vector(1 downto 0);
u2 : out std_logic_vector(1 downto 0)
);

```

```
end component;
```

```
signal switch_0_to_2 : STD_LOGIC_VECTOR(1 downto 0) := ( others=>'0'); -- segnale di intermezzo dallo
switch 0 al 2
```

```
signal switch_0_to_3 : STD_LOGIC_VECTOR(1 downto 0) := ( others=>'0'); -- dallo 0 al tre
```

```
signal switch_1_to_2 : STD_LOGIC_VECTOR(1 downto 0) := ( others=>'0');-- dal 1 al due
```

```
signal switch_1_to_3 : STD_LOGIC_VECTOR(1 downto 0) := ( others=>'0');-- dal 1 al 3
```

```
signal enable_sig : STD_LOGIC_VECTOR(3 downto 0) := ( others=>'0'); -- mappatura enable
```

```
begin
```

```
cu : control_unit
```

```
port map(
```

```
s0 => src(0),
d1 => dst(1),
enable =>enable_sig
);

switch_0 : switchElementare
port map(
enable => enable_sig(0),
i1 => data_in_0,
i2 => data_in_2,
s_sorg => src(1),
s_dest => dst(1),
u1 => switch_0_to_2,
u2 => switch_0_to_3
);

switch_1 : switchElementare
port map(
enable => enable_sig(1),
i1 => data_in_1,
i2 => data_in_3,
s_sorg => src(1),
s_dest => dst(1),
u1 => switch_1_to_2,
u2 => switch_1_to_3
);
```

```
switch_2 : switchElementare
port map(
    enable => enable_sig(2),
    i1 => switch_0_to_2,
    i2 => switch_1_to_2,
    s_sorg => src(0),
    s_dest => dst(0),
    u1 => data_out_0,
    u2 => data_out_1
);
switch_3 : switchElementare
port map(
    enable => enable_sig(3),
    i1 => switch_0_to_3,
    i2 => switch_1_to_3,
    s_sorg => src(0),
    s_dest => dst(0),
    u1 => data_out_2,
    u2 => data_out_3
);
end Structural;
```

Simulazione

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY tb_omega_network IS
END tb_omega_network;

ARCHITECTURE behavioral OF tb_omega_network IS

COMPONENT omega_network
PORT(
    data_in_0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    data_in_1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    data_in_2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    data_in_3 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    src      : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    dst      : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    data_out_0 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
    data_out_1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
    data_out_2 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
    data_out_3 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
);
END COMPONENT;

-- Signals
SIGNAL in_0, in_1, in_2, in_3 : STD_LOGIC_VECTOR(1 DOWNTO 0) := (others => '0');
```

```

SIGNAL src : STD_LOGIC_VECTOR(1 DOWNTO 0) := (others => '0'); -- inizializzazione
SIGNAL dst : STD_LOGIC_VECTOR(1 DOWNTO 0) := (others => '0'); -- inizializzazione
SIGNAL out_0, out_1, out_2, out_3 : STD_LOGIC_VECTOR(1 DOWNTO 0) ;

BEGIN
    uut: omega_network PORT MAP (
        data_in_0 => in_0,
        data_in_1 => in_1,
        data_in_2 => in_2,
        data_in_3 => in_3,
        src => src,
        dst => dst,
        data_out_0 => out_0,
        data_out_1 => out_1,
        data_out_2 => out_2,
        data_out_3 => out_3
    );

```



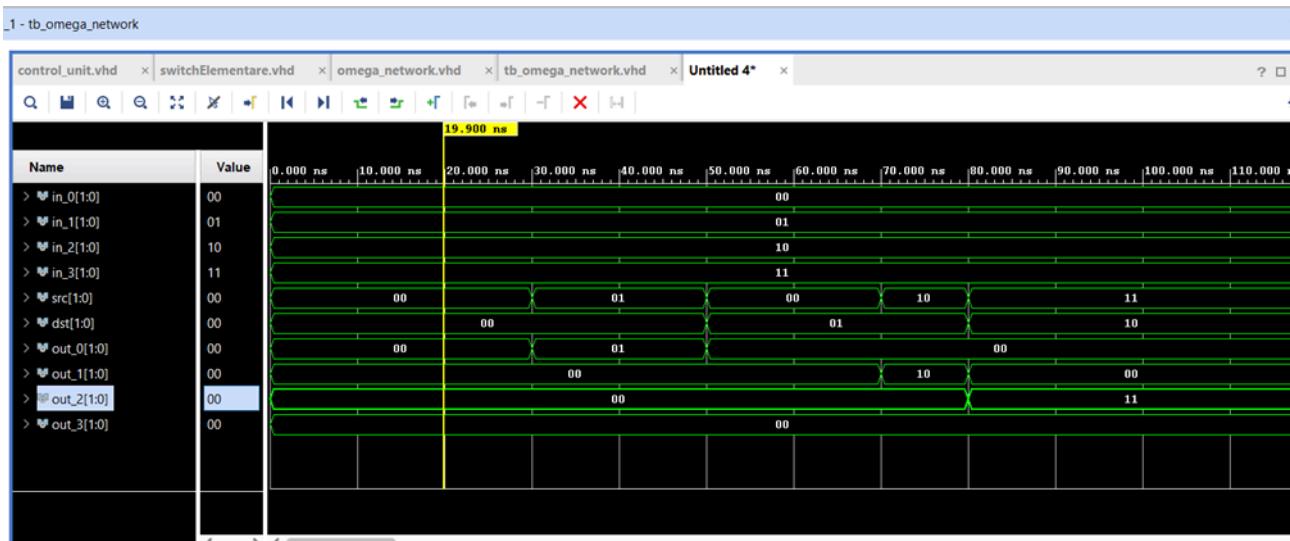
```

stim_proc: PROCESS
BEGIN
    in_0 <= "00"; -- Input 0
    in_1 <= "01"; -- Input 1
    in_2 <= "10"; -- Input 2
    in_3 <= "11"; -- Input 3

    wait for 10 ns;

```

```
src <= "00"; -- Source = 00  
dst <= "00"; -- Destination = 00  
-- DATA OUT_0 DEVE ESSERE ALTO  
wait for 20 ns;  
  
  
src <= "01"; -- Source = 01  
dst <= "00"; -- Destination = 00  
wait for 20 ns;  
  
  
src <= "00"; -- Source = 00  
dst <= "01"; -- Destination = 01  
wait for 20 ns;  
  
  
src <= "10"; -- Source = 10  
dst <= "01"; -- Destination = 01  
wait for 10 ns;  
  
  
src <= "11"; -- Source = 11  
dst <= "10"; -- Destination = 10  
wait for 20 ns;  
  
  
wait;  
END PROCESS;  
  
END behavioral;
```



Come si evince dal testbench riportato, facendo riferimento al primo test case 00->00, in out_0 è visualizzato correttamente il dato in input.

Capitolo 8: Esercizio d'esame

Esercizio 12 (prova di esame del 19 dicembre 2024)

Un sistema è composto da 2 nodi, A e B. A include una ROM (progettata come macchina sequenziale con READ sincrono) di 8 locazioni da 4 bit, mentre B include un sommatore parallelo in grado di effettuare la somma di 2 stringhe di 4 bit ciascuna e un registro R di 4 bit. Il sistema opera come segue: all'arrivo di un segnale di start, A inizia a prelevare gli elementi ROM[i] dalla propria memoria e li invia, uno alla volta, a B mediante handshaking. B somma progressivamente le stringhe ricevute utilizzando il sommatore e alla fine inserisce il risultato nel registro R.

1. Si disegni l'architettura complessiva del sistema tramite un diagramma a blocchi, identificando parte operativa e parte di controllo di ciascun nodo. Ogni nodo deve essere progettato seguendo un approccio strutturale, individuando tutti i componenti, le loro interfacce e le loro interconnessioni.
2. Si progettino le unità di controllo di A e B evidenziando gli stati, gli ingressi e le uscite negli automi risultanti. E' obbligatorio specificare la tempificazione che si intende dare alle macchine (fronte attivo del clock, tempificazione dei segnali di READ/WRITE su registri e memorie).
3. Si progetti il sommatore secondo un'architettura di tipo carry look ahead.
4. Si fornisca l'implementazione in VHDL dell'intero sistema e si proceda alla simulazione nel caso in cui il clock del sistema A e del sistema B siano diversi (A più lento e B più veloce).

MACCHINA CARRY LOOK AHEAD

Prima di descrivere il sistema abbiamo realizzato la macchina sommatore carry look ahead.

Questa macchina è in grado di effettuare la somma “anticipando il riporto”. Infatti possiamo osservare l’equazione del riporto:

- $C_{i+1} = x_i * y_i + c_i * (x_i + y_i)$

Possiamo notare due dipendenze:

- $G = x * y$ (condizione di generazione)
- $P = x + y$ (condizione di riporto)

Da qui possiamo riscrivere la prima formula nel seguente modo:

