```
-- Impliments craps logic to decide who wins the game
-- How it works:
-- Uses two linked state machines; the game state
-- and the roll changes state. Whenever the game
-- needs a sum to decide the next state it waits
-- until the changes state is in the both_rolls_changed
-- state. The game continues until it enters the win
-- or lose state where it waits for a reset to change

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity craps_game is
        port(roll_1: in std_logic_vector(2 downto 0);
                        roll_2: in std_logic_vector(2 downto 0);
                        roll_1_pressed: in std_logic;
                        roll_2_pressed: in std_logic;
                        clk: in std_logic;
                        rst: in std_logic;
                        sum: out std_logic_vector(3 downto 0);
                        win: out std_logic;
                        lose: out std_logic;
                        roll_1_stored: out std_logic_vector(2 downto 0);
                        roll_2_stored: out std_logic_vector(2 downto 0));
end craps_game;

architecture rtl of craps_game is
        signal lastPress1, lastPress2, currPress1, currPress2: std_logic;
        type state_t_game is (firstroll, firstroll_check, morerolls, morerolls_check,
win_s, lose_s);
        signal currstate_game, nextstate_game: state_t_game;
        -- state machine for the game
        type state_t_rolls is (nochange, changed_roll_1, changed_roll_2, both_rolls_ch
anged);
        signal currstate_rolls, nextstate_rolls: state_t_rolls;
        -- state machine for the rolls
        signal currsum, nextsum: unsigned(3 downto 0);
        signal currpoint, nextpoint: unsigned(3 downto 0);
        signal currroll_1, currroll_2, nextroll_1, nextroll_2: unsigned(2 downto 0);
        signal currchanged_1, currchanged_2, nextchanged_1, nextchanged_2: std_logic;
begin
        registers:process(clk, rst)
        begin
                if(rst = '0') then
                        currstate_game <= firstroll;
                        currstate_rolls <= nochange;
                        currsum <= "0000";
                        currroll_1 <= "000";
                        currroll_2 <= "000";
                        currchanged_1 <= '0';
                        currchanged_2 <= '0';
                        currpoint <= "0000";
                        lastPress1 <= '1';
                        lastPress2 <= '1';
                        currPress1 <= '1';
                        currPress2 <= '1';
                elsif (clk'event and clk = '1') then
                        currstate_game <= nextstate_game; -- update game state
                        currstate_rolls <= nextstate_rolls; -- update rolls state
                        currsum <= nextsum;
                        currroll_1 <= nextroll_1;
                        currroll_2 <= nextroll_2;
                        currchanged_1 <= nextchanged_1; -- detects new roll
```

```
                        currchanged_2 <= nextchanged_2;
                        currpoint <= nextpoint;
                        lastPress1 <= currPress1; -- update what the last press is
                        lastPress2 <= currPress2;
                        currPress1 <= roll_1_pressed; -- update the current press
                        currPress2 <= roll_2_pressed;
                end if;
        end process;

        -- state machine for managing the rolls
        state_machine_rolls: process(currstate_rolls, roll_1, roll_2, currroll_1, curr
roll_2, currstate_game)
        begin

                -- default roll over
                nextstate_rolls <= currstate_rolls;
                -- this detects when dice have been rolled and prevents rerolling
                case currstate_rolls is
                        when nochange =>
                                -- continue taking in rolls waiting for change
                                nextroll_1 <= unsigned(roll_1);
                                nextroll_2 <= unsigned(roll_2);
                                if(currchanged_1 = '1') then -- if changed
                                        nextstate_rolls <= changed_roll_1;
                                elsif(currchanged_2 = '1') then
                                        nextstate_rolls <= changed_roll_2;
                                end if;
                        when changed_roll_1 =>
                                -- stop taking in roll 1, continue waiting for roll 2
                                nextroll_1 <= currroll_1;
                                nextroll_2 <= unsigned(roll_2);
                                if(currchanged_2 = '1') then
                                        nextstate_rolls <= both_rolls_changed;
                                end if;
                        when changed_roll_2 =>
                                -- stop taking in roll 2, continue waiting for roll 1
                                nextroll_1 <= unsigned(roll_1);
                                nextroll_2 <= currroll_2;
                                if(currchanged_1 = '1') then
                                        nextstate_rolls <= both_rolls_changed;
                                end if;
                        -- roll over until the state machine for the game changes to t
he checking states
                        when both_rolls_changed =>
                                -- if nothing did change
                                if((currstate_game = firstroll_check) or (currstate_ga
me = morerolls_check)) then
                                        nextstate_rolls <= nochange;
                                end if;
                                -- will set next to be what it was before with cur
                                nextroll_1 <= currroll_1;
                                nextroll_2 <= currroll_2;
                end case;

        end process;

        -- state machine for detecting a new roll using the keys that are pressed
