

# Implementazione del gioco del blackjack in VHDL

Emanuele Aina (matr. 129548)

01GSS - Specifica e simulazione dei sistemi digitali

## 1 Specifiche di progetto

Si progettano, a livello RT, un circuito che permetta di giocare al gioco del Blackjack.

### 1.1 Spiegazione del gioco

Il gioco si svolge tra due giocatori, di seguito denominati Giocatore e Banco, con un mazzo di carte da poker. Inizia a giocare il Giocatore, decidendo se estrarre una carta dal mazzo, oppure passare la mano al Banco. Ad ogni estrazione, il valore della carta estratta viene sommato al punteggio corrente: lo scopo è arrivare a totalizzare un punteggio il più vicino possibile a 21, senza mai superarlo. Se tale valore viene superato durante la mano del Giocatore, il Banco vince immediatamente. Viceversa, il Banco comincia ad estrarre carte dal mazzo con lo scopo di realizzare un punteggio almeno pari a quello del Giocatore, senza comunque superare il limite di 21. Se ciò non accade, vince il Giocatore.

### 1.2 Implementazione

Il circuito possiede quattro ingressi di controllo Reset, NewGame, Stop ed En di parallelismo pari ad 1 bit e un ingresso di dato I di parallelismo pari a 3 bit, e si deve comportare nel seguente modo:

- Se  $\text{Reset} = 1$ , il circuito si porta sempre nello stato iniziale, in cui tutti i led e le 4 cifre del display a 7 segmenti sono spenti, indipendentemente dal valore degli altri segnali.
- La prima volta che il segnale NewGame assume il valore 1, il sistema si prepara a giocare una nuova partita: le due cifre più a destra del display a 7 segmenti si accendono, riportando il valore del punteggio iniziale (ovvero 00) del Giocatore, mentre le due cifre più a sinistra del display e i led sono spenti.
- Dopo aver iniziato una partita, ogni volta che En assume il valore 1, i segnali dell'ingresso di dato I vengono campionati e memorizzati. Essi rappresentano il valore, in binario, della carta estratta dal mazzo (si utilizzano carte con valori

limitati tra 1 e 8, dove il valore 8 è codificato su tre bit come 000). Tale valore deve essere visualizzato sui led, ed essere sommato al punteggio corrente.

- Nel momento in cui il segnale Stop si porta a 1, la mano passa al Banco: le due cifre più a sinistra del display a 7 segmenti si accendono e riportano il valore iniziale del punteggio del Banco (di nuovo, 00). Analogamente al passo precedente, ogni volta che il segnale assume il valore 1, una carta viene estratta dal mazzo, il suo valore viene visualizzato sui led e sommato al punteggio del Banco.
- La partita termina quando:
  - Il Banco ha totalizzato un punteggio superiore a quello del Giocatore, senza eccedere il limite di 21. In questo caso, i punti decimali relativi alle due cifre del punteggio del Banco si accendono.
  - Il Banco ha totalizzato un punteggio superiore al limite di 21: in questo caso, si illuminano i punti decimali relativi alle due cifre che riportano il punteggio del Giocatore.
  - Durante la sua mano, il Giocatore ha ottenuto un punteggio superiore a 21: come nel primo caso, si illuminano i punti decimali delle due cifre del punteggio del Banco. Si noti che in questo caso il Banco non gioca affatto, quindi i punti decimali si devono accendere, ma le due cifre del display a 7 segmenti devono rimanere spente.
- A questo punto, il sistema si ferma e aspetta che il segnale NewGame si riporti a 1, in modo da iniziare una nuova partita.

Si implementi il circuito richiesto sulla FPGA in dotazione, utilizzando i bottoni BTN3, BTN2, BTN1 e BTN0 per i segnali di Reset, NewGame, Stop ed En, e gli switch SW7, SW6 e SW5 per il segnale di dato I. Il circuito lavori ad una frequenza di clock pari a 50 kHz.

Si osservi che:

- Il numero visualizzato sul display deve essere codificato in base 10 (quindi, se il suo valore è 19 e viene incrementato, il nuovo valore da visualizzare è effettivamente 20, non 1A).
- I led da illuminare ogni volta che una carta viene estratta sono in numero pari al valore della carta, e partono sempre da LD0. Per esempio, se la carta estratta ha un valore pari a 3, si devono illuminare i led LD0, LD1 e LD2.

## 2 Implementazione

### 2.1 Metodologia

Il progetto è stato implementato esclusivamente mediante descrizioni VHDL, sia a livello comportamentale che a livello strutturale.

Lo sviluppo è avvenuto in ambiente Linux (Debian etch) utilizzando GHDL 0.25 per le simulazioni e, successivamente, in ambiente Windows con Xilinx 6.1i per la programmazione su FPGA.

Per ogni componente sviluppato è stato inoltre realizzato un testbench in VHDL puro, in modo da renderne possibile la verifica in modo automatico (make run).

## 2.2 Board

Il componente board rappresenta l'entità di più alto livello e costituisce la parte del progetto che verrà programmata sulla FPGA.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity board is
  generic (GAME_CLK_DIV: integer := 1000;
          DISP_CLK_DIV: integer := 100000);
  port (CLK:                in  std_logic;
        Reset, NewGame, Stop, En: in  std_logic;
        DATA_IN:          in  std_logic_vector (2 downto 0);
        OUTPUT:           out std_logic_vector (7 downto 0);
        AN:                out std_logic_vector (3 downto 0));
end board;

architecture structural of board is
  component blackjack
    port (CLK:                in  std_logic;
          Reset, NewGame, Stop, En: in  std_logic;
          DATA_IN:          in  std_logic_vector (7 downto 0);
          PLAYER_L, PLAYER_H: out std_logic_vector (3 downto 0);
          DEALER_L, DEALER_H: out std_logic_vector (3 downto 0);
          PLAYER_SHOW, DEALER_SHOW: out std_logic;
          PLAYER_WIN, DEALER_WIN: out std_logic);
  end component;
  component display
    port (CLK, RST:          in  std_logic;
          PLAYER_L, PLAYER_H: in  std_logic_vector (3 downto 0);
          DEALER_L, DEALER_H: in  std_logic_vector (3 downto 0);
          PLAYER_SHOW, DEALER_SHOW: in  std_logic;
          PLAYER_WIN, DEALER_WIN: in  std_logic;
          OUTPUT:          out std_logic_vector (7 downto 0);
          AN:              out std_logic_vector (3 downto 0));
  end component;
end architecture;
```

```

component pulse_generator
    port (CLK: in std_logic;
          I: in std_logic;
          O: out std_logic);
end component;
component input_encoder
    port (I: in std_logic_vector(2 downto 0);
          O: out std_logic_vector(3 downto 0));
end component;
component clock_divider
    generic (MODULUS: in positive range 2 to integer'high := 4);
    port (CLK, RST: in std_logic;
          O: out std_logic);
end component;

signal VALUE_IN: std_logic_vector (7 downto 0);
signal Reset_PULSE, Reset_BJ_PULSE, Reset_DISP_PULSE: std_logic;
signal NewGame_PULSE, Stop_PULSE, En_PULSE: std_logic;
signal PLAYER_L, PLAYER_H: std_logic_vector (3 downto 0);
signal DEALER_L, DEALER_H: std_logic_vector (3 downto 0);
signal PLAYER_SHOW, DEALER_SHOW: std_logic;
signal PLAYER_WIN, DEALER_WIN: std_logic;
signal NRESET, NRESET_BJ, NRESET_DISP: std_logic;
signal BJ_CLK, DISP_CLK: std_logic;
begin
    NRESET <= not Reset_PULSE;
    NRESET_BJ <= not Reset_BJ_PULSE;
    NRESET_DISP <= not Reset_DISP_PULSE;

    VALUE_IN(7 downto 4) <= "0000";

    in_enc: input_encoder
        port map (DATA_IN, VALUE_IN(3 downto 0));

    r_pgen: pulse_generator
        port map (CLK, Reset, Reset_PULSE);
    r_bj_pgen: pulse_generator
        port map (BJ_CLK, Reset, Reset_BJ_PULSE);
    r_disp_pgen: pulse_generator
        port map (DISP_CLK, Reset, Reset_DISP_PULSE);

    n_pgen: pulse_generator port map (BJ_CLK, NewGame, NewGame_PULSE);
    s_pgen: pulse_generator port map (BJ_CLK, Stop, Stop_PULSE);

```