- $C_{i+1} = G_i + P_i * c_i$

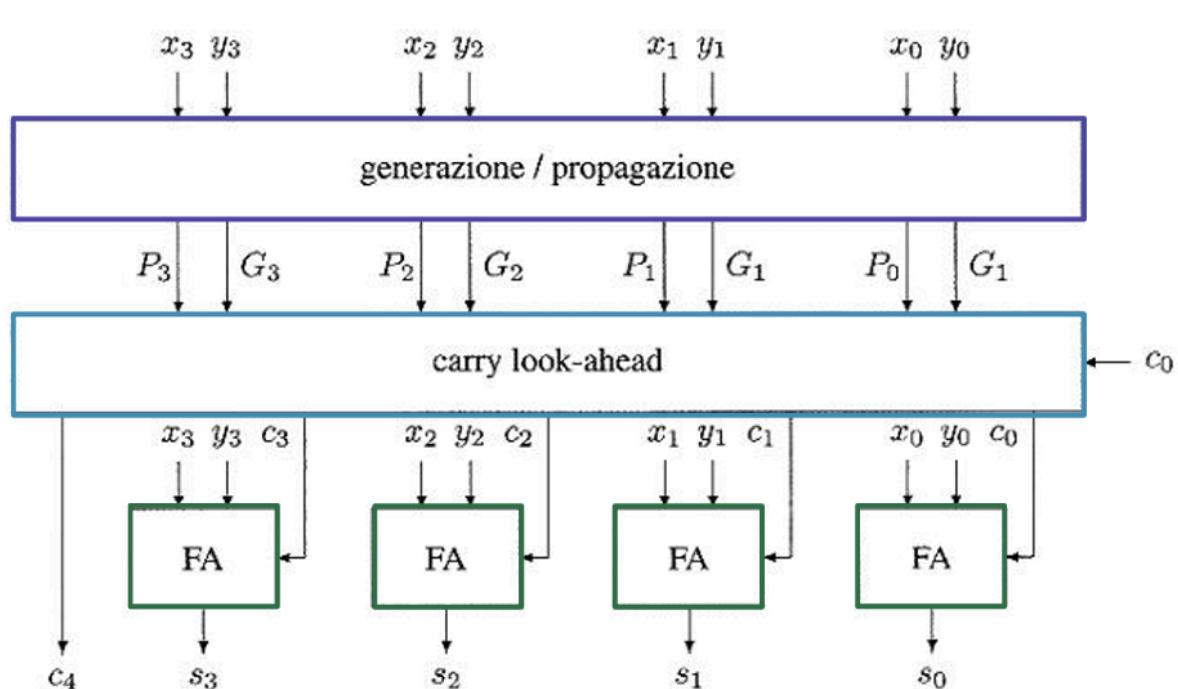
Notiamo che il riporto “ C_{i+1} ” dipende dal riporto precedente sia dalla generazione dello stato “ i ”. Da qui, possiamo generalizzare il calcolo dei riporti per poi poter effettuare la somma.

Otteniamo ciò:

- $C_1 = G_0 + P_0 c_0$
- $C_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$
- $C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$
- $C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$

Abbiamo ottenuto 4 riporti, tutti dipendenti da c_0 , e potremmo continuare così all’infinito, ma ciò comporterebbe una crescita esponenziale dell’architettura man mano che aumentano gli operandi.

Dunque l’architettura del carry look-ahead può essere realizzata tramite 3 livelli, nel primo si calcolano P e G mentre nel secondo livello si calcola il riporto e nel terzo livello si effettua la somma.



implementazione

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

entity carry_lookahead_adder is
    Generic (N : integer := 4); -- Numero di bit
    Port (
        A      : in std_logic_vector(N-1 downto 0);
        B      : in std_logic_vector(N-1 downto 0);
        Cin   : in std_logic;
        Sum   : out std_logic_vector(N-1 downto 0);
        Cout  : out std_logic
    );
end carry_lookahead_adder;

architecture Behavioral of carry_lookahead_adder is
    signal P, G : std_logic_vector(N-1 downto 0); -- Propagate and Generate signals
    signal C   : std_logic_vector(N downto 0); -- Carry signals
begin
    P <= A xor B; -- Propagate
    G <= A and B; -- Generate

    -- Logica Carry
    C(0) <= Cin;
    gen_carry: for i in 0 to N-1 generate
        C(i+1) <= G(i) or (P(i) and C(i));
    end generate;

    -- Calcolo della Somma

```

Sum <= P xor C(N-1 downto 0);

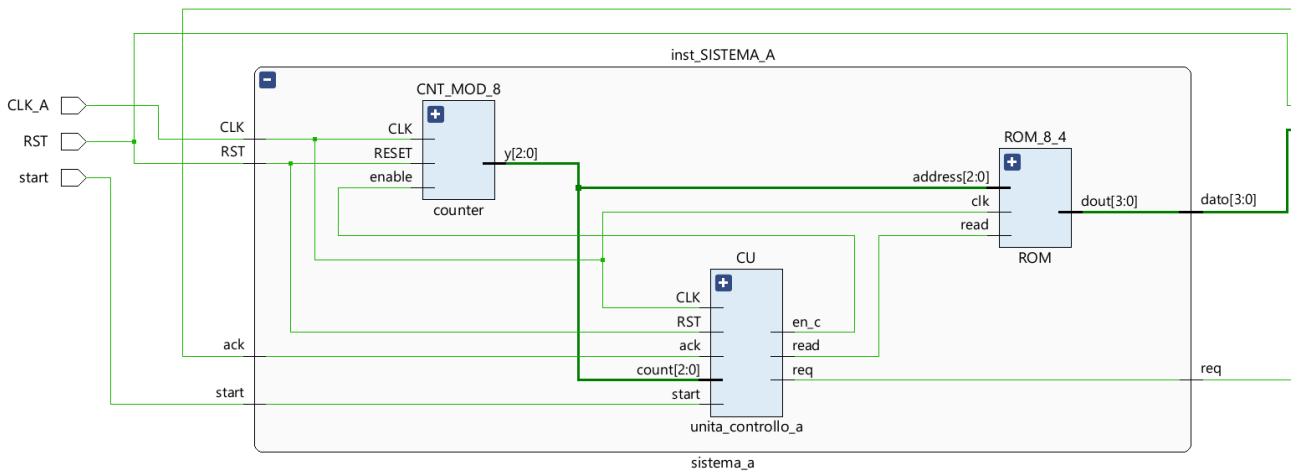
-- Carry finale

Cout <= C(N);

end Behavioral;

SISTEMA A

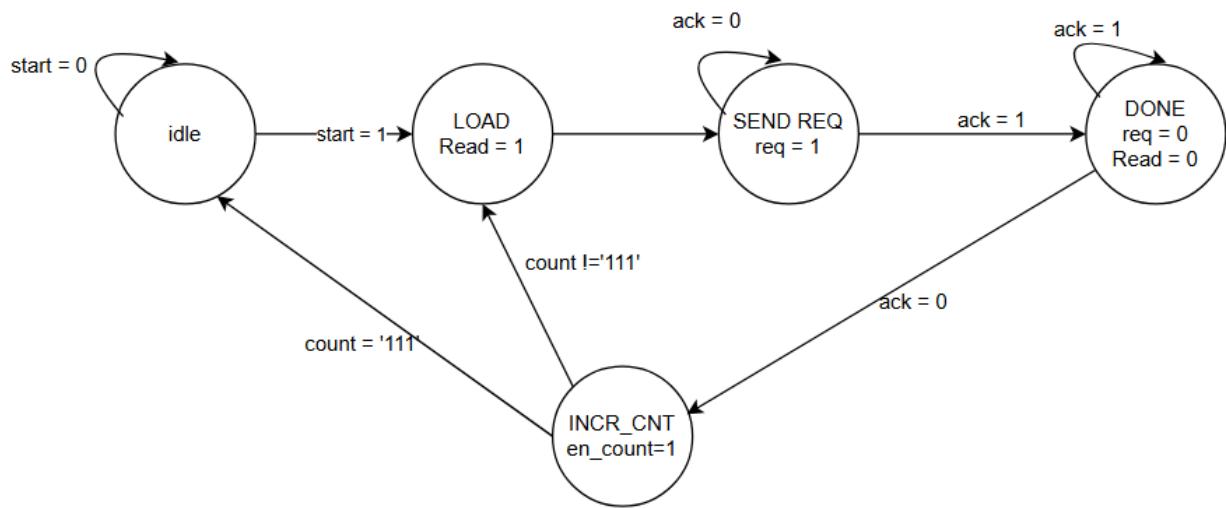
Il sistema A deve inoltrare al sistema B gli indirizzi di memoria memorizzati nella ROM, accedendo ad essa tramite un contatore in modo da gestire gli indirizzi. Prima di comunicare con B è necessario instaurare un protocollo ai fini di poter comunicare. Il sistema A è stato realizzato utilizzando la ROM descritta in appendice, effettuando una piccola modifica che consiste nell'inserimento di 8 indirizzi a 4 bit, e il contatore generico utilizzato precedentemente. E' stato necessario creare una unità di controllo che consente di instaurare un protocollo con il sistema B e poi scambiarsi i messaggi.



Progetto e architettura

L'automa dell'unità di controllo del sistema A si presenta con 5 stati :

- IDLE: il sistema resta in idle finché non riceve il segnale start dall'esterno.
- LOAD: il sistema prepara il dato, alzando il segnale read = '1'.
- SEND REQ: il sistema avendo già pronto il dato si prepara ad inviare il messaggio al sistema B. Viene avviato il protocollo handshaking, alzando il segnale di req, e il sistema resta in attesa in questo stato finché il sistema B non risponde con il segnale di ack.
- DONE: Dopo aver ricevuto la risposta del sistema B abbassa il segnale req e abbassa il segnale di read. Il sistema resta in questo stato finché il segnale di ack non viene abbassato, solo allora il sistema B sarà pronto a ricevere un altro dato.
- INCR_CNT: Nell'ultimo stato la trasmissione può considerarsi conclusa, quindi si incrementa il contatore e il sistema controlla se ci sono altri messaggi da mandare, quindi se il contatore è arrivato a 8 si ferma ritornando nello stato di idle, altrimenti ritorna in load pronto per inoltrare un altro messaggio.



Il sistema A completo non è altro che un componente strutturale contenente l'unità di controllo, la ROM e il contatore.

SISTEMA B

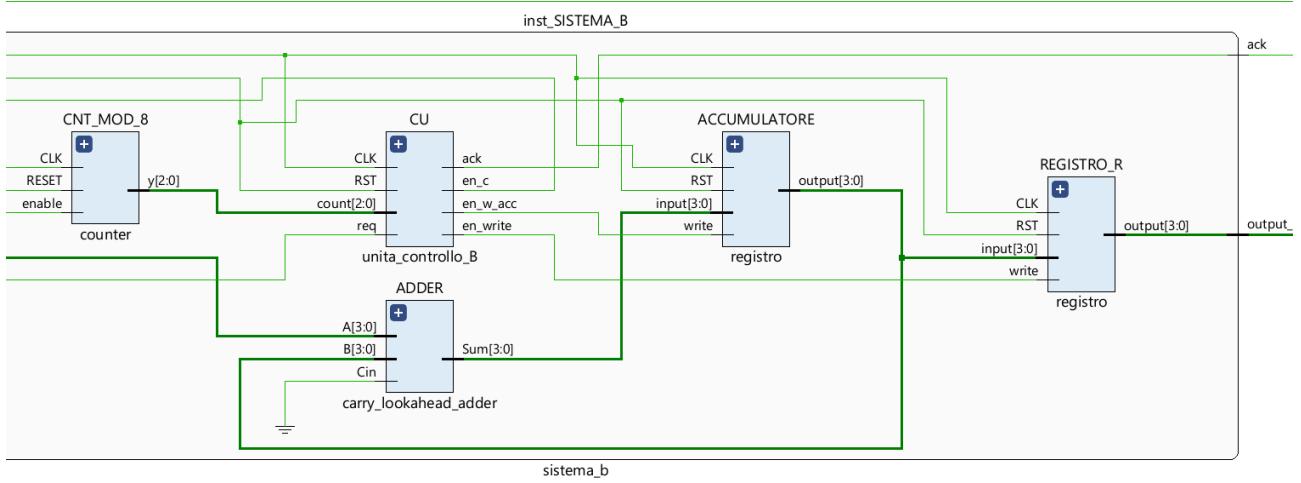
Il sistema B è composto da un contatore modulo 8, un sommatore, un accumulatore, un registro e infine un'unità di controllo.

Il contatore modulo 8 è lo stesso presente nell'appendice, è stato utilizzato per tenere traccia dei risultati ricevuti.

Il sistema dispone di due registri: l'accumulatore, che memorizza la somma parziale e la retroazione nell'addizionatore per sommare progressivamente ogni dato ricevuto dal sistema A, e il registro R, che memorizza il risultato definitivo.

L'addizionatore è stato presentato all'inizio, è stata usata una macchina carry look ahead.

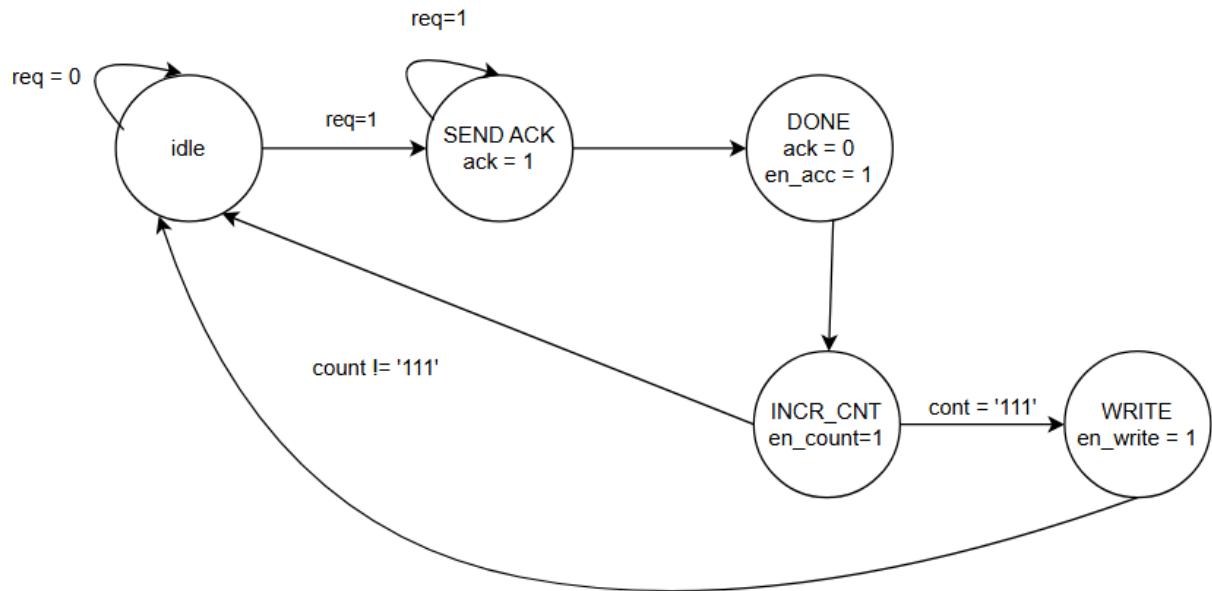
Infine, l'unità di controllo B gestisce il comportamento del sistema, coordinando il flusso dei dati e il funzionamento dei vari componenti.



Progetto e architettura

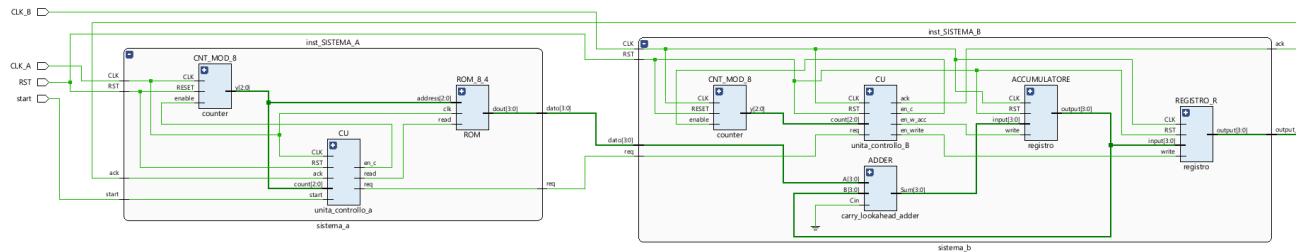
L'automa dell'unità di controllo del sistema B si presenta con 5 stati :

- IDLE: il sistema è in attesa di una richiesta (req) dal sistema A.
- SEND ACK: il sistema alza ACK per notificare al sistema A che ha ricevuto il segnale di req ed è pronto a ricevere, dunque si attende che il sistema A abbassa il segnale di req per passare allo stato di DONE.
- DONE: il sistema abbassa il segnale di ack, effettua la somma ed abilita l'accumulatore in modo da salvare il risultato.
- INCR_CNT: Il sistema incrementa il contatore e controlla se ci sono altri messaggi da ricevere, quindi se il contatore è arrivato a 8 si ferma andando nello stato di WRITE per salvare il risultato della somma, altrimenti ritorna in idle pronto per ricevere un altro messaggio.
- WRITE: il sistema salva la somma in un registro per poi andare in idle in attesa dell'inizio di una nuova comunicazione.



Il sistema B completo non è altro che un componente strutturale contenente l'unità di controllo, due registri, un contatore e la macchina sommatore.

SISTEMA COMPLESSIVO



Implementazione

Sistema complessivo:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```

