



**KOÇ
UNIVERSITY**

ELEC 204 Project Report

Title: Design and Implementation of a Random
Number Generator using Lagged-Fibonacci
Generator

Name: Ali Utkan Şahin, Hakan Şahin

Date: May 19th, 2019

Lab Section: 11:30 - 14:15, Friday, LABB

Introduction

The aim of this project is to design a random number generator with equal probability in every number. When we press the button of our choice, it will print out a number between 0 and $2^{13} - 1$, with equal probability.

In order to achieve this, first we have divided our project into 3 main parts. Linear Congruential Generator (LCG), Seed Generator (SeedGen) and Lagged-Fibonacci-Generator (LFG). We use LCG to generate a number between 0 and $(2^{16} - 1)$ with the help of the clock. Later, we will use LCG to generate 7 seeds in order to implement our main random number generator, LFG which requires seven seeds. Then in LFG, we sum up the 7th behind and 3rd behind of our last term and then we take the modulo of 2^{13} in our Lagged-Fibonacci Series since we want our maximum number to be $2^{13} - 1$, then we shift them so we can generate new random numbers.

Methodology

In order to implement the random generator, first we have designed a finite state machine of LCG, which follows the basic formula: $R_{n+1} = R_n * a + b \pmod{M}$, where a , b are constants and M is 2^{16} . The seed of LCG is a random number that increases with the rising edge of the clock. Then we use this LCG, to generate seven seeds another FSM, then we use this generated seeds to implement our Lagged-Fibonacci Generator, which uses another FSM with the following formula: $R_t = R_{t-j} + R_{t-k} \pmod{M}$ where $0 < j < k$, $j = 3$, $k = 7$, $M = 2^{13}$ since we want our maximum number to be $2^{13} - 1$. Then we use a BCD Converter, which uses another FSM to convert our 13-bit number to Binary Coded Decimal in order to print our result the Seven-Segment Display. Then we combine all of these under Main module, which is also a Finite State Machine as well.

Linear Congruential Gener

```

32 entity LCG is
33   Generic ( a : integer := 29;
34             b : integer := 7;
35             m : integer := 65535 -- 2^16-1
36             );
37   Port ( CLK : in  STD_LOGIC;
38         RUN : in  STD_LOGIC;
39         RUNNING : out STD_LOGIC;
40         DONE : out STD_LOGIC;
41         CURR : inout unsigned (0 to 15));
42 end LCG;
43
44 architecture Behavioral of LCG is
45   -- fsm begin
46   constant s_init : STD_LOGIC_VECTOR(0 to 1) := "00";
47   constant s_calculate : STD_LOGIC_VECTOR(0 to 1) := "01";
48   constant s_done : STD_LOGIC_VECTOR(0 to 1) := "11";
49   signal State : STD_LOGIC_VECTOR(0 to 1) := s_init;
50   -- fsm end
51
52 begin
53   process(CLK)
54     variable curr_int : integer range 0 to m := 30;
55     begin
56       if(rising_edge(CLK)) then
57         case State is
58           when s_init =>
59             DONE <= '0';
60
61             curr_int := curr_int + 1;
62             CURR <= to_unsigned(0, 16);
63             if(RUN = '1') then
64               State <= s_calculate;
65             else
66               State <= s_init;
67             end if;
68           when s_calculate =>
69             DONE <= '0';
70             RUNNING <= '1';
71             curr_int := (curr_int * a + b);
72             State <= s_done;
73           when s_done =>
74             DONE <= '1';
75             CURR <= to_unsigned(curr_int, 16);
76             if(RUN = '1') then
77               State <= s_calculate;
78             else
79               State <= s_init;
80             end if;
81           when others =>
82             State <= s_init;
83         end case;
84       end if;
85     end process;
86 end Behavioral;

```

Figure 1: LCG.vhd

As explained above, this is our LCG which is used to generate seeds with the formula that was given above. It generates a 16-bit binary number between 0 and $2^{16} - 1$ as soon as RUN, our input becomes 1, which later becomes controlled by our button. RUNNING and DONE outputs were created in order to make our FSMs work simultaneously. In this code our formula becomes: $\text{curr_int} * a + b \pmod{2^{16}}$ where $a = 29$ and $b = 7$, and curr_int is an increasing integer with the rising edge of the clock and between 0 and 2^{16} . The rest of the code is self-explanatory.

