# COL 215 Lab Project Design Document

# 7up 7down

Made By: Sankalan Pal Chowdhury(2016CS10701) and Shresth Tuli(2016CS10680)

# Description

The purpose of the project is to execute a 7/up 7/down game. It involves two players, bidding an amount from their purses and guessing whether the sum of the number on two dices will be greater than 7 or less than 7. If their guess is right, they get the bid amount from the other player's purse. The game ends when one of the players goes bankrupt.

## Rules:

- Both players start off with 1008 units of currency. These are indivisible. The choice is arbitrary.
- The bidding process is sequential and not concurrent i.e. One player bids after the other is done. This is because there is no way of hiding the bidding of one person from the other. If bidding is made concurrent, we may end up with a situation where both players want to go second, and hence stall the game.
- The bidding is open, each player gets to see what the other has bid. Again, this is natural given the game setting.
- At each bid, 25% of the bid amount is deducted as deposit. This is done to deter players from making very high bids. The entire deposit amount is given to the winning player at the end of the game. Floors are taken when required.
- The maximum amount that can be bid is 127 units, and is higher than what the previous rule would prompt one to bid. The choice is somewhat arbitrary, but it has to be there and should preferably be of the form 2^n-1 to make full use of switches.
- The player who bids first must ensure the amount he is bidding is not more than the amount remaining in the other player's purse as well as that he can pay the 25% deposit from his own purse.
- The second bidder must ensure the same, and must be able to pay the deposit even after subtracting the first players bid from his purse. This is required to ensure that the amount in his purse does not become negative irrespective of the outcome of the throw of dice.
- Failing the previous two conditions, the bid would not be accepted and the game would not progress until a valid bid is obtained.
- The player who bids first in one cycle must bid second in the next cycle, to make the process fully fair.
- The throw of dice is simulated by a random number generator.
- If 7 is obtained, player with higher bid wins. If both player have equal bids, nothing happens. Again, this is to promote higher bids.

# Overall Approach

The design can be split into three parts:

- The Random number generator
- The Control unit
- The input-output block

The Control unit has to be an FSM, as it is entirely sequential in nature.

The random number generator is the interesting part. There are several methods of generating pseudo-random numbers, but our requirement is limited: We need a random number only when a button is pressed. And since the button would be pressed by a human, we can count the number of clock cycles for which the button is pressed, which boils down to a single modulo counter. Note that this means that our Control FSM should also be pulse triggered and not edge triggered.

The fastest neurons in human body can fire at most 100 times in a second, giving a frequency of about 100Hz. We can then assume noise of the order 1 ms in the time of pressing. Since our clock runs at 100 MHz, the number of clock cycles for which button is pressed mod 36 is completely random. The actual number is obtained by assigning empirical probability weights to the outcome, as would be in the case of a double dice throw.

The IO unit could be done on the Basys 3 board itself, or on a VGA display. We intend to do the first to begin with, and the second only if time permits.

## How we came up with the rules

The project statement supplied to us offered a much simpler game: bid an amount, and if you win, you get it from the other player's purse. Now while this sounds easy, there are some problems with it.

In general, for a game to be enjoyable, it must fulfill three criteria:

- It must not have a trivial strategy to get a stalemate or victory.
- The players must interact
- It must complete

Now let us look at our current game in this perspective:

- Does the game have a trivial strategy? Well, any player would want to win as much as possible. Game theoretically, we can equate the payoffs of a player to what he wins in a given situation multiplied by the probability of the given situation arising. Here this boils down to (amount bid)*(15/36) which is directly proportional to the amount bid by the player. To maximize the payoff, one would bid as high as possible. And that is a trivial strategy.
- Do the players interact? What I bid does come out of the other player's purse, but that has no effect on his strategy. So, they don't.