```

entity sistema_complessivo is

```

```

Port (

```

```

    CLK_A : in STD_LOGIC; -- clock sistema A

```

```

    CLK_B : in STD_LOGIC; -- clock sistema B

```

```

    RST : in STD_LOGIC; -- segnale di reset

```

```
start : in STD_LOGIC; -- segnale di start
output_finale : out STD_LOGIC_VECTOR(3 downto 0) -- somma calcolata, contenuta nel registro R
);
end sistema_complessivo;
```

architecture Structural of sistema_complessivo is

```
component sistema_a
Port (
    CLK : in STD_LOGIC;
    RST : in STD_LOGIC;
    start : in STD_LOGIC;
    ack : in STD_LOGIC;
    req : out STD_LOGIC;
    dato : out STD_LOGIC_VECTOR(3 downto 0)
);
end component;
```

```
component sistema_b
Port (
    CLK : in STD_LOGIC;
    RST : in STD_LOGIC;
    req : in STD_LOGIC;
    ack : out STD_LOGIC;
    output_finale : out STD_LOGIC_VECTOR(3 downto 0);
    dato : in STD_LOGIC_VECTOR(3 downto 0)
);
end component;
```

-- segnali interni per l'interconnessione tra sistema_a e sistema_b

```
signal req_signal : STD_LOGIC;
signal ack_signal : STD_LOGIC;
```

```
signal dato_signal : STD_LOGIC_VECTOR(3 downto 0);
```

```
begin
```

```
-- Istanza del sistema_a
```

```
inst_SISTEMA_A : sistema_a
```

```
port map (
```

```
    CLK => CLK_A,
```

```
    RST => RST,
```

```
    start => start,
```

```
    ack => ack_signal,
```

```
    req => req_signal,
```

```
    dato => dato_signal
```

```
);
```

```
-- Istanza del sistema_b
```

```
inst_SISTEMA_B : sistema_b
```

```
port map (
```

```
    CLK => CLK_B,
```

```
    RST => RST,
```

```
    req => req_signal,
```

```
    ack => ack_signal,
```

```
    output_finale => output_finale,
```

```
    dato => dato_signal
```

```
);
```

```
end Structural;
```

Sistema A:

```
library IEEE;
```

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

```
;
```

```

entity sistema_a is
    Port ( CLK : in STD_LOGIC; -- clock
            RST : in STD_LOGIC; -- reset
            start : in STD_LOGIC; -- segnale start
            ack : in STD_LOGIC; -- segnale ack
            req : out STD_LOGIC; -- segnale req
            dato : out STD_LOGIC_VECTOR (3 downto 0)); -- dato da inviare a B
end sistema_a;

```

architecture Structural of sistema_a is

```

component unita_controllo_a
    Port (
        CLK      : in std_logic;
        RST      : in std_logic;
        start    : in std_logic; -- quando inizia il programma -> a leggere da rom
        count    : in std_logic_vector (2 downto 0); -- valore conteggio contatore mod 8
        ack      : in std_logic; -- Acknowledgment da sistema B

        req      : out std_logic; -- Req al sistema B
        read     : out std_logic; -- abilita lettura dalla Rom
        en_c    : out std_logic -- abilita conteggio contatore indirizzi rom
    );
end component;

```

```

component ROM
    port(
        clk : in std_logic;
        address : in std_logic_vector(2 downto 0);
        read : in std_logic;
        dout : out std_logic_vector(3 downto 0)
    );

```

```

end component;

component counter
generic(
    m : integer := 8; -- questo ci dice fino a che conta il counter
    n : integer := 3 -- questo ci dà la lunghezza del vector, il numero di bit
);
Port (
    CLK : in std_logic;
    RESET: in std_logic;
    enable: in std_logic;
    y: out std_logic_vector(n-1 downto 0) -- uscita contatore
);
end component;

--segnali interni per le interconnessioni
signal read : std_logic;
signal address : std_logic_vector(2 downto 0);
signal en_c : std_logic;

begin

CU : unita_controllo_a
port map (
    CLK => CLK,
    RST => RST,
    start => start,
    count => address,
    ack => ack,
    req => req,
    read => read,

```