```

e_pgen: pulse_generator port map (BJ_CLK, En, En_PULSE);

bj_div: clock_divider
    generic map (GAME_CLK_DIV)
    port map (CLK, NRESET, BJ_CLK);

disp_div: clock_divider
    generic map (DISP_CLK_DIV)
    port map (CLK, NRESET, DISP_CLK);

bj: blackjack
    port map (BJ_CLK,
             Reset_BJ_PULSE, NewGame_PULSE, Stop_PULSE, En_PULSE,
             VALUE_IN,
             PLAYER_L, PLAYER_H, DEALER_L, DEALER_H,
             PLAYER_SHOW, DEALER_SHOW,
             PLAYER_WIN, DEALER_WIN);

disp: display
    port map (DISP_CLK, NRESET_DISP,
             PLAYER_L, PLAYER_H, DEALER_L, DEALER_H,
             PLAYER_SHOW, DEALER_SHOW,
             PLAYER_WIN, DEALER_WIN,
             OUTPUT, AN);
end structural;

```

La descrizione è strutturale e provvede a istanziare i componenti `blackjack` e `display` connettendoli tra loro e a impiegare `clock_divider` per convertire il clock della scheda a frequenze inferiori per la logica di gioco e per la logica di gestione del display. I segnali di input vengono inoltre filtrati usando dei generatori di impulsi al fine di evitare campionamenti multipli di una singola pressione. Al fine di considerare l'ingresso 000 la codifica del valore 8, l'ingresso dati passa attraverso il componente `input_encoder`.

Il componente è dotato del seguente testbench per la verifica automatica.

```

library std;
library ieee;

use std.textio.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all; -- synopsys only
use ieee.std_logic_arith.all; -- synopsys only

entity tb_board is
end tb_board;

```

```

architecture test of tb_board is
    signal CLK: std_logic := '0';
    signal Reset, NewGame, Stop, En: std_logic;
    signal DATA_IN: std_logic_vector (2 downto 0);

    type character_vector is array(natural range <>) of character;
    type boolean_vector is array(natural range <>) of boolean;

    signal DIGIT: std_logic_vector (7 downto 0);
    signal AN:    std_logic_vector (3 downto 0);

    signal DIGITS: character_vector (3 downto 0);
    signal DOTS:   boolean_vector (3 downto 0);

    signal clock_counter: integer := -1;
    signal finished: boolean := false;

    component board
        generic (GAME_CLK_DIV: integer := 1000;
                DISP_CLK_DIV: integer := 1000000);
        port (CLK:                in std_logic;
              Reset, NewGame, Stop, En: in std_logic;
              DATA_IN: in std_logic_vector (2 downto 0);
              OUTPUT: out std_logic_vector (7 downto 0);
              AN: out std_logic_vector (3 downto 0));
    end component;

    component display_simulator
        port(E14, G13, N15, P15, R16, F13, N16: in std_logic;
              P16: in std_logic;
              E13, F14, G14, D14: in std_logic;

              DIGITS_0, DIGITS_1, DIGITS_2, DIGITS_3: out character;
              DOTS_0, DOTS_1, DOTS_2, DOTS_3: out boolean);
    end component;

begin
    U: board
        generic map (2, 2)
        port map (CLK, Reset, NewGame, Stop, En, DATA_IN, DIGIT, AN);

    simul: display_simulator

```

```

        port map (DIGIT(7), DIGIT(6), DIGIT(5), DIGIT(4),
                  DIGIT(3), DIGIT(2), DIGIT(1),
                  DIGIT(0),
                  AN(3), AN(2), AN(1), AN(0),
                  DIGITS(3), DIGITS(2), DIGITS(1), DIGITS(0),
                  DOTS(3), DOTS(2), DOTS(1), DOTS(0));

clock: process
begin
    CLK <= not CLK;
    if finished then wait; end if;
    wait for 0.5 ns;
end process;

count: process(CLK)
begin
    if rising_edge(CLK) then
        clock_counter <= clock_counter + 1;
    end if;
end process;

test: process
    variable testReset, testNewGame, testStop, testEn: std_logic;
    variable testDATA_IN: integer;

    variable testDIGITS: character_vector (3 downto 0);
    variable testDOTS: boolean_vector (3 downto 0);

    file test_file: text is in "board/tb_board.test";

    variable l: line;
    variable t: integer;
    variable good: boolean;
    variable space: character;

    variable enable_debug: boolean := false;
    variable l_out: line;
begin
    wait on clock_counter;

    while not endfile(test_file) loop
        readline(test_file, l);

```

```

-- read the time from the beginning of the line
-- skip the line if it doesn't start with an integer
read(l, t, good => good);
next when not good;

read(l, space);

read(l, testReset);
read(l, testNewGame);
read(l, testStop);
read(l, testEn);
read(l, space);

read(l, testDATA_IN);
read(l, space);

read(l, testDIGITS(3));
read(l, testDIGITS(2));
read(l, testDIGITS(1));
read(l, testDIGITS(0));
read(l, space);

read(l, testDOTS(3));
read(l, testDOTS(2));
read(l, testDOTS(1));
read(l, testDOTS(0));

Reset    <= testReset;
NewGame  <= testNewGame;
Stop     <= testStop;
En       <= testEn;
DATA_IN  <= conv_std_logic_vector(testDATA_IN, DATA_IN'length);

while clock_counter /= t*1000 loop
    wait on clock_counter;
end loop;

if enable_debug then
    write(l_out, string'("(tb_board: ")");
    write(l_out, now);
    write(l_out, string'(") ")");
    write(l_out, string'("DIGITS = ")");
    write(l_out, DIGITS(3));

```

```

        write(l_out, string'(" "));
        write(l_out, DIGITS(2));
        write(l_out, string'(" "));
        write(l_out, DIGITS(1));
        write(l_out, string'(" "));
        write(l_out, DIGITS(0));
        write(l_out, string'(", DOTS = "));
        write(l_out, DOTS(3));
        write(l_out, string'(" "));
        write(l_out, DOTS(2));
        write(l_out, string'(" "));
        write(l_out, DOTS(1));
        write(l_out, string'(" "));
        write(l_out, DOTS(0));
        write(l_out, string'(", AN = "));
        write(l_out, AN);
        writeline(output, l_out);
    end if;
    assert DIGITS(3) = testDIGITS(3)
        report "Mismatch on output DIGITS(3)";
    assert DIGITS(2) = testDIGITS(2)
        report "Mismatch on output DIGITS(2)";
    assert DIGITS(1) = testDIGITS(1)
        report "Mismatch on output DIGITS(1)";
    assert DIGITS(0) = testDIGITS(0)
        report "Mismatch on output DIGITS(0)";
    assert DOTS(3) = testDOTS(3)
        report "Mismatch on output DOTS(3)";
    assert DOTS(2) = testDOTS(2)
        report "Mismatch on output DOTS(2)";
    assert DOTS(1) = testDOTS(1)
        report "Mismatch on output DOTS(1)";
    assert DOTS(0) = testDOTS(0)
        report "Mismatch on output DOTS(0)";
end loop;

finished <= true;
wait;
end process;

end test;

```

Oltre all'unità da verificare, viene anche istanziato un simulatore per il sistema di gestione dei display a sette segmenti disponibili sulla FPGA, controllando il funzio-

amento della logica di codifica e multiplexing.

Il clock viene generato con un periodo di 1ns fino a che non esistano più eventi per la simulazione. A tal punto anche il clock viene fermato in modo da segnalare al simulatore la fine del test.

Il processo `count` si limita a conteggiare il numero di colpi di clock in modo che il processo `test` possa impostare i vettori di test con la cadenza specificata. Questo processo, infatti, si occupa di leggere i vettori di input e gli output attesi da un file e applicarli all'unità testata. Nel caso specifico, dovendo simulare l'input dell'utente, la frequenza con cui gli input vengono applicati è nell'ordine dei millisecondi anziché dei nanosecondi usati dal clock di simulazione.