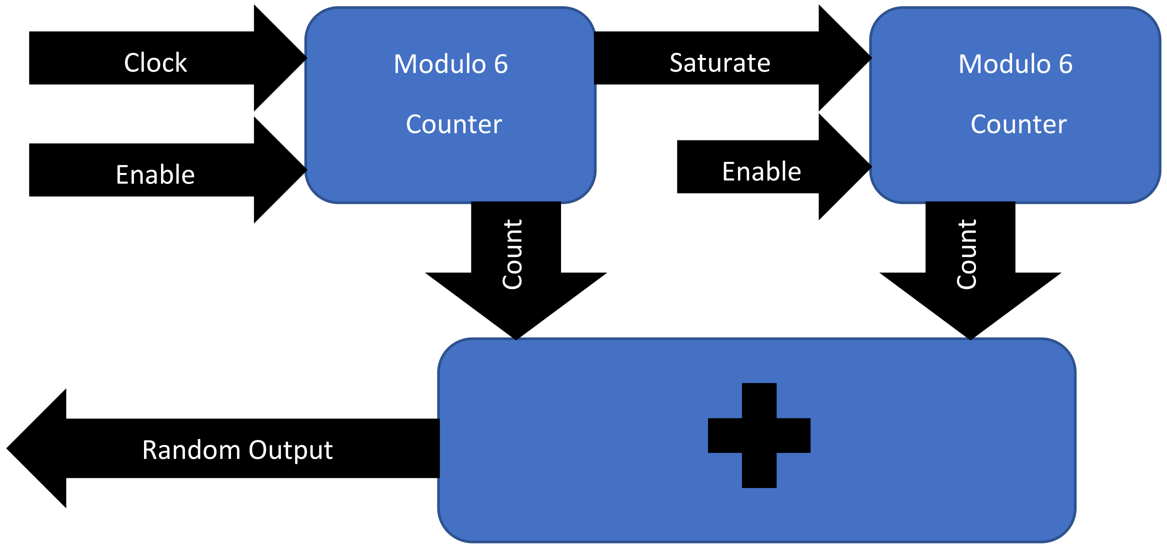
- Does the game terminate? Well, there is no guarantee to it. In fact, if whoever goes second just matches the bid of the first player, the game is assured to carry on indefinitely.

So, what can be done?

- Firstly, we need to have a mechanism to deter players form making high bids. This is done by the deposit rule. Every time you make a bid, you have to keep a part of your money(proportional to your bid) as a deposit, which cannot be used again during the game. Our original idea was to just throw this part away as tax, but that would make it difficult to check the game: the evaluator would have to manually track all the tax. To make the job easy, we add all the deposit to the winner's purse at the end. This change is trivial and can be undone if required.
- Next this is to make the players interact. The project statement does not specify what to do if we get a seven. We use it here. If we get a 7, the player with the higher bid wins. This means that what the second player bids now depends partially on the first players bid. Thus, they have interaction.
- Finally, termination. Now, this is also taken care by deposit: the amount of currency in the game reduces in each step, and thus the game heads towards termination. In fact, on solving, we find that the Nash equilibrium of the game is around 39 units bid per turn, which means in each turn, 18 units are thrown out of the game.