```
    en_c => en_c  
);
```

```
ROM_8_4 : ROM  
port map(  
    clk => CLK,  
    address => address,  
    read => read,  
    dout => dato  
);
```

```
CNT_MOD_8 : counter  
generic map (  
    m => 8,  
    n => 3  
)  
port map (  
    CLK => CLK,  
    RESET => RST,  
    enable => en_c,  
    y => address  
);
```

```
end Structural;
```

Unità di controllo A:

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

```

entity unita_controllo_a is

Port (
    CLK      : in std_logic;
    RST      : in std_logic;
    start    : in std_logic; -- quando inizia il programma -> a leggere da rom
    count    : in std_logic_vector (2 downto 0); -- valore conteggio contatore mod 8
    ack      : in std_logic; -- Acknowledgment da sistema B

    req      : out std_logic; -- Req al sistema B
    read     : out std_logic; -- abilita lettura dalla Rom
    en_c    : out std_logic -- abilita conteggio contatore indirizzi rom
);

end unita_controllo_a;

```

```

architecture Behavioral of unita_controllo_a is

type state_type is (IDLE, LOAD, SEND_REQ, DONE, INCR_CNT);

signal current_state : state_type := IDLE;
signal next_state : state_type;

begin

```

```

rf: process(CLK, RST)
begin
    if RST = '1' then
        current_state <= IDLE;

```

```
elsif rising_edge(CLK) then
    current_state <= next_state;
end if;
end process;
```

```
update: process(current_state, start, ack)
```

```
begin
```

```
case current_state is
    when IDLE =>
        en_c <= '0';
        read <= '0';
        req <= '0';
        if start = '1' then
            next_state <= LOAD;
        end if;
```

```
when LOAD =>
```

```
        en_c <= '0';
        read <= '1';
        req <= '0';
        next_state <= SEND_REQ;
```

```
when SEND_REQ =>
```

```
        en_c <= '0';
        read <= '1';
        req <= '1';
```

```
if ack = '1' then  
    next_state <= DONE;  
  
else  
    next_state <= SEND_REQ;  
  
end if;
```

when DONE =>

```
en_c <= '0';  
  
read <= '0';  
  
req <= '0';  
  
if ack = '0' then  
    next_state <= INCR_CNT;  
  
else  
    next_state <= DONE;  
  
end if;
```

when INCR_CNT =>

```
en_c <= '1';  
  
read <= '0';  
  
req <= '0';  
  
if count = "111" then  
    next_state <= IDLE;  
  
else  
    next_state <= LOAD;  
  
end if;
```

```

when others =>

    en_c <= '0';

    read <= '0';

    req <= '0';

    next_state <= IDLE;

end case;

end process;

end Behavioral;

```

Sistema B:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sistema_b is
    Port ( CLK : in STD_LOGIC; -- clock
            RST : in STD_LOGIC; -- reset
            req : in STD_LOGIC; -- segnale req
            ack : out STD_LOGIC; -- segnale ack
            output_finale : out std_logic_vector(3 downto 0); -- output del valore contenuto nel registro R, ovvero
            risultato della somma
            dato : in STD_LOGIC_VECTOR (3 downto 0) -- dato proveniente dal sistema A
        );
end sistema_b;

```

architecture Structural of sistema_b is

```

component unita_controllo_b
    Port (
        CLK : in std_logic;
        RST : in std_logic;

```

```

count : in std_logic_vector (2 downto 0); -- valore sommatore mod 8 per capire quanti dati devo
ancora ricevere

req : in std_logic; -- req da parte di A per B

ack : out std_logic; -- ack di B per A

en_c : out std_logic; -- abilita conteggio mod 8

en_w_acc : out std_logic; -- abilita scrittura accumulatore

en_write : out std_logic -- abilita scrittura registro finale

);

end component;

```

component registro

```

Port (
    CLK : in std_logic;
    RST : in std_logic;
    input : in std_logic_vector(3 downto 0);
    write : in std_logic;
    output: out std_logic_vector(3 downto 0)
);

end component;

```

component counter

```

generic(
    m : integer := 8; -- questo ci dice fino a che conta il counter
    n : integer := 3 -- questo ci dà la lunghezza del vector, il numero di bit
);

Port (
    CLK : in std_logic;
    RESET: in std_logic;
    enable: in std_logic;
    y: out std_logic_vector(n-1 downto 0) -- uscita contatore
);

```

```

end component;

component carry_loookahead_adder
Generic (N : integer := 4);
Port (
    A    : in std_logic_vector(N-1 downto 0);
    B    : in std_logic_vector(N-1 downto 0);
    Cin  : in std_logic;
    Sum  : out std_logic_vector(N-1 downto 0);
    Cout : out std_logic
);
end component;

```

```

--segnali interni per le interconnessioni
signal count : std_logic_vector(2 downto 0);
signal en_c : std_logic;
signal en_w_acc : std_logic;
signal write : std_logic;
signal sum_p : std_logic_vector(3 downto 0);
signal data_acc : std_logic_vector(3 downto 0);
signal overflow : std_logic;

```

```
begin
```

```

CU : unita_controllo_b
port map (
    CLK => CLK,
    RST => RST,
    count => count,
    req => req,
    ack => ack,
    en_c => en_c,

```

```
    en_w_acc => en_w_acc,  
    en_write => write  
);
```

```
REGISTRO_R : registro  
port map (  
    CLK => CLK,  
    RST => RST,  
    input => data_acc,  
    write => write,  
    output => output_finale  
);
```

```
ACCUMULATORE : registro  
port map (  
    CLK => CLK,  
    RST => RST,  
    input => sum_p,  
    write => en_w_acc,  
    output => data_acc  
);
```

```
CNT_MOD_8 : counter  
generic map (  
    m => 8,  
    n => 3  
)  
port map (  
    CLK => CLK,  
    RESET => RST,  
    enable => en_c,  
    y => count
```

```

    );
ADDER : carry_lookahead_adder
generic map (
    N => 4
)
port map (
    A => dato,
    B => data_acc,
    Cin => '0',
    Sum => sum_p,
    Cout => overflow
);

```

end Structural;

Unità di controllo B:

```

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

```

entity unita_controllo_B is

Port (

```

    CLK : in std_logic;
    RST : in std_logic;

```

count : in std_logic_vector (2 downto 0); -- valore sommatore mod 8 per capire quanti dati devo ancora ricevere

```
    req : in std_logic; -- req da parte di A per B
```

```
    ack : out std_logic; -- ack di B per A
```

```
    en_c : out std_logic; -- abilita conteggio mod 8
```

```
    en_w_acc : out std_logic; -- abilita scrittura accumulatore
```

```
    en_write : out std_logic -- abilita scrittura registro finale
```

```
 );
end unita_controllo_B;

architecture Behavioral of unita_controllo_B is
```

type state_type is (IDLE, SEND_ACK, DONE, INCR_CNT, WRITE);
signal current_state : state_type := IDLE;
signal next_state : state_type;

```
begin
```

```
rfr: process(CLK, RST)
begin
  if RST = '1' then
    current_state <= IDLE;
  elsif rising_edge(CLK) then
    current_state <= next_state;
  end if;
end process;
```

```
update: process(current_state, req)
begin
```

```
  case current_state is
    when IDLE =>
      ack <= '0';
      en_c <= '0';
      en_w_acc <= '0';
      en_write <= '0';
    if req = '1' then
      next_state <= SEND_ACK;
    else
      next_state <= IDLE;
```

```
end if;
```

```
when SEND_ACK =>
```

```
    ack <= '1';
    en_c <= '0';
    en_w_acc <= '0';
    en_write <= '0';
    if req = '0' then
        next_state <= DONE;
    else
        next_state <= SEND_ACK;
    end if;
```

```
when DONE =>
```

```
    ack <= '0';
    en_c <= '0';
    en_w_acc <= '1';
    en_write <= '0';
    next_state <= INCR_CNT;
```

```
when INCR_CNT =>
```

```
    ack <= '0';
    en_c <= '1';
    en_w_acc <= '0';
    en_write <= '0';
    if count = "111" then
        next_state <= WRITE;
    else
        next_state <= IDLE;
    end if;
```

```
when WRITE =>
```

```

    ack <= '0';
    en_c <= '0';
    en_w_acc <= '0';
    en_write <= '1';
    next_state <= IDLE;

```

when others =>

```

    ack <= '0';
    en_c <= '0';
    en_w_acc <= '0';
    en_write <= '0';
    next_state <= IDLE;
end case;
end process;

```

end Behavioral;

Registro:

Il registro e l'accumulatore non sono altro che due registri, solo che uno funge da registro mentre il secondo accumula di volta in volta la somma.

```

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

entity registro is
  Port (
    CLK : in std_logic;
    RST : in std_logic;
    input : in std_logic_vector(3 downto 0);
    write : in std_logic;
    output: out std_logic_vector(3 downto 0)
  );
end registro;

```

```

architecture Behavioral of registro is

    signal reg : std_logic_vector(3 downto 0);

begin

    main: process(CLK)
        begin
            if(CLK'event and CLK='1') then
                if(RST = '1') then
                    reg <= "0000";
                elsif(write = '1') then
                    reg <= input;
                end if;
            end if;
        end process;

        output <= reg;
    end Behavioral;

```

Simulazione

Sono state effettuate due simulazioni, la prima col clock A più veloce del clock B mentre la seconda col clock A più lento del clock B. Per eseguire il test bench è stato sufficiente dare il segnale di start alla macchina e attendere che evolvesse come desiderato.