```
-- time Reset NewGame Stop En DATA_IN DIGITS DOTS
1 1000 0      False False False False
2 0100 0 00   False False False False
3 0001 7 07   False False False False
5 0000 7 07   False False False False
6 0001 0 15   False False False False
7 0000 0 15   False False False False
8 0001 7 22   False False True  True
9 0000 0 22   False False True  True
10 0100 0 00  False False False False
11 0001 3 03  False False False False
12 0010 0 0300 False False False False
13 0001 1 0301 False False False False
14 0000 1 0301 False False False False
15 0001 0 0309 False False True  True
20 1000 0      False False False False
30 0100 0 00   False False False False
31 0010 0 0000 False False False False
```

Su ogni riga è specificato l'istante corrispondente, gli input e gli output.

## 2.3 Blackjack

Il componente `blackjack` contiene l'intera logica di gioco descritta nelle specifiche, utilizzando al suo interno i componenti `game_logic`, il quale implementa le regole di vittoria/perdita, `Fsm`, che determina gli stati di gioco, `bcd_encoder` per convertire i punteggi in Binary Coded Decimal, nonché `reg` e `rca`, rispettivamente un registro PIPO e un Ripple Carry Adder.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all; -- synopsys only

entity blackjack is
```

```

port (CLK:                in  std_logic;
      Reset, NewGame, Stop, En: in  std_logic;
      DATA_IN:           in  std_logic_vector (7 downto 0);
      PLAYER_L, PLAYER_H: out  std_logic_vector (3 downto 0);
      DEALER_L, DEALER_H: out  std_logic_vector (3 downto 0);
      PLAYER_SHOW, DEALER_SHOW: out std_logic;
      PLAYER_WIN, DEALER_WIN: out  std_logic);
end blackjack;

architecture structural of blackjack is
  component game_logic
    port (PLAYER, DEALER: in  std_logic_vector (7 downto 0);
          WIN, BUST:      out std_logic);
  end component;
  component reg is
    generic (N: integer := 8);
    port (CLK, RST: in  std_logic;
          EN:      in  std_logic;
          A:      in  std_logic_vector (N-1 downto 0);
          O:      out std_logic_vector (N-1 downto 0));
  end component;
  component rca
    generic (N: integer := 8);
    port (A, B: in  std_logic_vector (N-1 downto 0);
          Ci:  in  std_logic;
          S:   out std_logic_vector (N-1 downto 0);
          Co:  out std_logic);
  end component;
  component fsm
    port (CLK, RST: in  std_logic;
          NewGame, En, Stop, Bust, Win: in std_logic;
          ShowPlayer, ShowDealer: out std_logic;
          PlayerWin, DealerWin: out std_logic;
          PlayerRead, DealerRead: out std_logic;
          Clear: out std_logic);
  end component;
  component bcd_encoder
    port (I: in  std_logic_vector(7 downto 0);
          H, L: out std_logic_vector(3 downto 0));
  end component;

  signal Bust, Win: std_logic;
  signal ShowPlayer, ShowDealer,

```

```

        PlayerWin, DealerWin,
        PlayerRead, DealerRead,
        Clear: std_logic;
    signal NRESET, NPLAYER_EN, NDEALER_EN: std_logic;
    signal PLAYER, DEALER,
        PLAYER_NEXTSCORE,
        DEALER_NEXTSCORE: std_logic_vector (7 downto 0);
begin
    NPLAYER_EN <= not PlayerRead;
    NDEALER_EN <= not DealerRead;
    NRESET      <= not Clear;

    PLAYER_SHOW <= ShowPlayer;
    DEALER_SHOW <= ShowDealer;
    PLAYER_WIN   <= PlayerWin;
    DEALER_WIN   <= DealerWin;

    p_score: reg
        generic map(8)
        port map (CLK, NRESET, NPLAYER_EN, PLAYER_NEXTSCORE, PLAYER);
    player_adder: rca
        generic map(8)
        port map (PLAYER, DATA_IN, '0', PLAYER_NEXTSCORE);

    d_score: reg
        generic map(8)
        port map (CLK, NRESET, NDEALER_EN, DEALER_NEXTSCORE, DEALER);

    dealer_adder: rca
        generic map(8)
        port map (DEALER, DATA_IN, '0', DEALER_NEXTSCORE);

    game: game_logic
        port map (PLAYER, DEALER, Win, Bust);

    state_machine: fsm
        port map (CLK, Reset, NewGame, En, Stop, Bust, Win,
            ShowPlayer, ShowDealer, PlayerWin, DealerWin,
            PlayerRead, DealerRead, Clear);

    bcd_player: bcd_encoder
        port map (PLAYER, PLAYER_H, PLAYER_L);

```

```

bcd_dealer: bcd_encoder
    port map (DEALER, DEALER_H, DEALER_L);
end structural;

```

Anche qui la descrizione è strutturale e si occupa di istanziare e interconnettere i componenti presenti. In particolare i due registri sono usati per memorizzare il punteggio del Giocatore e del Mazziere: a ciascuno è connesso un sommatore il quale somma il punteggio con il valore in input ed è usato per calcolare il punteggio futuro. `game_logic` determina se uno tra Giocatore o Mazziere ha vinto o ha sballato al fine di segnalarlo alla macchina a stati, la quale reagirà opportunamente. Entrambi i punteggi vengono poi convertiti in BCD a due cifre.

```

-- time Reset NewGame Stop En DATA_IN
--   PLAYER DEALER PLAYER_SHOW DEALER_SHOW PLAYER_WIN DEALER_WIN
1 1110 0 0 0 00 00 -- Reset
2 0100 0 0 0 00 00 -- NewGame
3 0100 0 0 0 00 00 -- The fsm is in CLEAN
4 0001 4 0 0 00 00 -- Put the first player card
5 0000 4 0 0 10 00 -- Setup signals
6 0000 4 0 0 10 00 -- Read it
7 0001 4 4 0 10 00 -- Check it
8 0001 1 4 0 10 00 -- Put the second player card
9 0000 1 4 0 10 00
10 0001 1 4 0 10 00
11 0001 1 5 0 10 00
12 0010 2 5 0 10 00 -- Stop the player
13 0001 3 5 0 10 00 -- Put the first dealer card
14 0000 2 5 0 11 00 -- Setup signals
15 0000 2 5 0 11 00 -- Read it
16 0000 3 5 2 11 00 -- Check it
17 0001 20 5 2 11 00
18 0001 20 5 2 11 00
19 0001 20 5 2 11 00
20 0000 0 5 22 11 00 -- Check dealer busted
21 0001 2 5 22 11 00 -- Pull up PLAYER_WIN
22 0000 2 5 22 11 10 -- The new card is ignored
23 0100 0 5 22 11 10
24 0000 0 5 22 11 10 -- CLEAN
25 0100 0 0 0 00 00
26 0001 23 0 0 10 00
27 0001 23 0 0 10 00
28 0000 23 0 0 10 00
29 0000 0 23 0 10 00 -- Check player busted
30 0001 2 23 0 10 00 -- Pull up DEALER_WIN

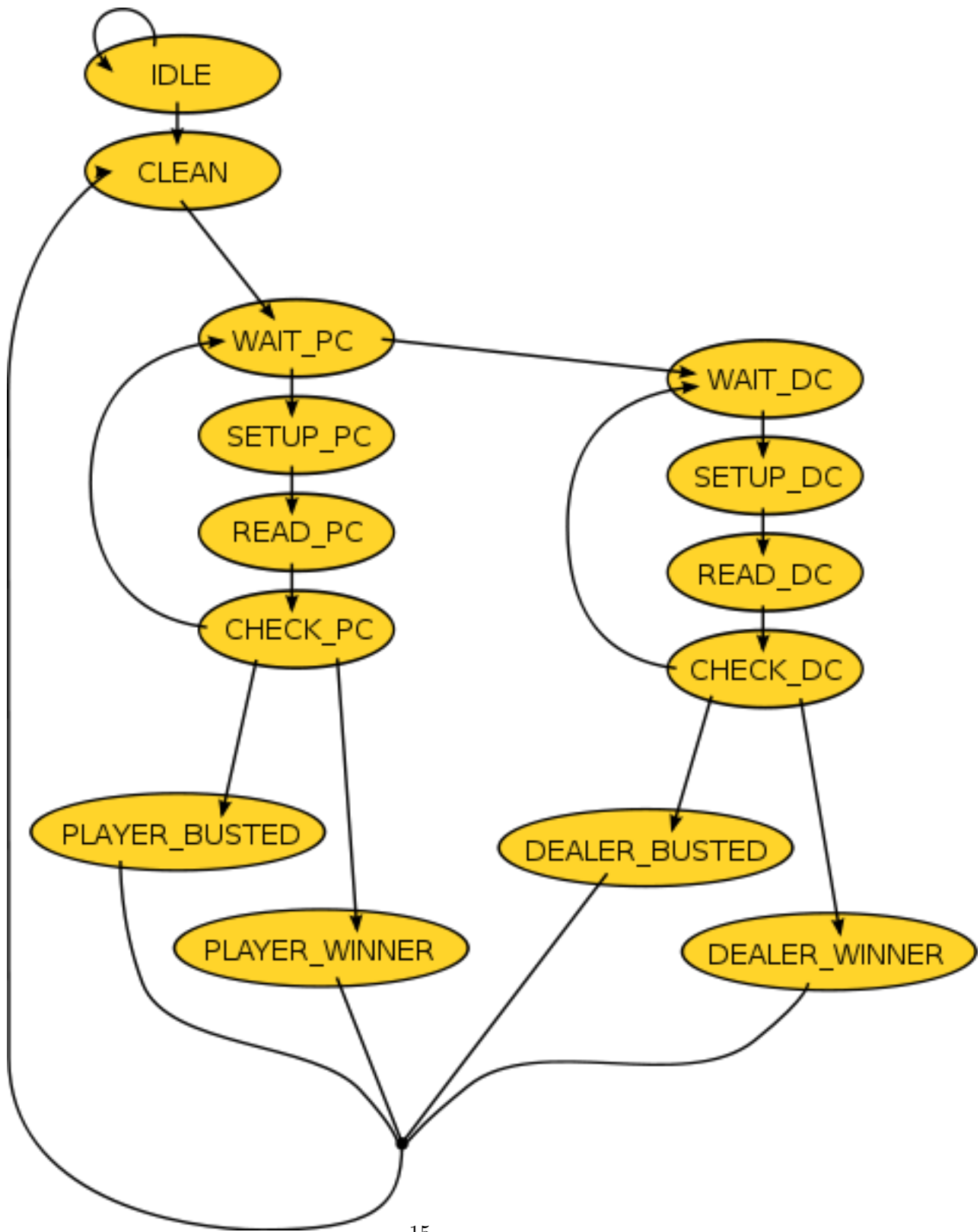
```

```
31 0000 2 23 0 10 01
32 0000 2 23 0 10 01
```