        changed_flag_statemachines: process(rst, currstate_rolls, roll_1_pressed, roll
_2_pressed, currchanged_1, currchanged_2, currPress1, currPress2, lastPress1, lastPres
s2)
        begin
                -- set flags if change is detected
                if(rst = '0') then
                        nextchanged_1 <= '0';
                -- on rising edge of button press set change flag to 1
```

```
                elsif(currPress1 = '1' and lastPress1 = '0') then
                        nextchanged_1 <= '1';
                -- reset flags if currstate_rolls is both changed
                elsif(currstate_rolls = both_rolls_changed) then
                        nextchanged_1 <= '0';
                -- roll over values otherwise
                else
                        nextchanged_1 <= currchanged_1;
                end if;

                -- set flags if change is detected
                if(rst ='0') then
                        nextchanged_2 <= '0';
                -- on rising edge of button press set change flag to 1
                elsif(currPress2 = '1' and lastPress2 = '0') then
                        nextchanged_2 <= '1';
                -- reset flags if currstate_rolls is both changed
                elsif(currstate_rolls = both_rolls_changed) then
                        nextchanged_2 <= '0';
                -- roll over values otherwise
                else
                        nextchanged_2 <= currchanged_2;
                end if;
        end process;

        state_machine_game: process(currstate_game, currstate_rolls, currsum, currpoin
t)
        begin
                -- default rollovers
                nextsum <= currsum;
                nextpoint <= currpoint;
                case currstate_game is
                        when firstroll =>
                                -- check if both dice rolled
                                if(currstate_rolls = both_rolls_changed) then
                                        -- prep the sum and go to check sum state
                                        nextstate_game <= firstroll_check;
                                        nextsum <= ('0' & currroll_1) + ('0' & currrol
l_2);
                                else
                                        -- wait
                                        nextstate_game <= currstate_game;
                                end if;
                        when firstroll_check =>
                                case currsum is
                                        -- check for winning sums
                                        when "0111" => nextstate_game <= win_s;
                                        when "1011" => nextstate_game <= win_s;
                                        -- check for losing sums
                                        when "0010" => nextstate_game <= lose_s;
                                        when "0011" => nextstate_game <= lose_s;
                                        when "1100" => nextstate_game <= lose_s;
                                        -- more rolls needed otherwise
                                        when others =>
                                                -- set point register
                                                nextpoint <= currsum;
                                                -- reset sum
                                                nextsum <= currsum;
                                                -- go to next state of the game -> mor
e rolls needed
                                                nextstate_game <= morerolls;
                                end case;
                        when morerolls =>
                                -- check if both dice rolled
                                if(currstate_rolls = both_rolls_changed) then
```

```
                                        -- prep the sum and go to check sum state
                                        nextstate_game <= morerolls_check;
                                        nextsum <= ('0' & currroll_1) + ('0' & currrol
l_2);
                                else
                                        -- wait
                                        nextstate_game <= currstate_game;
                                end if;
                        when morerolls_check =>

                                -- check if winning sum
                                if (currsum = currpoint) then
                                        nextstate_game <= win_s;
                                -- check if losing sum
                                elsif (currsum = 7) then
                                        nextstate_game <= lose_s;
                                -- go back to more rolls state otherwise
                                else
                                        -- reset sum
                                        nextsum <= currsum;
                                        -- more rolls needed
                                        nextstate_game <= morerolls;
                                end if;
                        -- wait until reset
                        when win_s => nextstate_game <= currstate_game;
                        when lose_s => nextstate_game <= currstate_game;
                end case;
        end process;

        output_logic_game: process(currstate_game)
        begin
                case currstate_game is
                        when win_s =>
                                win <= '1';
                                lose <= '0';
                        when lose_s =>
                                win <= '0';
                                lose <= '1';
                        when others =>
                                win <= '0';
                                lose <= '0';
                end case;
        end process;

        -- dummy assignment
        sum <= std_logic_vector(currsum);
        roll_1_stored <= std_logic_vector(currroll_1);
        roll_2_stored <= std_logic_vector(currroll_2);
end rtl;
```