Clock A più veloce:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

```

entity tb_sistema_complessivo is
end tb_sistema_complessivo;

```

```
architecture Behavioral of tb_sistema_complessivo is
```

```
-- Component under test (CUT)
component sistema_complessivo
  Port (
    CLK_A : in STD_LOGIC;
    CLK_B : in STD_LOGIC;
    RST : in STD_LOGIC;
    start : in STD_LOGIC;
    output_finale : out STD_LOGIC_VECTOR(3 downto 0)
  );
end component;
```

```
-- Testbench signals
signal CLK_A_tb : STD_LOGIC := '0';
signal CLK_B_tb : STD_LOGIC := '0';
signal RST_tb : STD_LOGIC := '1';
signal start_tb : STD_LOGIC := '0';
signal output_finale_tb : STD_LOGIC_VECTOR(3 downto 0);
```

```
-- Clock periods
constant CLK_A_PERIOD : time := 15 ns; -- CLK_A piu' veloce
constant CLK_B_PERIOD : time := 25 ns; -- CLK_B piu' lento
```

```
begin
```

```
-- Instanza del sistema complessivo
CUT : sistema_complessivo
port map (
  CLK_A => CLK_A_tb,
  CLK_B => CLK_B_tb,
  RST => RST_tb,
  start => start_tb,
```

```
    output_finale => output_finale_tb
);
```

```
-- Processo per generare CLK_A
```

```
process
begin
  while true loop
    CLK_A_tb <= '0';
    wait for CLK_A_PERIOD / 2;
    CLK_A_tb <= '1';
    wait for CLK_A_PERIOD / 2;
  end loop;
end process;
```

```
-- Processo per generare CLK_B
```

```
process
begin
  while true loop
    CLK_B_tb <= '0';
    wait for CLK_B_PERIOD / 2;
    CLK_B_tb <= '1';
    wait for CLK_B_PERIOD / 2;
  end loop;
end process;
```

```
-- Stimuli process
```

```
process
begin
  -- Reset iniziale
  RST_tb <= '1';
  wait for 50 ns;
  RST_tb <= '0';
```

```
-- Avvio del sistema
```

```
wait for 20 ns;
```

```
start_tb <= '1';
```

```
wait for 10 ns;
```

```
start_tb <= '0';
```

```
-- Simulazione in esecuzione per un po'
```

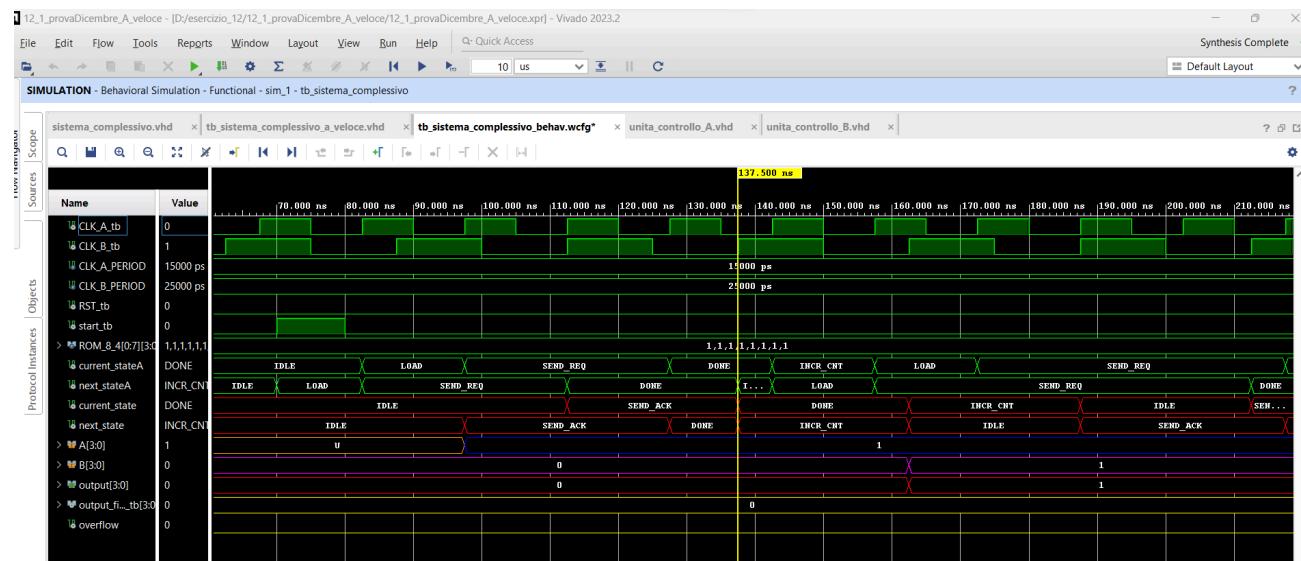
```
wait for 900 ns;
```

```
-- Fine simulazione
```

```
wait;
```

```
end process;
```

```
end Behavioral;
```



Clock B più veloce:

```
library IEEE;
```

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

```
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```

entity tb_sistema_compressivo is
end tb_sistema_compressivo;

architecture Behavioral of tb_sistema_compressivo is

-- Component under test (CUT)
component sistema_compressivo
    Port (
        CLK_A : in STD_LOGIC;
        CLK_B : in STD_LOGIC;
        RST : in STD_LOGIC;
        start : in STD_LOGIC;
        output_finale : out STD_LOGIC_VECTOR(3 downto 0)
    );
end component;

-- Testbench signals
signal CLK_A_tb : STD_LOGIC := '0';
signal CLK_B_tb : STD_LOGIC := '0';
signal RST_tb : STD_LOGIC := '1';
signal start_tb : STD_LOGIC := '0';
signal output_finale_tb : STD_LOGIC_VECTOR(3 downto 0);

-- Clock periods
constant CLK_A_PERIOD : time := 25 ns; -- CLK_A piu' lento
constant CLK_B_PERIOD : time := 15 ns; -- CLK_B piu' veloce

begin

-- Instanza del sistema compressivo
CUT : sistema_compressivo
    port map (

```

```
CLK_A => CLK_A_tb,  
CLK_B => CLK_B_tb,  
RST => RST_tb,  
start => start_tb,  
output_finale => output_finale_tb  
);
```

-- Processo per generare CLK_A

```
process  
begin  
while true loop  
    CLK_A_tb <= '0';  
    wait for CLK_A_PERIOD / 2;  
    CLK_A_tb <= '1';  
    wait for CLK_A_PERIOD / 2;  
end loop;  
end process;
```

-- Processo per generare CLK_B

```
process  
begin  
while true loop  
    CLK_B_tb <= '0';  
    wait for CLK_B_PERIOD / 2;  
    CLK_B_tb <= '1';  
    wait for CLK_B_PERIOD / 2;  
end loop;  
end process;
```

-- Stimuli process

```
process  
begin
```

```
-- Reset iniziale
```

```
RST_tb <= '1';
```

```
wait for 50 ns;
```

```
RST_tb <= '0';
```

```
-- Avvio del sistema
```

```
wait for 20 ns;
```

```
start_tb <= '1';
```

```
wait for 10 ns;
```

```
start_tb <= '0';
```

```
-- Simulazione in esecuzione per un po'
```

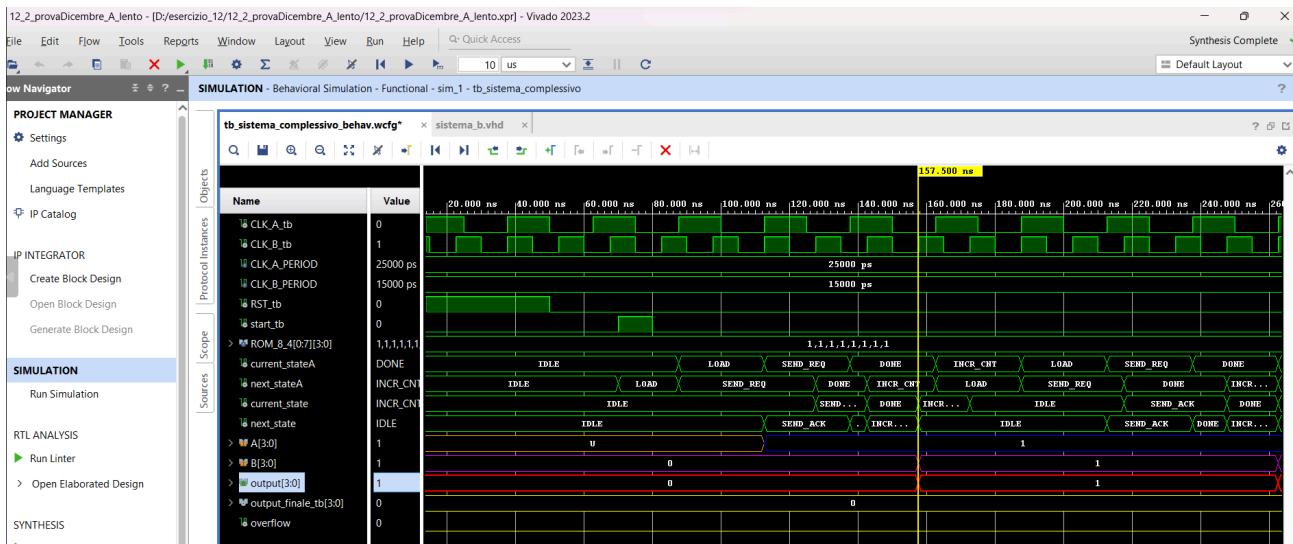
```
wait for 1000 ns;
```

```
-- Fine simulazione
```

```
wait;
```

```
end process;
```

```
end Behavioral;
```