I vettori di test simulano una serie di partite controllando il comportamento del sistema in alcune situazioni plausibili. In questo caso a ogni riga corrisponde un colpo di clock, per cui è necessario consentire alla macchina a stati il tempo necessario a compiere le proprie funzioni, essendo l'unico componente a non reagire nell'arco di un periodo di clock. Il codice VHDL del testbench è stato ommesso essendo analogo a quanto visto nel caso del componente `board`.

## 2.4 Fsm

La macchina a stati si occupa di gestire le differenti fasi del gioco, nonché di controllare l'acquisizione e la verifica dei valori in input e di prendere le opportune decisioni. Questo è l'unico componente realizzato il cui modo di operare richiede più colpi di clock: questa è una scelta voluta, in modo da concentrare la logica per la gestione delle tempistiche e degli stati in un singolo componente.



Lo stato iniziale è lo stato di IDLE, nel quale la macchina resta in attesa di eventi. Da qui CLEAN imposta le condizioni iniziali del gioco, in modo da permettere il funzionamento degli stati successivi. Tra questi, quelli i cui nomi terminano con PC sono relativi al turno del Giocatore, mentre quelli con DC spettano al Mazziere. Questi rappresentano le fasi di attesa, acquisizione e controllo dei valori in input, mentre quelli che terminano con BUSTED o WINNER indicano le condizioni terminali del gioco.

```

library ieee;
use ieee.std_logic_1164.all;

entity fsm is
    port (CLK, RST: in std_logic;
          NewGame, En, Stop, Bust, Win: in std_logic;
          ShowPlayer, ShowDealer, PlayerWin, DealerWin: out std_logic;
          PlayerRead, DealerRead: out std_logic;
          Clear: out std_logic);
end fsm;

architecture state_machine of fsm is
    type STATE is (IDLE, CLEAN,
                  WAIT_PC, WAIT_DC,
                  SETUP_PC, SETUP_DC,
                  READ_PC, READ_DC,
                  CHECK_PC, CHECK_DC,
                  PLAYER_BUSTED, PLAYER_WINNER,
                  DEALER_BUSTED, DEALER_WINNER);
    signal current_state, next_state: STATE;
begin
    process (CLK, RST)
    begin
        if RST='1' then
            current_state <= IDLE;
        elsif rising_edge(CLK) then
            current_state <= next_state;
        end if;
    end process;

    process (current_state, NewGame, Stop, En, Bust, Win)
    begin
        case current_state is
            when IDLE =>
                if NewGame = '1' then
                    next_state <= CLEAN;
                else

```

```

        next_state <= IDLE;
    end if;
when CLEAN =>
    next_state <= WAIT_PC;
when WAIT_PC =>
    if Stop = '1' then
        next_state <= WAIT_DC;
    elsif En = '1' then
        next_state <= SETUP_PC;
    end if;
when SETUP_PC =>
    next_state <= READ_PC;
when READ_PC =>
    next_state <= CHECK_PC;
when CHECK_PC =>
    if Bust = '1' then
        next_state <= PLAYER_BUSTED;
    elsif Win = '1' then
        next_state <= PLAYER_WINNER;
    else
        next_state <= WAIT_PC;
    end if;
when WAIT_DC =>
    if En = '1' then
        next_state <= SETUP_DC;
    end if;
when SETUP_DC =>
    next_state <= READ_DC;
when READ_DC =>
    next_state <= CHECK_DC;
when CHECK_DC =>
    if Bust = '1' then
        next_state <= DEALER_BUSTED;
    elsif Win = '1' then
        next_state <= DEALER_WINNER;
    else
        next_state <= WAIT_DC;
    end if;
when PLAYER_BUSTED =>
    if NewGame = '1' then
        next_state <= CLEAN;
    end if;
when PLAYER_WINNER =>

```

```

        if NewGame = '1' then
            next_state <= CLEAN;
        end if;
    when DEALER_BUSTED =>
        if NewGame = '1' then
            next_state <= CLEAN;
        end if;
    when DEALER_WINNER =>
        if NewGame = '1' then
            next_state <= CLEAN;
        end if;
    end case;
end process;

process (CLK)
begin
    if rising_edge(CLK) then
        case current_state is
            when IDLE =>
                Clear      <= '0';
                ShowPlayer <= '0';
                ShowDealer <= '0';
                PlayerWin  <= '0';
                DealerWin  <= '0';
                PlayerRead <= '0';
                DealerRead <= '0';
            when CLEAN =>
                Clear      <= '1';
                ShowPlayer <= '0';
                ShowDealer <= '0';
                PlayerWin  <= '0';
                DealerWin  <= '0';
                PlayerRead <= '0';
                DealerRead <= '0';
            when WAIT_PC =>
                Clear      <= '0';
                ShowPlayer <= '1';
            when SETUP_PC =>
                PlayerRead <= '1';
            when READ_PC =>
                PlayerRead <= '0';
            when CHECK_PC =>
            when WAIT_DC =>

```

```

        ShowDealer <= '1';
    when SETUP_DC =>
        DealerRead <= '1';
    when READ_DC =>
        DealerRead <= '0';
    when CHECK_DC =>
    when PLAYER_BUSTED =>
        DealerWin <= '1';
    when PLAYER_WINNER =>
        PlayerWin <= '1';
    when DEALER_BUSTED =>
        PlayerWin <= '1';
    when DEALER_WINNER =>
        DealerWin <= '1';
    end case;
    end if;
end process;
end state_machine;

```

La macchina a stati è stata costruita con tre processi: uno molto semplice per propagare lo stato futuro, uno per determinare quale questo sia in funzione degli input e uno per determinare gli output in funzione dello stato corrente.

```

-- time RST NewGame En Stop Bust Win ShowPlayer ShowDealer
--   PlayerWin DealerWin PlayerRead DealerRead Clear
1 1 00110 000000 0
2 0 10000 000000 0 -- IDLE
3 0 00000 000000 1 -- CLEAN
4 0 01000 100000 0 -- WAIT_PC
5 0 00000 100010 0 -- SETUP_PC
6 0 00000 100000 0 -- READ_PC
7 0 00000 100000 0 -- CHECK_PC
8 0 00100 100000 0 -- WAIT_PC
9 0 01000 110000 0 -- WAIT_DC
10 0 00000 110001 0 -- SETUP_DC
11 0 00000 110000 0 -- READ_DC
12 0 00010 110000 0 -- CHECK_DC
13 0 10000 111000 0 -- DEALER_BUSTED
14 0 00000 000000 1 -- CLEAN
15 0 01000 100000 0 -- WAIT_PC [...]
18 0 00001 100000 0 -- CHECK_PC
19 0 00000 101000 0 -- PLAYER_WINNER

```

## 2.5 game\_logic

La logica che determina la vittoria o la sconfitta dei giocatori è contenuta nel componente combinatorio `game_logic`, il quale si occupa di controllare i punteggi nel caso questi superino 21 o che si sia verificata una delle condizioni di vittoria.