Seed Generator

```

32 entity SeedGen is
33   Port ( CLK : in  STD_LOGIC;
34         RUN : in  STD_LOGIC;
35         S0 : out  unsigned (0 to 15);
36         S1 : out  unsigned (0 to 15);
37         S2 : out  unsigned (0 to 15);
38         S3 : out  unsigned (0 to 15);
39         S4 : out  unsigned (0 to 15);
40         S5 : out  unsigned (0 to 15);
41         S6 : out  unsigned (0 to 15);
42         DONE : out STD_LOGIC
43   );
44 end SeedGen;
45
46 architecture Behavioral of SeedGen is
47   -- fsm begin
48   constant s_init : STD_LOGIC_VECTOR(0 to 3) := "0000";
49   constant s_start : STD_LOGIC_VECTOR(0 to 3) := "1011";
50   constant s_0 : STD_LOGIC_VECTOR(0 to 3) := "00001";
51   constant s_1 : STD_LOGIC_VECTOR(0 to 3) := "00010";
52   constant s_2 : STD_LOGIC_VECTOR(0 to 3) := "00011";
53   constant s_3 : STD_LOGIC_VECTOR(0 to 3) := "0100";
54   constant s_4 : STD_LOGIC_VECTOR(0 to 3) := "0101";
55   constant s_5 : STD_LOGIC_VECTOR(0 to 3) := "0110";
56   constant s_6 : STD_LOGIC_VECTOR(0 to 3) := "0111";
57   constant s_done : STD_LOGIC_VECTOR(0 to 3) := "1001";
58   signal State : STD_LOGIC_VECTOR(0 to 3) := s_init;
59   -- fsm end
60
61   signal lcg_curr : unsigned(0 to 15);
62   signal lcg_run : STD_LOGIC := '0';
63   signal lcg_running : STD_LOGIC;
64   signal lcg_done : STD_LOGIC;
65
66 begin
67   LCG : entity work.LCG(Behavioral)
68     PORT MAP(
69       CLK => CLK,
70       RUN => lcg_run,
71       CURR => lcg_curr,
72       RUNNING => lcg_running,
73       DONE => lcg_done
74     );
75
76   process (CLK)
77   begin
78     if(rising_edge(CLK)) then
79       case State is
80         when s_init =>
81           DONE <= '0';
82           lcg_run <= '0';
83           S0 <= to_unsigned(0, 16);
84           S1 <= to_unsigned(0, 16);
85           S2 <= to_unsigned(0, 16);
86           S3 <= to_unsigned(0, 16);
87           S4 <= to_unsigned(0, 16);
88           S5 <= to_unsigned(0, 16);
89           S6 <= to_unsigned(0, 16);
90           if(RUN = '1') then
91             State <= s_start;
92           else
93             State <= s_init;
94           end if;
95
96         when s_start =>
97           DONE <= '0';
98           lcg_run <= '1';
99           if(lcg_done = '1') then
100             State <= s_0;
101           else
102             State <= s_start;
103           end if;
104         when s_0 =>
105           S0 <= lcg_curr;
106           if(lcg_done = '1') then
107             State <= s_1;
108           else
109             State <= s_0;
110           end if;
111         when s_1 =>
112           S1 <= lcg_curr;
113           if(lcg_done = '1') then
114             State <= s_2;
115           else
116             State <= s_1;
117           end if;
118         when s_2 =>
119           S2 <= lcg_curr;
120           if(lcg_done = '1') then
121             State <= s_3;
122           else
123             State <= s_2;
124           end if;
125         when s_3 =>
126           S3 <= lcg_curr;
127           if(lcg_done = '1') then
128             State <= s_4;
129           else
130             State <= s_3;
131           end if;
132         when s_4 =>
133           S4 <= lcg_curr;
134           if(lcg_done = '1') then
135             State <= s_5;
136           else
137             State <= s_4;
138           end if;
139         when s_5 =>
140           S5 <= lcg_curr;
141           if(lcg_done = '1') then
142             State <= s_6;
143           else
144             State <= s_5;
145           end if;
146         when s_6 =>
147           S6 <= lcg_curr;
148           if(lcg_done = '1') then
149             State <= s_done;
150           else
151             State <= s_6;
152           end if;
153         when s_done =>
154           DONE <= '1';
155           lcg_run <= '0';
156           if(RUN = '1') then
157             State <= s_done;
158           else
159             State <= s_init;
160           end if;
161         when others =>
162           State <= s_init;
163       end case;
164     end if;
165   end process;
166
167 end Behavioral;