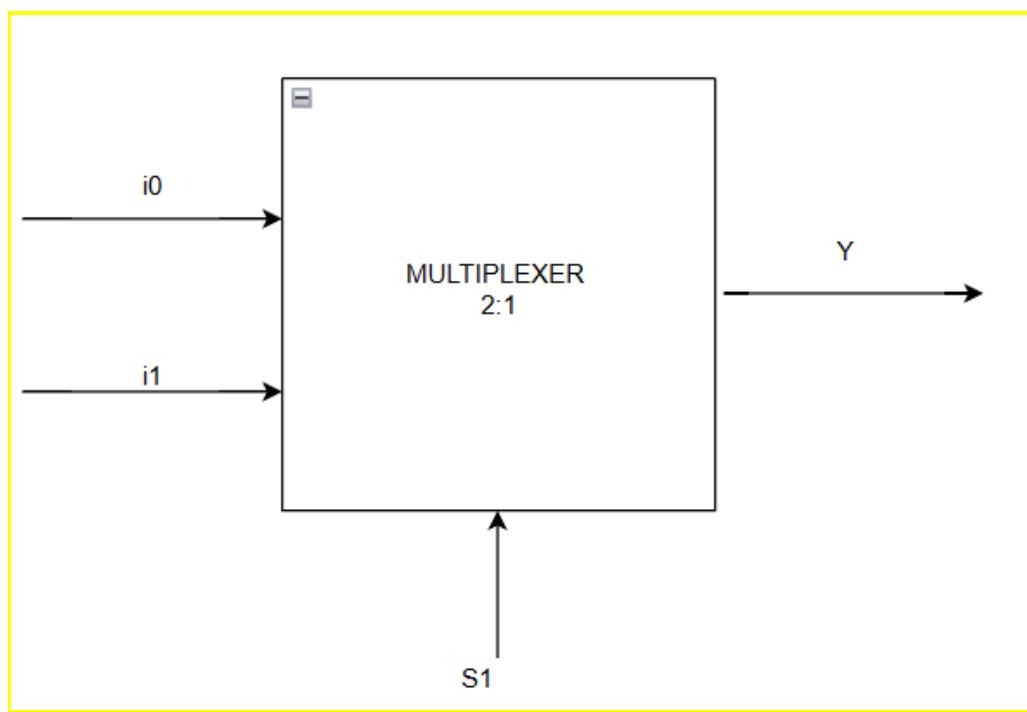


Appendice

MUX2_1

Progetto e architettura

Il multiplexer 2_1 è una macchina combinatoria notevole che prende in ingresso n-ingressi dati e riporta in uscita solo un dato che verrà selezionato tramite apposita linea di selezione. Abbiamo realizzato un multiplexer indirizzabile, questo tramite apposite linee di selezione(in questo caso una linea di selezione) è possibile trasmettere in uscita il dato in ingresso. Questo si differisce dal mux lineare che ha "n" linee dato e "n" linee di controllo(una per ciascun ingresso). Utilizzando un **decoder** in combinazione con le linee di selezione, è possibile realizzare un multiplexer indirizzabile, **riducendo il numero di linee di selezione** da controllare. Questo rende il circuito più efficiente in termini di complessità e cablaggio.



Implementazione

Per l'implementazione della macchina è stato scelto un modello Behavioral:

```
entity MUX_2_1 is
    Port(
        A : in STD_LOGIC; -- Ingresso A
        B : in STD_LOGIC; -- Ingresso B
        S : in STD_LOGIC; -- Selettore
        Y : out STD_LOGIC --Uscita
    );
end MUX_2_1;
```

```

architecture Behavioral of MUX_2_1 is
begin
    -- Logica del MUX 2:1
    process(A, B, S)
    begin
        if S = '0' then
            Y <= A; -- Se il selettore è 0, l'uscita è A
        else
            Y <= B; -- Se il selettore è 1, l'uscita è B
        end if;
    end process;
end Behavioral;

```

MULTIPLEXER4_1

Progetto e architettura

Analogamente al multiplexer 2_1 è stato preferito l'approccio behavioral, nonostante sarebbe stato possibile utilizzare l'approccio structural utilizzando 3 multiplexer 2_1.

Implementazione

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Entity declaration for the 4:1 MUX
entity Mux4_1 is
    Port ( A0 : in STD_LOGIC;
           A1 : in STD_LOGIC;
           A2 : in STD_LOGIC;
           A3 : in STD_LOGIC;
           S0 : in STD_LOGIC;
           S1 : in STD_LOGIC;
           Y : out STD_LOGIC);
end Mux4_1;

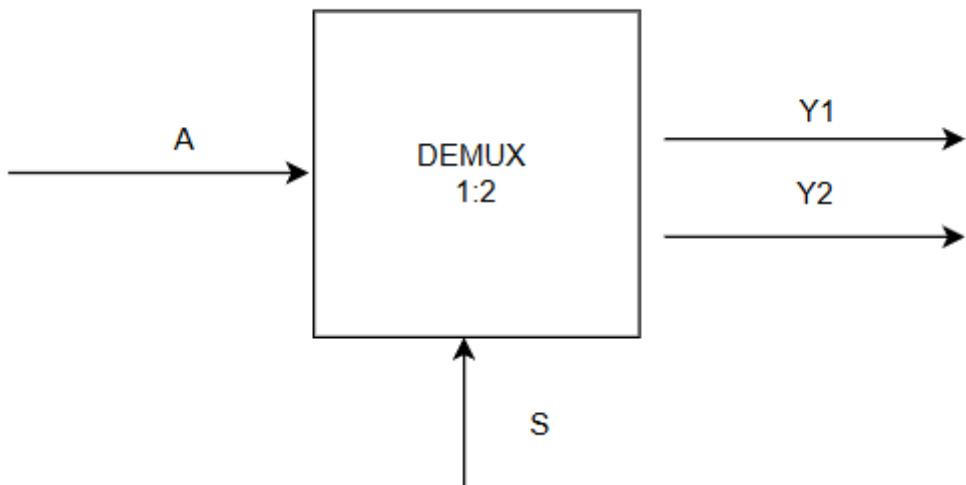
```

```
architecture Behavioral of Mux4_1 is
begin
process(A0,A1,A2,A3,S0,S1)
begin
if S0 = '0' and S1 = '0' then
    Y <= A0;
elsif S0 = '0' and S1 = '1' then
    Y <= A1;
elsif S0 = '1' and S1 = '0' then
    Y <= A2;
elsif S0 = '1' and S1 = '1' then
    Y <= A3;
end if;
end process;
end Behavioral;
```

DEMUX1_2

Progetto e architettura

Il demultiplexer è la versione speculare del multiplexer. Mentre il multiplexer prende in ingresso “n” bit e ne trasmette solo uno sulla linea di uscita selezionata, il demultiplexer riceve un solo bit in ingresso e, tramite apposite linee di selezione, lo instrada verso una delle uscite disponibili.



Implementazione

Per progettare il demux è stato scelta l'approccio Behavioral:

```
entity DEM1_2 is
```

```
Port(
```

```
  A : in STD_LOGIC;
```

```
  S1 : in STD_LOGIC;
```

```
  Y : out STD_LOGIC_VECTOR (1 downto 0) -- uscite
```

```
);
```

```
-- S1 Y
```

```
-- 0 MUX1
```

```
-- 1 MUX2
```

```
end DEM1_2;
```

```
architecture Behavioral of DEM1_2 is
```

```
begin
```

```
  process (S1, A)
```

```
    variable temp_Y : STD_LOGIC_VECTOR(1 downto 0);
```

```
    begin
```

```
      -- Impostiamo tutte le uscite su '0' per essere sicuri che non rimangano attive
```

```

temp_Y := "00";

-- Logica per selezionare l'uscita
if (S1 = '0') then
    temp_Y(0) := A;
else
    temp_Y(1) := A;
end if;

Y<= temp_Y;

end process;

```

end Behavioral;

DEMUX1_4

Progetto e architettura

Analogamente al demultiplexer 1_2 è stato utilizzato un approccio behavioral. A differenza del demux 1:2, il demux 1:4 dispone di **due linee di selezione e quattro linee di uscita**. In base al valore specificato sulle linee di selezione, **il dato in ingresso viene instradato su una sola delle uscite**, mentre le altre restano disattivate.

Implementazione

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```

entity Demux1_4 is

```

Port (
    A : in STD_LOGIC;          -- Ingresso
    Y : out STD_LOGIC_VECTOR(0 to 3); -- Uscite
    S1 : in STD_LOGIC;         -- Selezione
    S2 : in STD_LOGIC          -- Selezione
);

```

end Demux1_4;

architecture Behavioral of Demux1_4 is

```

begin
process (S1, S2, A)
    variable temp_Y : STD_LOGIC_VECTOR(0 to 3); -- Variabile per memorizzare le uscite temporanee
begin
    -- Impostiamo tutte le uscite su '0' per evitare che restino attive
    temp_Y := "0000";

    -- Logica per selezionare l'uscita in base ai segnali di selezione S1 e S2
    if (S1 = '0' and S2 = '0') then
        temp_Y(0) := A; -- MUX1
    elsif (S1 = '0' and S2 = '1') then
        temp_Y(1) := A; -- MUX2
    elsif (S1 = '1' and S2 = '0') then
        temp_Y(2) := A; -- MUX3
    elsif (S1 = '1' and S2 = '1') then
        temp_Y(3) := A; -- MUX4
    end if;
    -- Assegniamo il valore alla variabile segnale
    Y <= temp_Y;
end process;
end Behavioral;

```

ROM

Progetto e architettura

La rom è una macchina combinatoria notevole che consente solo di leggere i dati che sono contenuti al suo interno.

La macchina presenta 3 SEGNALI :

- ADDR: un ingresso a **4 bit** che specifica la locazione di memoria da cui leggere il dato.
- DATA:un'uscita a 8 bit che restituisce il valore contenuto nella posizione di memoria selezionata da ADDR.
- RST : n segnale di reset asincrono che, quando attivo (RST='1'), forza l'uscita DATA al valore della prima locazione di memoria (ROM(0)).