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

entity game_logic is
    port (PLAYER, DEALER:          in std_logic_vector (7 downto 0);
          WIN, BUST: out std_logic);
end game_logic;

architecture behavioral of game_logic is

winner: process (PLAYER, DEALER)
    variable P, D: unsigned(7 downto 0);
begin
    P := to_01(unsigned(PLAYER));
    D := to_01(unsigned(DEALER));

    if (P = 0) and (D = 0) then
        WIN <= '0';
    elsif (P = 21) or (P <= D) then
        WIN <= '1';
    else
        WIN <= '0';
    end if;
end process;

busted: process (PLAYER, DEALER)
    variable P, D: unsigned(7 downto 0);
begin
    P := to_01(unsigned(PLAYER));
    D := to_01(unsigned(DEALER));

    if (P > 21) or (D > 21) then
        BUST <= '1';
    else
```

```

        BUST <= '0';
    end if;
end process;

```

```

end behavioral;

```

La descrizione è comportamentale e prevede un processo per ciascun output, utilizzando il tipo `unsigned` per effettuare i confronti tra i punteggi.

```

library std;
library ieee;

```

```

use std.textio.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all; -- synopsys only

```

```

entity tb_game_logic is
end tb_game_logic;

```

```

architecture test of tb_game_logic is

```

```

    signal PLAYER, DEALER: std_logic_vector (7 downto 0) := "00000000";
    signal WIN, BUST:      std_logic;

```

```

    component game_logic
        port (PLAYER, DEALER: in  std_logic_vector (7 downto 0);
              WIN, BUST:      out std_logic);
    end component;

```

```

begin
    U: game_logic port map (PLAYER, DEALER, WIN, BUST);

```

```

test: process

```

```

    variable testPLAYER, testDEALER: std_logic_vector (7 downto 0);
    variable testWIN, testBUST:      std_logic;
    file test_file: text is in "game_logic/tb_game_logic.test";

```

```

    variable l: line;
    variable t: time;
    variable i: integer;
    variable good: boolean;
    variable space: character;

```

```

begin

```

```

    while not endfile(test_file) loop
        readline(test_file, l);

```

```

        -- read the time from the beginning of the line

```

```

-- skip the line if it doesn't start with an integer
read(l, i, good => good);
next when not good;

read(l, space);

read(l, testPLAYER);
read(l, space);
read(l, testDEALER);
read(l, space);

read(l, testWIN);
read(l, space);
read(l, testBUST);

PLAYER <= testPLAYER;
DEALER <= testDEALER;

t := i * 1 ns; -- convert an integer to time
if (now < t) then
    wait for t - now;
end if;

assert WIN = testWIN report "Mismatch on output WIN";
assert BUST = testBUST report "Mismatch on output BUST";
end loop;

wait;
end process;

end test;

```

Essendo il componente combinatorio, il testbench risulta essere leggermente più semplice rispetto a quelli impiegati per i componenti precedenti, seppur rimanendo molto simile.

```

-- time PLAYER DEALER WIN BUST
1 00000010 00000100 1 0
2 00001110 00001000 0 0
3 00010101 00010101 1 0
4 00011000 00000100 0 1
5 00001010 00010110 1 1
6 00000000 00000000 0 0
7 00011000 00011001 1 1

```

```
8 00010101 00000000 1 0
9 00000000 00000001 1 0
```

Il test si limita a sottoporre al circuito alcune situazioni di gioco senza un ordine particolare, valutando l'esito di ciascuna.

## 2.6 reg

L'entità `reg` descrive un semplice registro PIPO sincrono dotato di segnale di abilitazione (EN) oltre agli ingressi (A) e alle uscite (O). Il parametro N permette di impostare la dimensione in bit del registro.

```
library ieee;
use ieee.std_logic_1164.all;

entity reg is
    generic (N: integer := 8);

    port (CLK, RST: in std_logic;
          EN:      in std_logic;
          A:       in std_logic_vector (N-1 downto 0);
          O:       out std_logic_vector (N-1 downto 0));
end reg;

architecture behavioral of reg is
begin
    process (RST, CLK)
    begin
        if RST = '0' then
            O <= (O'range => '0');
        elsif CLK'event and CLK = '1' then
            if EN = '0' then
                O <= A;
            end if;
        end if;
    end process;
end behavioral;

architecture structural of reg is
    component fda is
        port (CLK, RST: in std_logic;
              EN:      in std_logic;
              D:       in std_logic;
              Q:       out std_logic);
    end component;
```

```

begin
  fd_vect: for i in N-1 downto 0 generate
    fd_i: fda port map (CLK, RST, EN, A(i), O(i));
  end generate;
end structural;

```

La stessa entità è stata implementata sia con una descrizione comportamentale che con una strutturale. La prima contiene il costrutto condizionale comunemente usato per implementare circuiti sincroni con segnale di reset asincrono e un semplice assegnamento, mentre la seconda istanzia in un ciclo un vettore di flip-flop di tipo D a cui connette ingressi e uscite.

```

-- time RST EN A O
1 01 011 000
2 00 011 000
3 11 011 000
4 10 101 000
5 10 010 101
6 10 111 010
7 10 000 111
8 10 101 000
9 11 010 101
10 11 111 101
11 11 000 101
12 01 111 101
13 10 111 000

```

Il test viene eseguito con un registro a tre bit verificandone il comportamento.

## 2.7 rca

Il sommatore utilizzato è un comune Ripple Carry Adder con dimensione determinata dal parametro N.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rca is
  generic (N: integer := 8);
  port (A, B: in std_logic_vector (N-1 downto 0);
        Ci: in std_logic;
        S: out std_logic_vector (N-1 downto 0);
        Co: out std_logic);
end rca;

```

```

architecture structural of rca is
    signal carry_propagate: std_logic_vector (N downto 0);
    component fa is
        port (A, B: in std_logic;
              Ci: in std_logic;
              S: out std_logic;
              Co: out std_logic);
    end component;
begin
    carry_propagate(0) <= Ci;

    fa_array: for i in 0 to N-1 generate
        fa_i: fa port map (A(i), B(i), carry_propagate(i),
                           S(i), carry_propagate(i+1));
    end generate;

    Co <= carry_propagate(N);
end structural;

```

La descrizione è ovviamente strutturale e provvede a istanziare un vettore di full-adder definendo anche le connessioni necessarie per la propagazione del riporto.

```

-- time A B Ci S Co
1 000 101 1 110 0
2 001 101 0 110 0
3 010 110 1 001 1
4 011 100 0 111 0
5 100 001 1 110 0
6 101 010 1 000 1
7 110 000 1 111 0
8 111 000 1 000 1
9 000 111 1 000 1
10 001 101 1 111 0
11 010 111 1 010 1
12 111 101 1 101 1
13 111 111 1 111 1

```

Ad ogni passo di test viene verificata la corrispondenza tra i valori degli input e del riporto in ingresso con quelli attesi del valore in uscita e del riporto in uscita.