```

Figure 2: SeedGen.vhd

This is the part where we generate the seeds we need for the next part, Lagged-Fibonacci Generator. Since we look at the 7th behind and 3rd behind. Generating 7 seeds from S0 to S6 would be enough for our implementation. This part also uses a FSM to generate seeds. The code is pretty much self-explanatory, it takes 7 randomly generated seeds from the LCG.

Lagged-Fibonacci Generator

```

32 entity LFG is
33   Port ( CLK : in  STD_LOGIC;
34         RUN : in  STD_LOGIC;
35         DONE : out STD_LOGIC;
36         CURR : out unsigned (0 to 16));
37 end LFG;
38
39 architecture Behavioral of LFG is
40   -- fsm begin
41   constant s_init : STD_LOGIC_VECTOR(0 to 1) := "00";
42   constant s_acquire : STD_LOGIC_VECTOR(0 to 1) := "01";
43   constant s_calculate : STD_LOGIC_VECTOR(0 to 1) := "10";
44   constant s_done : STD_LOGIC_VECTOR(0 to 1) := "11";
45   signal State : STD_LOGIC_VECTOR(0 to 1) := s_init;
46   -- fsm end
47
48   signal seedgen_r0 : unsigned(0 to 15);
49   signal seedgen_r1 : unsigned(0 to 15);
50   signal seedgen_r2 : unsigned(0 to 15);
51   signal seedgen_r3 : unsigned(0 to 15);
52   signal seedgen_r4 : unsigned(0 to 15);
53   signal seedgen_r5 : unsigned(0 to 15);
54   signal seedgen_r6 : unsigned(0 to 15);
55   signal seedgen_run : STD_LOGIC;
56   signal seedgen_done : STD_LOGIC;
57
58 begin
59   SDGN : entity work.SeedGen(Behavioral)
60     PORT MAP(
61       CLK => CLK,
62       DONE => seedgen_done,
63       RUN => seedgen_run,
64       S0 => seedgen_r0,
65       S1 => seedgen_r1,
66       S2 => seedgen_r2,
67       S3 => seedgen_r3,
68       S4 => seedgen_r4,
69       S5 => seedgen_r5,
70       S6 => seedgen_r6
71     );
72
73   process(CLK)
74     variable r0 : integer range 0 to 2**16 := 0;
75     variable r1 : integer range 0 to 2**16 := 0;
76     variable r2 : integer range 0 to 2**16 := 0;
77     variable r3 : integer range 0 to 2**16 := 0;
78     variable r4 : integer range 0 to 2**16 := 0;
79     variable r5 : integer range 0 to 2**16 := 0;
80     variable r6 : integer range 0 to 2**16 := 0;
81     variable r7 : integer range 0 to 2**16 := 0;
82   begin
83     if(rising_edge(CLK)) then
84       case State is
85         when s_init =>
86           r0 := 0;
87           r1 := 0;
88
89           r2 := 0;
90           r3 := 0;
91           r4 := 0;
92           r5 := 0;
93           r6 := 0;
94           r7 := 0;
95           DONE <= '0';
96           seedgen_run <= '0';
97           if(RUN = '1') then
98             State <= s_acquire;
99           else
100             State <= s_init;
101           end if;
102         when s_acquire =>
103           seedgen_run <= '1';
104           if(seedgen_done = '1') then
105             r0 := to_integer(seedgen_r0);
106             r1 := to_integer(seedgen_r1);
107             r2 := to_integer(seedgen_r2);
108             r3 := to_integer(seedgen_r3);
109             r4 := to_integer(seedgen_r4);
110             r5 := to_integer(seedgen_r5);
111             r6 := to_integer(seedgen_r6);
112             seedgen_run <= '0';
113             State <= s_calculate;
114           else
115             State <= s_acquire;
116           end if;
117         when s_calculate =>
118           DONE <= '0';
119           r7 := r0 + r4; -- calculate
120           r0 := r1; -- and shift
121           r1 := r2;
122           r2 := r3;
123           r3 := r4;
124           r4 := r5;
125           r5 := r6;
126           r6 := r7;
127           CURR <= to_unsigned(r7, 16);
128           State <= s_done;
129         when s_done =>
130           DONE <= '1';
131           if(RUN = '1') then
132             State <= s_calculate;
133           else
134             State <= s_done;
135           end if;
136         when others =>
137           State <= s_init;
138       end case;
139     end if;
140   end process;
141 end Behavioral;