```
Implementazione
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
entity rom_16_8 is
  Port ( ADDR : in STD_LOGIC_VECTOR (3 downto 0);
         DATA : out STD_LOGIC_VECTOR (7 downto 0);
         RST : in STD_LOGIC);
end rom_16_8;
```

```
architecture Behavioral of rom_16_8 is
```

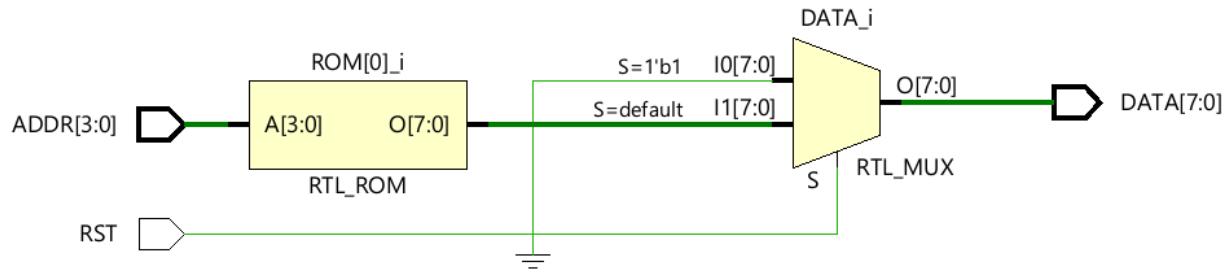
```
type rom_type is array (0 to 15) of std_logic_vector(7 downto 0);
constant ROM : rom_type := (
  X"00",-- 0000-0000
  X"11",-- 0001-0001
  X"22",-- 0010-0010
  X"33",-- 0011-0011
  X"44",--0100-0100
  X"55", -- 0101-0101
  X"66", -- 0110-0110
  X"77", -- 0111-0111
  X"88", -- 1000-1000
  X"99", -- 1001-1001
  X"aa", -- 1010-1010
  X"bb", -- 1011-1011
  X"cc", -- 1100-1100
  X"dd",-- 1101-1101
  X"ee",-- 1110-1110
  X"ff"--1111-1111
);
```

```

begin
process(ADDR, RST)
begin
if(RST = '1')
then DATA <= ROM(0);
else
DATA <= ROM(to_integer(unsigned(ADDR)));
end if;
end process;
end Behavioral;

```

SCHEMATIC :



Full Adder

Progetto e architettura

Il Full Adder è un circuito combinatorio che somma due bit di ingresso (input_a e input_b) e un riporto in ingresso (carry_in).

Produce come uscita:

- sum: il bit risultante della somma.
- carry_out: il riporto verso il bit successivo.

Viene comunemente utilizzato come blocco base per realizzare addizionatori a più bit concatenando più Full Adder.

Implementazione

-- Full Adder a 1 bit
-- Somma due bit di ingresso più un riporto in ingresso
ENTITY full_adder IS

```

PORT (
    input_a, input_b : IN STD_LOGIC;      -- Bit di ingresso da sommare
    carry_in : IN STD_LOGIC;             -- Riporto in ingresso
    carry_out, sum : OUT STD_LOGIC      -- Riporto in uscita e risultato della somma
);
END full_adder;

ARCHITECTURE rtl OF full_adder IS

BEGIN
    -- Calcolo del bit di somma usando XOR a tre ingressi
    sum <= input_a XOR input_b XOR carry_in;

    -- Calcolo del riporto in uscita
    -- Si ha riporto quando almeno due degli ingressi sono a 1
    carry_out <= (input_a AND input_b) OR (carry_in AND (input_a XOR input_b));

```

END rtl;

Adder Carry Ripple

Progetto e architettura

Il Ripple Carry Adder (sommatore a propagazione di riporto) è un circuito combinatorio che nella versione proposta di seguito, somma due vettori di 8 bit (X e Y) e un riporto in ingresso (c_{in}).

Come risultato fornisce:

- Z : il vettore somma a 8 bit.
- c_{out} : il riporto in uscita, utile se si vogliono concatenare più blocchi da 8 bit.

Si chiama "Ripple Carry" perché il riporto ("carry") si propaga sequenzialmente da un bit al successivo.

Implementazione

Utilizza 8 Full Adder a 1 bit, collegati in serie e ognuno di essi somma:

- un bit di X
- un bit di Y
- il riporto proveniente dal Full Adder precedente.

Il circuito è organizzato nel seguente modo:

- FA0: somma X(0), Y(0) e c_in. Il riporto generato (internal_carry(0)) va a FA1.
- FA1–FA6: sommano rispettivamente i bit 1–6, usando il riporto del Full Adder precedente.
- FA7: somma X(7), Y(7), e il riporto da FA6, producendo il c_out.

Per i Full Adder da 1 a 6, si usa un costrutto FOR GENERATE, che semplifica la scrittura ripetitiva e il vettore internal_carry (8 bit) gestisce i riporti intermedi tra i vari Full Adder.

Di seguito il codice VHDL:

```
-- Sommatore a propagazione di riporto (Ripple Carry Adder) a 8 bit
-- Somma due operandi a 8 bit con riporto in ingresso

ENTITY ripple_carry IS
  PORT (
    X, Y : IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- Operandi di ingresso
    c_in : IN STD_LOGIC;                      -- Riporto in ingresso
    c_out : OUT STD_LOGIC;                     -- Riporto in uscita
    Z : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)        -- Risultato della somma
  );
END ripple_carry;
```

```
ARCHITECTURE structural OF ripple_carry IS
  COMPONENT full_adder IS
    PORT (
      input_a, input_b : IN STD_LOGIC;      -- Bit di ingresso da sommare
      carry_in : IN STD_LOGIC;              -- Riporto in ingresso
      carry_out, sum : OUT STD_LOGIC;       -- Riporto in uscita e risultato della somma
    );
  END COMPONENT;

  -- Segnali interni per la propagazione del riporto tra i full adder
  SIGNAL internal_carry : STD_LOGIC_VECTOR(7 DOWNTO 0);
```

BEGIN

```
-- Full adder per il bit meno significativo (bit 0)
```

```

-- Utilizza il riporto in ingresso esterno
FA0 : full_adder PORT MAP(X(0), Y(0), c_in, internal_carry(0), Z(0));

-- Full adder per i bit intermedi (bit 1-6)
-- Ogni full adder riceve il riporto dal full adder precedente
FA1to6 : FOR i IN 1 TO 6 GENERATE
    FA : full_adder PORT MAP(X(i), Y(i), internal_carry(i - 1), internal_carry(i), Z(i));
END GENERATE;

-- Full adder per il bit più significativo (bit 7)
-- Il riporto in uscita diventa il riporto in uscita dell'intero sommatore
FA7 : full_adder PORT MAP(X(7), Y(7), internal_carry(6), c_out, Z(7));

END structural;

```

Registro PIPO

Progetto e architettura

Il registro a 8 bit memorizza un valore di 8 bit (data_in) in modo sincrono al fronte di salita del clock (clock), con:

- Reset sincrono (reset) per azzerare il contenuto.
- Segnale di caricamento (enable) per decidere se aggiornare o mantenere il valore corrente.

L'uscita del registro è sempre disponibile su data_out.

Implementazione

-- Registro a 8 bit con reset sincrono e segnale di caricamento

ENTITY registro8 IS

```

PORT (
    data_in : IN STD_LOGIC_VECTOR(7 DOWNTO 0);      -- Dati in ingresso
    clock, reset, enable : IN STD_LOGIC;            -- Segnali di controllo
    data_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)      -- Dati in uscita
);

```

```
END registro8;
```

```
ARCHITECTURE behavioural OF registro8 IS
```

```
    SIGNAL internal_value : STD_LOGIC_VECTOR(7 DOWNTO 0); -- Valore memorizzato internamente
```

```
BEGIN
```

```
    -- Processo di aggiornamento del registro sincronizzato sul fronte di salita del clock
```

```
    register_process : PROCESS (clock)
```

```
BEGIN
```

```
    IF (clock'event AND clock = '1') THEN
```

```
        IF (reset = '1') THEN
```

```
            -- Reset sincrono: azzera il contenuto del registro
```

```
            internal_value <= (OTHERS => '0');
```

```
        ELSE
```

```
            IF (enable = '1') THEN
```

```
                -- Caricamento del nuovo valore quando enable è attivo
```

```
                internal_value <= data_in;
```

```
            END IF;
```

```
        END IF;
```

```
    END IF;
```

```
END PROCESS;
```

```
    -- Assegnazione del valore interno all'uscita
```

```
    data_out <= internal_value;
```

```
END behavioural;
```

Contatore Modulo 8

Progetto e architettura

Il Contatore Modulo 8 (cont_mod8) è un circuito sequenziale che:

- Conta da 0 a 7 (3 bit) in modo sincrono al fronte di salita del clock (clk).
- Si incrementa solo se il segnale enable_count è attivo ('1').
- Può essere azzerato tramite un segnale di reset sincrono (rst).

L'uscita counter_value fornisce in tempo reale il valore corrente del contatore.

Implementazione

```
-- Contatore modulo 8 (conta da 0 a 7)
-- Incrementa il valore quando enable_count è attivo
ENTITY cont_mod8 IS
  PORT (
    clk, rst : IN STD_LOGIC;          -- Segnale di clock e reset
    enable_count : IN STD_LOGIC;       -- Abilita il conteggio quando alto
    counter_value : OUT STD_LOGIC_VECTOR(2 DOWNTO 0) -- Valore corrente del contatore (3 bit)
  );
END cont_mod8;
```

```
ARCHITECTURE behavioural OF cont_mod8 IS
  -- Registro interno per memorizzare il valore corrente del contatore
  SIGNAL current_count : STD_LOGIC_VECTOR(2 DOWNTO 0) := (OTHERS => '0');
```

```
BEGIN
  -- Processo di conteggio sincronizzato sul fronte di salita del clock
  counter_process : PROCESS (clk)
    BEGIN
      IF (clk'event AND clk = '1') THEN
        -- Reset asincrono: azzera il contatore
        IF (rst = '1') THEN
```

```

    current_count <= (OTHERS => '0');

    ELSE
        -- Incrementa il contatore solo quando enable_count è attivo
        IF (enable_count = '1') THEN
            current_count <= STD_LOGIC_VECTOR(unsigned(current_count) + 1);
        -- c <= c + "111"; posso fare direttamente questa somma se importo IEEE.std_logic_unsigned.ALL
        -- preferibile non farlo perchè sono package non standard
        END IF;
    END IF;
END IF;

END PROCESS;

-- Assegna il valore interno all'uscita
counter_value <= current_count;

END behavioural;

```

Contatore generico

Progetto e architettura

Il contatore mod-N è stato implementato utilizzando un approccio comportamentale (usando un solo process). Tramite GENERIC è possibile scegliere il modulo del conteggio (N) e, di conseguenza, il numero di bit che conterrà l'uscita (espresso come $\log_2(N)$ bit arrotondato per eccesso al prossimo intero). È inoltre possibile settare il contatore ad un valore iniziale tramite il segnale di SET e caricando il valore nel segnale di ingresso I (anch'esso rappresentato su $\lceil \log_2(N) \rceil$ bit). In uscita, invece, troviamo due segnali: CONT (uscita del contatore) e DIV (uscita divisore, alta quando il contatore raggiunge il massimo).

Implementazione

Di seguito è riportato il codice VHDL del contatore mod-N:

```

entity cont_mod_N is
    generic(N : in integer := 60);
    port(
        I : in STD_LOGIC_VECTOR((integer(ceil(log2(real(N)))))-1 DOWNTO 0); --input
        EN : in STD_LOGIC; -- enable

```