## 2.8 input\_encoder

Questo semplice componente provvede a convertire il segnale in ingresso “000” in “1000”, lasciando invece invariate tutte le altre combinazioni.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity input_encoder is
    port (I: in std_logic_vector(2 downto 0);
          O: out std_logic_vector(3 downto 0));
end input_encoder;

architecture behavioral of input_encoder is
begin
process(I)
begin
    if I = "000" then
        O <= "1000";
    else
        O <= '0' & I;
    end if;
end process;
end behavioral;

```

## 2.9 bcd\_encoder

Questo componente si occupa di convertire un numero binario puro privo di segno nel suo corrispondente codificato secondo lo schema Binary Encoded Decimal, il quale prevede che vadano assegnati quattro bit a ogni cifra decimale. Ognuno di questi gruppi potrà quindi assumere solo valori nell'intervallo tra 0 e 9. In particolare il componente codifica su due cifre, H quella di grado maggiore e L quella inferiore.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity bcd_encoder is
    port (I: in std_logic_vector(7 downto 0);
          H, L: out std_logic_vector(3 downto 0));
end bcd_encoder;

```

**-- WARNING: This architecture is limited to numbers smaller than 39!**

```

architecture naive_behavioral of bcd_encoder is
begin
process(I)
    variable vI, t: unsigned(7 downto 0);

```

```

begin
  vI := to_01(unsigned(I));
  if vI >= 30 then
    H <= "0011";
    t := vI - 30;
  elsif vI >= 20 then
    H <= "0010";
    t := vI - 20;
  elsif vI >= 10 then
    H <= "0001";
    t := vI - 10;
  else
    H <= "0000";
    t := vI - 0;
  end if;
  L <= std_logic_vector(t(3 downto 0));
end process;
end naive_behavioral;

```

L'implementazione scelta è alquanto limitata e riduttiva, in quanto non in grado di convertire valori superiori a 39. Questo è dovuto al fatto che si è scelto di evitare l'uso di divisori e di usare semplici sottrattori e comparatori, in modo da rendere il componente puramente combinatorio. La limitazione è comunque tollerabile, in quanto per il progetto è necessario poter convertire solamente numeri inferiori a 20 più il valore della carta massima.

```

library std;
library ieee;

use std.textio.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all; -- synopsys only
use ieee.std_logic_textio.all; -- synopsys only

entity tb_bcd_encoder is
end tb_bcd_encoder;

architecture test of tb_bcd_encoder is
  signal I:      std_logic_vector(7 downto 0) := "00000000";
  signal H, L:  std_logic_vector(3  downto 0);

  signal n: integer := -1;

  component bcd_encoder
    port (I:      in  std_logic_vector(7 downto 0);

```

```

        H, L: out std_logic_vector(3 downto 0));
    end component;

begin
    U: bcd_encoder port map (I, H, L);

test: process
    variable l_out: line;
    variable ref: integer;
begin
    wait on n;
    while true loop
        ref := n;
        wait on n;

        assert H = conv_std_logic_vector(ref / 10, 4)
            report "Mismatch on output H";
        assert L = conv_std_logic_vector(ref mod 10, 4)
            report "Mismatch on output L";
    end loop;
end process;

I <= conv_std_logic_vector(n, I'length);

count: process
begin
    while n <= 39 loop
        n <= n + 1;
        wait for 0.1 ns;
    end loop;
    wait;
end process;
end test;

```

Il testbench in questo caso non legge alcun valore da un file esterno, bensì verifica il comportamento del componente con ognuno dei valori compresi nell'intervallo supportato facendo uso delle funzioni aritmetiche disponibili per controllarne la correttezza.

## 2.10 display

La visualizzazione dello stato del gioco è affidata al componente `display` il quale si occupa di codificare i valori e di gestire il refresh dei display, dal momento che sulla FPGA presa in considerazione è necessario accedervi in multiplexing.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity display is
    port (CLK, RST:          in  std_logic;
          PLAYER_L, PLAYER_H: in  std_logic_vector (3 downto 0);
          DEALER_L, DEALER_H: in  std_logic_vector (3 downto 0);
          PLAYER_SHOW, DEALER_SHOW: in  std_logic;
          PLAYER_WIN, DEALER_WIN: in  std_logic;
          OUTPUT:          out std_logic_vector (7 downto 0);
          AN:              out std_logic_vector (3 downto 0));
end display;

architecture structural of display is
    component sevensegment_encoder
        port (A: in  std_logic_vector(3 downto 0);
              O: out std_logic_vector(6 downto 0));
    end component;
    component mux21
        generic (N: integer);
        port (A, B: in  std_logic_vector (N-1 downto 0);
              SEL: in  std_logic;
              O:   out std_logic_vector (N-1 downto 0));
    end component;
    component display_controller
        port(CLK, RST: in  std_logic;
              DIGIT_3, DIGIT_2,
              DIGIT_1, DIGIT_0: in  std_logic_vector(7 downto 0);
              OUTPUT: out  std_logic_vector (7 downto 0);
              AN:     out  std_logic_vector (3 downto 0));
    end component;

    signal PL, PH, DL, DH: std_logic_vector (6 downto 0);
    signal ONES: std_logic_vector (6 downto 0) := (others => '1');
    signal DISPLAY_PL, DISPLAY_PH,
           DISPLAY_DL, DISPLAY_DH: std_logic_vector (6 downto 0);
    signal DIGIT_PL, DIGIT_PH,
           DIGIT_DL, DIGIT_DH: std_logic_vector(7 downto 0);
begin
    sse_pl: sevensegment_encoder port map (PLAYER_L, PL);
    sse_ph: sevensegment_encoder port map (PLAYER_H, PH);
    sse_dl: sevensegment_encoder port map (DEALER_L, DL);

```

```

sse_dh: sevensegment_encoder port map (DEALER_H, DH);

mux_pl: mux21 generic map(7)
        port map (ONES, PL, PLAYER_SHOW, DISPLAY_PL);
mux_ph: mux21 generic map(7)
        port map (ONES, PH, PLAYER_SHOW, DISPLAY_PH);
mux_dl: mux21 generic map(7)
        port map (ONES, DL, DEALER_SHOW, DISPLAY_DL);
mux_dh: mux21 generic map(7)
        port map (ONES, DH, DEALER_SHOW, DISPLAY_DH);

DIGIT_PL <= DISPLAY_PL & not PLAYER_WIN;
DIGIT_PH <= DISPLAY_PH & not PLAYER_WIN;
DIGIT_DL <= DISPLAY_DL & not DEALER_WIN;
DIGIT_DH <= DISPLAY_DH & not DEALER_WIN;

display: display_controller
        port map (CLK, RST,
                DIGIT_PH, DIGIT_PL, DIGIT_DH, DIGIT_DL,
                OUTPUT, AN);
end structural;

```

Nell'implementazione strutturale è possibile evidenziare le due funzioni svolte: la codifica è delegata a una batteria di `sevensegment_encoder` che convertono le cifre BCD nelle corrispondenti configurazioni per i display a sette segmenti. L'uscita di ognuno di questi encoder viene posta in ingresso a un multiplexer che controlla l'accensione o meno della relativa cifra sul display. Il multiplexing e il refresh sono invece effettuati dal componente `display_controller`.

```

library ieee;
use ieee.std_logic_1164.all;

entity sevensegment_encoder is
    port (A: in  std_logic_vector(3 downto 0);
          O: out std_logic_vector(6 downto 0));
end sevensegment_encoder;

architecture behavioral of sevensegment_encoder is
begin
process(A)
begin
    if    A = "0000" then
        O <= "0000001";
    elsif A = "0001" then

```

```

        O <= "1001111";
    elsif A = "0010" then
        O <= "0010010";
    elsif A = "0011" then
        O <= "0000110";
    elsif A = "0100" then
        O <= "1001100";
    elsif A = "0101" then
        O <= "0100100";
    elsif A = "0110" then
        O <= "0100000";
    elsif A = "0111" then
        O <= "0001111";
    elsif A = "1000" then
        O <= "0000000";
    elsif A = "1001" then
        O <= "0000100";
    elsif A = "1010" then
        O <= "0001000";
    elsif A = "1011" then
        O <= "1100000";
    elsif A = "1100" then
        O <= "0110001";
    elsif A = "1101" then
        O <= "1000010";
    elsif A = "1110" then
        O <= "0010000";
    elsif A = "1111" then
        O <= "0111000";
    end if;
end process;
end behavioral;

```

La codifica per i display a sette segmenti viene effettuata mediante una semplice tabella che in hardware viene implementata con una CAM o una più semplice ROM.