```

Figure 3: LFG.vhd

This is the part where we generate our random number using another FSM. We take 7 seeds that was generated from the seed generator and LCG then we shift and apply the following formula for the 7th term: $R_t = R_{t-j} + R_{t-k} \pmod{M}$ where $0 < j < k$, $j = 3$, $k = 7$, $M = 2^{13}$. We didn't apply the modulo yet as we will apply it in the Main module of the code. The rest of the code is self-explanatory.

Binary Coded Decimal Converter

```

32 entity FourDigits is
33   Port ( BIN : in  unsigned (0 to 12);
34         RUN  : in  STD_LOGIC;
35         CLK  : in  STD_LOGIC;
36         D1   : out unsigned (0 to 3);
37         D2   : out unsigned (0 to 3);
38         D3   : out unsigned (0 to 3);
39         D4   : out unsigned (0 to 3);
40         DONE : out  STD_LOGIC
41   );
42 end FourDigits;
43
44 architecture Behavioral of FourDigits is
45   signal current : unsigned(0 to 12) := to_unsigned(0, 13);
46   signal digit1 : integer range 0 to 9 := 0;
47   signal digit2 : integer range 0 to 9 := 0;
48   signal digit3 : integer range 0 to 9 := 0;
49   signal digit4 : integer range 0 to 9 := 0;
50
51   -- fsm begin
52   constant s_init : STD_LOGIC_VECTOR(0 to 2) := "000";
53   constant s_dig1 : STD_LOGIC_VECTOR(0 to 2) := "001";
54   constant s_dig2 : STD_LOGIC_VECTOR(0 to 2) := "010";
55   constant s_dig3 : STD_LOGIC_VECTOR(0 to 2) := "100";
56   constant s_dig4 : STD_LOGIC_VECTOR(0 to 2) := "101";
57   constant s_done : STD_LOGIC_VECTOR(0 to 2) := "110";
58   signal State : STD_LOGIC_VECTOR(0 to 2) := s_init;
59   -- fsm end
60 begin
61   process(CLK)
62   begin
63     if(rising_edge(CLK)) then
64       case State is
65         when s_init =>
66           DONE <= '0';
67           current <= unsigned(BIN);
68           digit1 <= 0;
69           digit2 <= 0;
70           digit3 <= 0;
71           digit4 <= 0;
72           State <= s_dig1;
73         when s_dig1 =>
74           if(current > 999) then
75             digit1 <= digit1 + 1;
76             current <= current - 1000;
77           end if;
78           if(current < 1000) then
79             State <= s_dig2;
80           else
81             State <= s_dig1;
82
83           when s_dig2 =>
84             if(current > 99) then
85               digit2 <= digit2 + 1;
86               current <= current - 100;
87             end if;
88             if(current < 100) then
89               State <= s_dig3;
90             else
91               State <= s_dig2;
92             end if;
93           when s_dig3 =>
94             if(current > 9) then
95               digit3 <= digit3 + 1;
96               current <= current - 10;
97             end if;
98             if(current < 10) then
99               State <= s_dig4;
100            else
101              State <= s_dig3;
102            end if;
103           when s_dig4 =>
104             if(current > 0) then
105               digit4 <= digit4 + 1;
106               current <= current - 1;
107             end if;
108             if(current = 0) then
109               State <= s_done;
110             else
111               State <= s_dig4;
112             end if;
113           when s_done =>
114             DONE <= '1';
115             -- report the digits
116             D1 <= to_unsigned(digit1, 4);
117             D2 <= to_unsigned(digit2, 4);
118             D3 <= to_unsigned(digit3, 4);
119             D4 <= to_unsigned(digit4, 4);
120             if(RUN = '1') then
121               State <= s_init;
122             else
123               State <= s_done;
124             end if;
125           when others =>
126             State <= s_init;
127           end case;
128         end if;
129       end process;
130     end Behavioral;