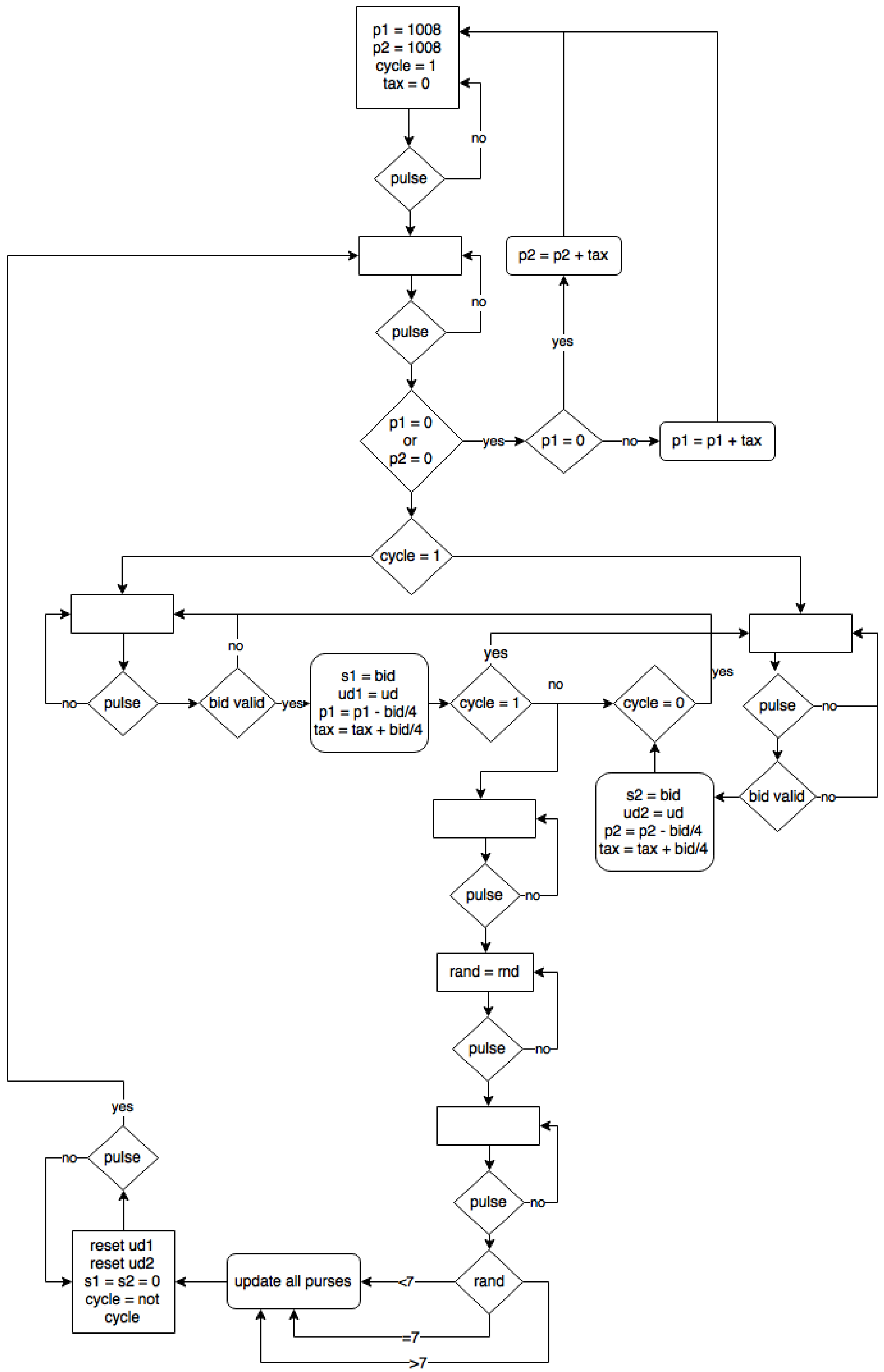
# Block Diagram

## Random Number Generator:



## Control FSM:

This circuit is much more involved and a block diagram is not easy to draw. Nevertheless, The ASM is attached.


p1 = 1008
p2 = 1008
cycle = 1
tax = 0
pulse
no
pulse
no
p2 = p2 + tax
yes
p1 = 0
or
p2 = 0
yes
p1 = 0
no
p1 = p1 + tax
cycle = 1
no
pulse
no
bid valid
yes
s1 = bid
ud1 = ud
p1 = p1 - bid/4
tax = tax + bid/4
yes
cycle = 1
no
cycle = 0
yes
pulse
no
bid valid
no
s2 = bid
ud2 = ud
p2 = p2 - bid/4
tax = tax + bid/4
pulse
no
rand = rnd
pulse
no
pulse
no
rand
<7
=7
>7
update all purses
reset ud1
reset ud2
s1 = s2 = 0
cycle = not
cycle
pulse
no
yes

This entire thing is implemented synchronously with a clock and a proceed button. Since the state needs to change on a pulse and not an edge, The validation is performed when proc goes from 0=>1 but actual state transition takes place when it goes from 1=>0. The transitions are handled by a case-when block.

Code for Control unit is also attached

### IO Block:

Not finalized presently, but would not involve any complex circuitry except for the SSD;

## Testing

Since we rely upon the uncertainty of a human pressing a button, the entire circuit cannot be tested on a simulated test bench. We can however test by playing the game. As far as the control unit is concerned, it can be tested on a test bench, where the Random values are manually provided. The corner cases are the termination cases and bid valid cases.

# VHDL Implementation of Control Block

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;


entity Control is
    Port ( proc : in STD_LOGIC;
           amt1 : out STD_LOGIC_VECTOR(10 downto 0);
           amt2 : out STD_LOGIC_VECTOR(10 downto 0);
           tax : out STD_LOGIC_VECTOR(10 downto 0);
           bid : in STD_LOGIC_VECTOR (6 downto 0);
           rnd : in STD_LOGIC_VECTOR (3 downto 0);
           st : out STD_LOGIC_VECTOR (2 downto 0);
           ud: in STD_LOGIC;
           O_ud1,O_ud2:out STD_LOGIC_VECTOR(1 downto 0);
           sub1 : out STD_LOGIC_VECTOR (6 downto 0);
           sub2 : out STD_LOGIC_VECTOR (6 downto 0);
           clk: in STD_LOGIC);
end Control;

architecture Behavioral of Control is
signal state:std_logic_vector(3 downto 0):="0000";
signal Next_state:std_logic_vector(2 downto 0);
--need to initialise this--
signal p1,p2,tx:std_logic_vector(10 downto 0);
--the two purses. Need to be latches/flip-flops--
signal s1,s2:std_logic_vector(6 downto 0);      --the two bids--
signal bid_val:std_logic;
signal ud1,ud2:std_logic_vector(1 downto 0):="00";
signal cycle:std_logic:='1';
signal rand:std_logic_vector(3 downto 0);
begin
    st<=state(2 downto 0);
    amt1<=p1;
    amt2<=p2;
    tax<=tx;
    sub1<=s1;
    sub2<=s2;
    O_ud1<=ud1;
    O_ud2<=ud2;
    process(clk)
    begin
        if(clk='1' and clk'event) then
            if(state(3)='1') then
                if(proc='0') then
                    state<='0' & next_state;
                end if;
            else
                if(proc='1') then
                     case state(2 downto 0) is
```