```

-- time RST P(H L) D(H L) P(SHOW) D(SHOW) P(WIN) D(WIN) DIGIT AN
1 0 0 0 0 0 0000 11111111 1111
2 1 1 2 3 4 1000 11111111 1111
3 1 1 2 3 4 1100 10011001 1110
4 1 1 2 3 4 1100 00001101 1101
5 1 1 2 3 4 1100 00100101 1011
6 1 1 2 3 4 1100 10011111 0111
7 1 1 2 3 4 1110 10011001 1110

```

```

8 1 1 2 3 4 1001 11111110 1101
9 1 1 2 3 4 1001 00100101 1011

```

Il test verifica il comportamento del circuito verificando i segnali di refresh (AN) e controllando che il valore della cifra che viene rinfrescata sia quello corretto.

## 2.11 display\_controller

La gestione del multiplexing e del refresh del display della FPGA è demandata al componente `display_controller`. Questo si occupa di effettuare il multiplexing temporale sull'uscita, ponendo in sequenza il valore di ciascuno dei segnali in ingresso. Il bit azzerato in AN indica la cifra che verrà rinfrescata.

```

library ieee;
use ieee.std_logic_1164.all;

entity display_controller is
    port(CLK, RST: in std_logic;
         DIGIT_3, DIGIT_2,
         DIGIT_1, DIGIT_0: in std_logic_vector(7 downto 0);

         OUTPUT: out std_logic_vector (7 downto 0);
         AN:     out std_logic_vector (3 downto 0));
end display_controller;

architecture structural of display_controller is
    component shift_reg is
        generic (N: integer := 8);
        port (CLK, RST: in std_logic;
              EN:      in std_logic;
              A:        in std_logic;
              D:        inout std_logic_vector (N-1 downto 0));
    end component;

    component mux21_1bit is
        port (A, B: in std_logic;
              SEL: in std_logic;
              O:   out std_logic);
    end component;

    component mux41_1bit is
        port (A, B, C, D: in std_logic;
              SEL:      in std_logic_vector(1 downto 0);
              O:        out std_logic);
    end component;

```

```

    end component;

    signal STARTED, SHIFT_IN: std_logic;
    signal SHIFT_OUT: std_logic_vector(3 downto 0);
    signal SEL: std_logic_vector(1 downto 0);
begin

STARTED  <= SHIFT_OUT(0) or SHIFT_OUT(1)
           or SHIFT_OUT(2) or SHIFT_OUT(3);
SHIFT_IN <= not STARTED or SHIFT_OUT(3);

SEL(0)  <= SHIFT_OUT(1) or SHIFT_OUT(3);
SEL(1)  <= SHIFT_OUT(2) or SHIFT_OUT(3);

reg: shift_reg generic map (4)
    port map (CLK, RST, '0', SHIFT_IN, SHIFT_OUT);

mux: for i in 0 to 7 generate
    mux_i: mux41_1bit
        port map (DIGIT_0(i), DIGIT_1(i), DIGIT_2(i), DIGIT_3(i),
            SEL, OUTPUT(i));
end generate;

AN <= not SHIFT_OUT;
end structural;

```

La descrizione è strutturale e fa uso di un registro a scorrimento per la gestione del multiplexing: inizialmente vuoto, viene caricato con un singolo bit che viene traslato in modo circolare ad ogni colpo di clock. Un vettore di multiplexer a quattro ingressi controllati dal registro a scorrimento determina il valore in uscita. I bit del registro a scorrimento vengono invertiti in quanto il segnale di abilitazione al refresh della FPGA è attivo basso.

```

-- time RST DIGIT_0 DIGIT_1 DIGIT_2 DIGIT_3 OUTPUT AN
1 1 11111111 11111111 11111111 11111111 11111111 1111
2 1 11110111 11111011 11111101 11111110 11111110 1110
3 1 11110111 11111011 11111101 11111110 11111101 1101
4 1 11110111 11111011 11111101 11111110 11111011 1011
5 1 11110111 11111011 11111101 11111110 11110111 0111
6 1 11110111 11111011 11111101 00000001 00000001 1110
7 1 11110111 11111011 10000000 00000001 10000000 1101
8 0 11110111 11111011 10000000 00000001 00000001 1111
9 1 11110111 11111011 11111101 00000001 00000001 1111
10 1 11110111 11111011 11111101 00000001 00000001 1110

```

Il test è simile a quello del componente `display` ma qui i valori in ingresso sono del tutto arbitrari.

## 2.12 `display_simulator`

Il componente `display_simulator` viene utilizzato solo per scopi di test e simula il comportamento del display della FPGA, restituendo come output i valori dei display convertiti in interi e con i punti decimali convertiti in valori booleani. I nomi degli ingressi riproducono i nomi dei pin disponibili sulla FPGA.

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

entity display_simulator is
    port(-- display
         E14, G13, N15, P15, R16, F13, N16: in std_logic;
         -- DP
         P16:                               in std_logic;
         -- AN(3 downto 0)
         E13, F14, G14, D14:               in std_logic;

         DIGITS_0, DIGITS_1, DIGITS_2, DIGITS_3: out character := ' ';
         DOTS_0, DOTS_1, DOTS_2, DOTS_3: out boolean);
end display_simulator;

architecture structural of display_simulator is
    signal DIGIT: std_logic_vector (6 downto 0);
    signal DOT: std_logic;
    signal AN: std_logic_vector (3 downto 0);
begin

    DIGIT <= E14 & G13 & N15 & P15 & R16 & F13 & N16;
    DOT <= P16;
    AN <= E13 & F14 & G14 & D14;

    process(DIGIT, DOT, AN)
        variable char: character;
        variable point: boolean;
    begin
        if DIGIT(6 downto 0) = "0000001" then
            char := '0';
        elsif DIGIT(6 downto 0) = "1001111" then
```

```

    char := '1';
elsif DIGIT(6 downto 0) = "0010010" then
    char := '2';
elsif DIGIT(6 downto 0) = "0000110" then
    char := '3';
elsif DIGIT(6 downto 0) = "1001100" then
    char := '4';
elsif DIGIT(6 downto 0) = "0100100" then
    char := '5';
elsif DIGIT(6 downto 0) = "0100000" then
    char := '6';
elsif DIGIT(6 downto 0) = "0001111" then
    char := '7';
elsif DIGIT(6 downto 0) = "0000000" then
    char := '8';
elsif DIGIT(6 downto 0) = "0000100" then
    char := '9';
elsif DIGIT(6 downto 0) = "1111111" then
    char := ' ';
else
    char := '*';
end if;

if DOT = '0' then
    point := True;
else
    point := False;
end if;

if AN(3) = '0' then
    DIGITS_0 <= char;
    DOTS_0 <= point;
end if;

if AN(2) = '0' then
    DIGITS_1 <= char;
    DOTS_1 <= point;
end if;

if AN(1) = '0' then
    DIGITS_2 <= char;
    DOTS_2 <= point;
end if;

```

```

    if AN(0) = '0' then
        DIGITS_3 <= char;
        DOTS_3 <= point;
    end if;
end process;
end structural;

```

L'architettura è comportamentale con un singolo processo in cui vengono decodificate le cifre mediante una tabella e in cui i valori decodificati vengono propagati verso l'uscita corretta.

```

-- time DIGIT DOT AN DIGITS DOTS
1 1111111 1 0000      False False False False
2 1001111 0 0111     1 False False False True
3 0001111 1 1101    7 1 False False False True
4 0000000 1 0111    7 8 False False False False
5 0010010 0 1110   27 8 True False False False
6 0000001 1 1101   20 8 True False False False
7 1111111 1 0111   20  True False False False

```

Il test simula l'uso del display verificando il comportamento dei punti e delle cifre.