```

Figure 4: FourDigits.vhd

This is the part where we convert our 13-bit number that we received from the LFG to a binary coded decimal number in order to print our result to the Seven Segment Display. Since the maximum amount we can get from 2^{13} is four digits. Having four 4-bit number is enough for the BCD. It basically uses subtraction and addition to the following 4 bit-number until it reaches a certain threshold. It follows it until our number reaches 0. We need to have this converter since we need to print out 4 digits at the same time to our Seven Segment Display. The rest of the code is self-explanatory.

Seven Segment Decoder

```
32 entity SevenSegmentDecoder is
33   Port ( BCD : unsigned (0 to 3);
34         MASK : out STD_LOGIC_VECTOR (0 to 6));
35 end SevenSegmentDecoder;
36
37 architecture Behavioral of SevenSegmentDecoder is
38   signal ISZERO : STD_LOGIC;
39   signal IONE : STD_LOGIC;
40   signal ISTWO : STD_LOGIC;
41   signal ISTHREE : STD_LOGIC;
42   signal ISFOUR : STD_LOGIC;
43   signal ISFIVE : STD_LOGIC;
44   signal ISSIX : STD_LOGIC;
45   signal ISSEVEN : STD_LOGIC;
46   signal ISEIGHT : STD_LOGIC;
47   signal ISNINE : STD_LOGIC;
48 begin
49   ISZERO <= (BCD(0) xnor '0') and (BCD(1) xnor '0') and (BCD(2) xnor '0') and (BCD(3) xnor '0');
50   IONE <= (BCD(0) xnor '0') and (BCD(1) xnor '0') and (BCD(2) xnor '0') and (BCD(3) xnor '1');
51   ISTWO <= (BCD(0) xnor '0') and (BCD(1) xnor '0') and (BCD(2) xnor '1') and (BCD(3) xnor '0');
52   ISTHREE <= (BCD(0) xnor '0') and (BCD(1) xnor '0') and (BCD(2) xnor '1') and (BCD(3) xnor '1');
53   ISFOUR <= (BCD(0) xnor '0') and (BCD(1) xnor '1') and (BCD(2) xnor '0') and (BCD(3) xnor '0');
54   ISFIVE <= (BCD(0) xnor '0') and (BCD(1) xnor '1') and (BCD(2) xnor '0') and (BCD(3) xnor '1');
55   ISSIX <= (BCD(0) xnor '0') and (BCD(1) xnor '1') and (BCD(2) xnor '1') and (BCD(3) xnor '0');
56   ISSEVEN <= (BCD(0) xnor '0') and (BCD(1) xnor '1') and (BCD(2) xnor '1') and (BCD(3) xnor '1');
57   ISEIGHT <= (BCD(0) xnor '1') and (BCD(1) xnor '0') and (BCD(2) xnor '0') and (BCD(3) xnor '0');
58   ISNINE <= (BCD(0) xnor '1') and (BCD(1) xnor '0') and (BCD(2) xnor '0') and (BCD(3) xnor '1');
59   MASK(0) <= ISZERO OR IONE OR ISTWO OR ISTHREE OR ISFOUR OR ISSEVEN OR ISEIGHT OR ISNINE;
60   MASK(1) <= ISZERO OR IONE OR ISTHREE OR ISFOUR OR ISFIVE OR ISSIX OR ISSEVEN OR ISEIGHT OR ISNINE;
61   MASK(2) <= ISZERO OR ISTWO OR ISTHREE OR ISFIVE OR ISSIX OR ISEIGHT OR ISNINE;
62   MASK(3) <= ISZERO OR ISTWO OR ISSIX OR ISEIGHT;
63   MASK(4) <= ISZERO OR ISFOUR OR ISFIVE OR ISSIX OR ISEIGHT OR ISNINE;
64   MASK(5) <= ISZERO OR ISTWO OR ISTHREE OR ISFIVE OR ISSIX OR ISSEVEN OR ISEIGHT OR ISNINE;
65   MASK(6) <= ISTWO OR ISTHREE OR ISFOUR OR ISFIVE OR ISSIX OR ISEIGHT OR ISNINE;
66 end Behavioral;
```