```

SET : in STD_LOGIC; -- segnale per settare l'ingresso
CLK : in STD_LOGIC;
RST : in STD_LOGIC;
CONT : out STD_LOGIC_VECTOR((integer(ceil(log2(real(N)))))-1 downto 0); -- uscita del contatore mod N
DIV : out STD_LOGIC -- uscita divisore (alta quando il contatore raggiunge il massimo)
);

end cont_mod_N;

```

```

architecture Behavioral of cont_mod_N is
begin
process(CLK, RST, SET)
variable c: INTEGER range 0 to N-1 := 1;
begin
if(RST = '1') then
  c := 1;
  CONT <= (others => '0');
  DIV <= '0';
elsif(SET = '1') then
  c := to_integer(unsigned(I));
  CONT <= std_logic_vector(to_unsigned(c, CONT'length));
  if c = N-1 then
    DIV <= '1';
  else
    DIV <= '0';
  end if;
  c := c + 1;
elsif (falling_edge(CLK)) then
  if EN = '1' then
    if(c < N) then
      CONT <= std_logic_vector(to_unsigned(c, CONT'length));
      if c = N-1 then
        DIV <= '1';
      end if;
    end if;
  end if;
end if;
end process;
end Behavioral;

```

```

        else
            DIV <= '0';
        end if;

        c := c + 1;

    else
        DIV <= '0';
        CONT <= (others => '0');

        c := 1;
    end if;
end if;
end if;
end process;

```

end Behavioral;

Memoria

Progetto e architettura

La memoria presenta una struttura simile a quella della ROM ma oltre alla lettura si può effettuare anche la scrittura. Anche per essa è stata utilizzato un GENERIC per impostare la lunghezza degli indirizzi e di conseguenza il numero di locazioni interne alla memoria in modo da renderla più integrabile per più progetti

Implementazione

Il componente presenta i seguenti segnali:

- Un segnale di clock in ingresso per scandire le operazioni
- Un segnale *write* per effettuare la scrittura sulla memoria
- Un segnale *address* per scandire le locazioni della memoria
- Un input *input_val* che corrisponde al valore da scrivere sulla memoria
- Un output *output_val* che corrisponde al valore letto dalla memoria

E' stato utilizzato poi un segnale temporaneo *address_temp* che memorizza l'ultima locazione ove è stato scritto in memoria, utile poi per restituire in output il valore letto in quella locazione

entity memoria is

```

    Generic(
        len_add : positive := 4
    );

```

```

Port(
    CLK_mem : in std_logic;
    write : in std_logic;
    address : in std_logic_vector (len_add-1 downto 0);
    inp_val : in std_logic_vector(3 downto 0);
    out_val : out std_logic_vector(3 downto 0)
);

end memoria;

```

```

architecture Behavioral of memoria is

CONSTANT N : positive := 2**len_add;
signal address_temp : std_logic_vector(len_add-1 downto 0);

type MEMORY_N_4 is array (0 to N-1) of std_logic_vector(len_add-1 downto 0);
signal MEM : MEMORY_N_4;
begin
process (CLK_mem)
begin
    if rising_edge(CLK_mem) then
        if(write = '1') then
            MEM(to_integer(unsigned(address))) <= inp_val;
            address_temp <= address;
        end if;
        out_val <= MEM(to_integer(unsigned(address_temp)));
    end if;
end process;
end Behavioral;

```

Button Debouncer

Progetto e architettura

Il Button Debouncer è un componente che "ripulisce" il segnale in input dal bottone, in modo da presentare in uscita un impulso della durata di un colpo di clock per segnalare l'avvenuta pressione del bottone. Il debouncing dei pulsanti è una tecnica comunemente utilizzata nella progettazione di circuiti digitali per garantire che la singola pressione di un pulsante o di un interruttore produca un unico segnale di uscita. Senza il debouncing, la natura meccanica di questi pulsanti o interruttori può far sì che il segnale di uscita "rimbalzi" rapidamente tra gli stati on e off prima di stabilizzarsi, con conseguenti comportamenti imprevisti.

Implementazione

Per realizzare il Button Debouncer è stato creato un process sensibile al solo segnale di clock, attivo sul suo fronte di salita; all'interno di questo process abbiamo definito le transizioni dell'automa con cui è possibile modellare il componente. Gli stati possibili della macchina sono i seguenti:

- NOT_PRESSED: se il pulsante (BTN) è alto, lo stato del pulsante viene impostato su CHK_PRESSED, indicando che la pressione del pulsante viene verificata per il rimbalzo. Se il pulsante non è alto, lo stato del pulsante rimane NOT_PRESSED;
- CHK_PRESSED: un contatore viene controllato rispetto al conteggio calcolato a partire dalle costanti dichiarate come generics. Se il contatore ha raggiunto questo massimo e il pulsante è ancora alto, si presume che la pressione del pulsante non sia un rimbalzo. Il contatore viene azzerato, e viene generato l'impulso alto ripulito e lo stato del pulsante viene impostato su PRESSED. Se a questo punto il pulsante non è alto, lo stato del pulsante viene riportato a NOT_PRESSED;
- PRESSED: Viene abbassata l'uscita per garantire che il segnale sia alto per un solo impulso di clock. Se il pulsante è basso, lo stato del pulsante viene impostato su CHK_NOT_PRESSED, indicando che il rilascio del pulsante viene controllato per evitare il rimbalzo. Se il pulsante è ancora alto, lo stato del pulsante rimane PRESSED;
- CHK_NOT_PRESSED: simile a CHK_PRESSED, il contatore viene controllato rispetto al conteggio massimo. Se il contatore ha raggiunto il massimo e il pulsante è ancora basso, si presume che il rilascio del pulsante non sia un rimbalzo. Il contatore viene azzerato e lo stato del pulsante viene impostato su NOT_PRESSED. Se il pulsante non è ancora basso a questo punto, lo stato del pulsante viene riportato a PRESSED.

Di seguito è riportato il codice VHDL di una FSM che implementa il meccanismo suddetto.

entity ButtonDebouncer is

```
generic (
    CLK_period: integer := 10; -- periodo del clock (della board) in nanosecondi
    btn_noise_time: integer := 10000000 -- durata stimata dell'oscillazione del bottone in nanosecondi
        -- il valore di default è 10 millisecondi
);
Port ( RST : in STD_LOGIC;
    CLK : in STD_LOGIC;
    BTN : in STD_LOGIC;
```

```
CLEARED_BTN : out STD_LOGIC);  
end ButtonDebouncer;
```

```
architecture Behavioral of ButtonDebouncer is
```

```
type stato is (NOT_PRESSED, CHK_PRESSED, PRESSED, CHK_NOT_PRESSED);
```

```
signal BTN_state : stato := NOT_PRESSED;
```

```
constant max_count : integer := btn_noise_time/CLK_period; -- 10000000/10= conto 1000000 colpi di clock
```

```
begin
```

```
deb: process (CLK)
```

```
variable count: integer := 0;
```

```
begin
```

```
if rising_edge(CLK) then
```

```
if( RST = '1') then
```

```
    BTN_state <= NOT_PRESSED;
```

```
    CLEARED_BTN <= '0';
```

```
else
```

```
    case BTN_state is
```

```
        when NOT_PRESSED =>
```

```
            if( BTN = '1' ) then
```

```
                BTN_state <= CHK_PRESSED;
```

```
            else
```

```
                BTN_state <= NOT_PRESSED;
```

```
            end if;
```

```
        when CHK_PRESSED =>
```

```
            if(count = max_count -1) then
```

```
                if(BTN = '1') then
```

```

count:=0;
CLEARED_BTN <= '1';
BTN_state <= NOT_PRESSED;

else
    count:=0;
    BTN_state <= NOT_PRESSED;
end if;

else
    count:= count+1;
    BTN_state <= CHK_PRESSED;
end if;

when PRESSED =>
    CLEARED_BTN<= '0';

        if(BTN = '0') then
            BTN_state <= CHK_NOT_PRESSED;
        else
            BTN_state <= NOT_PRESSED;
        end if;

when CHK_NOT_PRESSED =>
    if(count = max_count -1) then
        if(BTN = '0') then
            count:=0;
            BTN_state <= NOT_PRESSED;
        else
            count:=0;
            BTN_state <= NOT_PRESSED;
        end if;

```

```

else
    count:= count+1;
    BTN_state <= CHK_NOT_PRESSED;
end if;

when others =>
    BTN_state <= NOT_PRESSED;
    end case;
end if;
end if;
end process;

end Behavioral;

```

Divisore di frequenza

Progetto e architettura

Il divisore di frequenza (da 100MHz ad 1 Hz) è stato realizzato adottando un approccio comportamentale. Esso riceve in ingresso un clock ad una frequenza di 100 MHz (a causa della board) e fornisce in uscita un clock con una frequenza di 1Hz. Quindi, il tempo tra due fronti di salita sarà pari ad 1 secondo. Utilizzando una variabile contatore (counter), incrementata ad ogni fronte di salita del clock, una volta raggiunto il valore 49MHz (ovvero $\text{clock_frequency_in}/\text{clock_frequency_out} - 1$), il valore in uscita viene negato. Ciò genererà in uscita un segnale di clock di 1 Hz con duty cycle del 50%. Quest'ultima affermazione vuol dire che il segnale è alto per metà del suo periodo e basso per la rimanente metà.

Implementazione

Di seguito è riportato il codice VHDL del componente:

```

entity clock_divider is
    generic(
        clock_frequency_in : in integer := 100000000;
        clock_frequency_out : in integer := 1);
    port(
        CLK_IN : in STD_LOGIC;
        CLK_OUT : out STD_LOGIC
    );
end entity;

```

```
 );
end clock_divider;
```

```
architecture Behavioral of clock_divider is
```

```
signal clk_tmp : STD_LOGIC := '0';
signal counter : integer := 0;
constant count_max_value : integer := (clock_frequency_in/clock_frequency_out) - 1;
```

```
begin
```

```
process(CLK_IN)
begin
  if rising_edge(CLK_IN) then
    counter <= counter + 1;
    if(counter = 49999999) then
      clk_tmp <= NOT clk_tmp; --
      counter <= 0; -- resetto il contatore
    end if;
    end if;
    CLK_OUT <= clk_tmp;
  end process;
```

```
end Behavioral;
```