### 2.13 pulse\_generator

Il generatore di impulsi viene usato semplicemente per evitare che una pressione prolungata di un pulsante da parte dell'utente venga campionata erroneamente in momenti successivi. Non appena viene campionato in input un fronte di salita, l'uscita rimane alta per un solo colpo di clock, rimanendo poi in attesa del fronte di salita successivo.

```

library ieee;
use ieee.std_logic_1164.all;

entity pulse_generator is
    port (CLK: in  std_logic;
          I:   in  std_logic;
          O:   out std_logic);
end pulse_generator;

architecture behavioral of pulse_generator is
begin
process (CLK)
    variable old: std_logic := '0';
begin

```

```

    if rising_edge(CLK) then
        if old = '0' and I = '1' then
            O <= '1';
        else
            O <= '0';
        end if;

        old := I;
    end if;
end process;
end behavioral;

```

La descrizione è comportamentale ed è molto semplice, facendo uso di una variabile per memorizzare il valore campionato in precedenza in modo di riconoscere i fronti di salita.

```

-- time RST I O
1 0 0
2 0 0
3 1 0
4 1 1
5 1 0
6 0 0
7 1 0
8 1 1
9 0 0
10 1 0
11 0 1
12 1 0
13 0 1

```

## 2.14 clock\_divider

Dal clock di sistema vengono generati due clock con periodi più lunghi per la logica di gioco e per la gestione del display. Per fare questo è stato usato un divisore di frequenza costruito sulla base di quanto riportato nelle FAQ del newsgroup comp.lang.vhdl.

-- Adapted from the example in the comp.lang.vhdl FAQ

```

library ieee;
use ieee.std_logic_1164.all;

entity clock_divider is
    generic (MODULUS: in positive range 2 to integer'high := 4);

```

```

    port (CLK, RST: in std_logic;
          O: out std_logic);
end clock_divider;

architecture behavioral of clock_divider is
begin
process (CLK, RST)
    variable count: natural range 0 to MODULUS-1;
begin
    if RST = '0' then
        count := 0;
        O <= '0';
    elsif rising_edge(CLK) then
        if count = MODULUS-1 then
            count := 0;
        else
            count := count + 1;
        end if;
        if count >= MODULUS/2 then
            O <= '0';
        else
            O <= '1';
        end if;
    end if;
end process;
end behavioral;

```

L'implementazione è comportamentale e contiene un contatore che viene incrementato. Il parametro MODULUS indica il numero di colpi di clock ad alta frequenza a cui corrisponde il nuovo clock: questo resterà alto per MODULUS/2 colpi e basso per i rimanenti.

```

-- time RST O
1 00
2 10
3 11
4 10
5 10
6 11
7 11
8 10
9 10
10 11

```

```

11 11
12 10
13 10
14 11
15 11

```

## 2.15 shift\_reg

Il componente `shift_reg` implementa un registro SIPO, in cui si ha un singolo ingresso seriale (A) mentre l'uscita è parallela (D).

```

library ieee;
use ieee.std_logic_1164.all;

entity shift_reg is
    generic (N: integer := 8);

    port (CLK, RST: in std_logic;
          EN:      in std_logic;
          A:       in std_logic;
          D:       inout std_logic_vector (N-1 downto 0));
end shift_reg;

architecture structural of shift_reg is
    component fd is
        port (CLK, RST: in std_logic;
              EN:      in std_logic;
              D:       in std_logic;
              Q:       out std_logic);
    end component;

begin
    fd_1: fd port map (CLK, RST, EN, A, D(0));

    fd_vect: for i in N-1 downto 1 generate
        fd_i: fd port map (CLK, RST, EN, D(i-1), D(i));
    end generate;
end structural;

```

L'architettura strutturale istanzia il flip-flop D a cui è connesso il segnale di input e poi procede a istanziare e connettere i restanti flip-flop collegati in cascata.

```

-- time RST EN A D
1 00 0 000
2 11 1 000

```

```

3 10 1 000
4 10 0 001
5 10 1 010
6 11 0 101
7 10 1 101
8 10 0 011

```

## 2.16 fd

Il componente `fd` implementa un flip-flop di tipo D con segnale di reset asincrono attivo basso e un segnale di abilitazione anch'esso attivo basso.

```

library ieee;
use ieee.std_logic_1164.all;

entity fda is
    port (CLK, RST: in std_logic;
          EN:      in std_logic;
          D:       in std_logic;
          Q:       out std_logic);
end fda;

architecture behavioral of fda is
begin
    process (CLK, RST)
    begin
        if RST = '0' then
            Q <= '0';
        elsif rising_edge(CLK) then
            if EN = '0' then
                Q <= D;
            end if;
        end if;
    end process;
end behavioral;

```

La descrizione comportamentale presenta i costrutti condizionali classici per l'implementazione di circuiti sincroni con reset asincrono.

```

-- time RST EN D Q
1 011 0
2 111 0
3 110 0
4 101 0
5 100 1

```

```
6 101 0
7 101 1
8 001 1
9 101 0
10 111 1
11 111 1
12 110 1
```

## 2.17 fa

Il componente `fa` implementa il full-adder necessario per la costruzione del Ripple Carry Adder.

```
library ieee;
use ieee.std_logic_1164.all;

entity fa is
    port (A, B: in std_logic;
          Ci: in std_logic;
          S: out std_logic;
          Co: out std_logic);
end fa;

architecture logic of fa is
begin
    S <= A xor B xor Ci;
    Co <= (A and B) or (A and Ci) or (B and Ci);
end logic;
```

Entrambi gli output sono definiti esclusivamente mediante le rispettive funzioni logiche.

```
-- time A B Ci S Co
1 00000
2 01010
3 10010
4 11001
5 00110
6 01101
7 10101
8 11111
9 00000
10 10010
11 00110
12 11001
13 01010
```

## 2.18 mux21

Questo componente implementa un multiplexer dotato di due ingressi e una uscita: se il segnale SEL è pari a 0 viene selezionato l'ingresso A, altrimenti quello B. Il parametro N specifica invece la dimensione in bit del multiplexer.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux21 is
    generic (N: integer := 8);

    port (A, B: in std_logic_vector (N-1 downto 0);
          SEL: in std_logic;
          O: out std_logic_vector (N-1 downto 0) );
end mux21;

architecture behavioral of mux21 is
begin
process (A, B, SEL)
begin
    if SEL = '0' then
        O <= A;
    else
        O <= B;
    end if;
end process;
end behavioral;
```

La descrizione comportamentale è tra le più semplici realizzabili, con solo una selezione condizionale tra due possibili assegnamenti.

```
-- time A B SEL O (terminating space needed)
1 00 00 0 00
2 01 00 0 01
3 00 01 1 01
4 10 00 0 10
5 00 10 1 10
6 11 00 0 11
7 00 11 1 11
8 01 10 0 01
9 01 10 1 10
10 11 00 1 00
11 11 10 1 10
12 01 11 0 01
13 01 10 1 10
```

## 2.19 mux21\_1bit

In modo analogo al componente mux21, mux21\_1bit implementa un multiplexer a due ingressi. Ingressi e uscite in questo caso però non sono vettori ma `std_logic` singoli.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux21_1bit is
    port (A, B: in std_logic;
          SEL: in std_logic;
          O: out std_logic);
end mux21_1bit;

architecture behavioral of mux21_1bit is
begin
    process (A, B, SEL)
    begin
        if SEL = '0' then
            O <= A;
        else
            O <= B;
        end if;
    end process;
end behavioral;
```

## 2.20 mux41\_1bit

La sola differenza tra il componente mux21\_1bit e mux41\_1bit è il fatto che quest'ultimo possiede quattro ingressi e, di conseguenza, il selettore è su due bit.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux41_1bit is
    port (A, B, C, D: in std_logic;
          SEL: in std_logic_vector(1 downto 0);
          O: out std_logic);
end mux41_1bit;

architecture behavioral of mux41_1bit is
begin
    process (A, B, C, D, SEL)
    begin
```

```
if    SEL = "00" then
    0 <= A;
elsif SEL = "01" then
    0 <= B;
elsif SEL = "10" then
    0 <= C;
elsif SEL = "11" then
    0 <= D;
end if;
end process;
end behavioral;
```

### 3 Ulteriori informazioni

L'intero progetto è disponibile presso <http://techn.ocracy.org/blackjack>, da cui è inoltre possibile visionare tutte le revisioni realizzate durante lo sviluppo. Il repository contiene inoltre i sorgenti di questo documento e il Makefile per GNU make usato per automatizzare il processo di compilazione e test.