```
                        when "000"=>
                        --init:initialise both the purses---
                                p1<="01111110000";
                                p2<="01111110000";
                                tx<="00000000000";
                                state(3)<='1';
                                next_state<="001";
                        when "001"=>
                        --start:display both purse values. Decide next state
depending on cycle--
                                state(3)<='1';
                                if(cycle='1')then
                                    next_state<="010";
                                else
                                    next_state<="011";
                                end if;
                                if(p1="00000000000") then
                                    p2<=p2+tx;
                                    tx<="00000000000";
                                    next_state<="000";
                                elsif(p2="00000000000")then
                                    p1<=p1+tx;
                                    tx<="00000000000";
                                    next_state<="000";
                                end if;
                        when "010"=>
                        --bid1:take bid from player 1. Proceed if bid is
valid--
                                if((cycle='1' and unsigned(bid)<=unsigned(p2)
and unsigned(bid(6 downto 2))<=unsigned(p1))or(cycle='0' and
unsigned(bid)<=unsigned(p2) and unsigned(bid(6 downto 2))<=(unsigned(p1)-
unsigned(s2)))) then
                                    s1<=bid;
                                    if(ud='1') then
                                        ud1<="10";
                                    else
                                        ud1<="01";
                                    end if;
                                    p1<=p1-bid(6 downto 2);
                                    tx<=tx+bid(6 downto 2);
                                    state(3)<='1';
                                    if(cycle='1') then
                                        next_state<="011";
                                    else
                                        next_state<="100";
                                    end if;
                                end if;
                        when "011"=>
                        --bid2:take bid from player 2. Proceed if bid is
valid--
                            if((cycle='0' and unsigned(bid)<=unsigned(p1) and
unsigned(bid(6 downto 2))<=unsigned(p2))or(cycle='1' and
unsigned(bid)<=unsigned(p1) and unsigned(bid(6 downto 2))<=(unsigned(p2)-
unsigned(s1)))) then
                                s2<=bid;
                                if(ud='1') then
                                    ud2<="10";
                                else
                                    ud2<="01";
                                end if;
                                p2<=p2-bid(6 downto 2);
                                tx<=tx+bid(6 downto 0);
                                state(3)<='1';
                                if(cycle='1')then
                                    next_state<="100";
                                else
                                    next_state<="010";
                                end if;
                            end if;
                        when "100"=>
                        --Buffer state:move to next state unconditionally--
                            state(3)<='1';
                            next_state<="101";
                        when "101"=>
                        --Get Random:The random number genereator works in
the 100=>101 transition. Random output can be displayed here--
                            rand<=rnd;
                            state(3)<='1';
                            next_state<="110";
                        when "110"=>
                        --Main calc: Check the result and update all the
purses--
                            if(rand>"0111")then
                                if(ud1="10")then
                                    p1<=p1+s1;
                                    p2<=p2-s1;
                                end if;
                                if(ud2="10")then
                                    p1<=p1-s2;
                                    p2<=p2+s2;
                                end if;
                            elsif(rand<"0111")then
                                if(ud1="01")then
                                    p1<=p1+s1;
                                    p2<=p2-s1;
                                end if;
                                if(ud2="01")then
                                    p1<=p1-s2;
                                    p2<=p2+s2;
                                end if;
                            else
                                if(s1>s2) then
                                    p1<=p1+s1;
                                    p2<=p2-s1;
                                elsif(s2>s1) then
                                    p1<=p1-s2;
                                    p2<=p2+s2;
                                end if;
                            end if;
                            state(3)<='1';
                            next_state<="111";
                        when "111"=>
                        --reset:reset all values
                            cycle<=not cycle;
```

```vhdl
                            ud1<="00";
                            ud2<="00";
                            s1<="0000000";
                            s2<="0000000";
                            state(3)<='1';
                            next_state<="001";
                        when others=>
                            state(3)<='1';
                            next_state<="000";
                        end case;
                end if;
            end if;
        end if;


    end process;


end Behavioral;
```