Figure 5: SevenSegmentDecoder.vhd

This is the part where we design our Seven Segment Decoder. Basically, how it works is first figure out what the our 4-bit number represents in decimal. Then we tell which lights should turn on each anode according to what the number is. This is done by 'xnor' gates in figuring out the number in decimal and 'or' gates on which lights should turn on.

Main (I/O)

```

32 entity Main is
33   Generic( MAX : INTEGER := 2**16);
34   Port ( CLK : in  STD_LOGIC;
35         GET : in  STD_LOGIC;
36         ANODES : out STD_LOGIC_VECTOR(0 to 7);
37         SEVENSEGMENT : out STD_LOGIC_VECTOR(0 to 6)
38       );
39 end Main;
40
41 architecture Behavioral of Main is
42   -- fsm begin
43   constant s_init : STD_LOGIC_VECTOR(0 to 2) := "000";
44   constant s_start : STD_LOGIC_VECTOR(0 to 2) := "001";
45   constant s_acquire : STD_LOGIC_VECTOR(0 to 2) := "010";
46   constant s_done : STD_LOGIC_VECTOR(0 to 2) := "100";
47   constant s_wait : STD_LOGIC_VECTOR(0 to 2) := "101";
48   signal State : STD_LOGIC_VECTOR(0 to 2) := s_init;
49   -- fsm end
50
51   signal lfg_run : STD_LOGIC := '0';
52   signal lfg_done : STD_LOGIC;
53   signal lfg_curr : unsigned(0 to 15);
54
55   -- RANDOM output begin
56   signal curr_rand_int : unsigned(0 to 12);
57   -- bcd decoded version
58   signal curr_rand_d1 : unsigned(0 to 3);
59   signal curr_rand_d2 : unsigned(0 to 3);
60   signal curr_rand_d3 : unsigned(0 to 3);
61   signal curr_rand_d4 : unsigned(0 to 3);
62   -- 7-segment masks
63   signal curr_rand_d1_mask : std_logic_vector(0 to 6);
64   signal curr_rand_d2_mask : std_logic_vector(0 to 6);
65   signal curr_rand_d3_mask : std_logic_vector(0 to 6);
66   signal curr_rand_d4_mask : std_logic_vector(0 to 6);
67   -- RANDOM output end
68
69   signal bcd_decoder_run : STD_LOGIC := '0';
70   signal bcd_decoder_done : STD_LOGIC;
71
72 begin
73   FDG : entity work.FourDigits(Behavioral)
74     PORT MAP(
75       CLK => CLK,
76       BIN => curr_rand_int,
77       RUN => bcd_decoder_run,
78       DONE => bcd_decoder_done,
79       D1 => curr_rand_d1,
80       D2 => curr_rand_d2,
81       D3 => curr_rand_d3,
82       D4 => curr_rand_d4
83     );
84
85   SSD1 : entity work.SevenSegmentDecoder(Behavioral)
86     PORT MAP(
87       BCD => curr_rand_d1,
88       MASK => curr_rand_d1_mask
89     );
90
91   SSD2 : entity work.SevenSegmentDecoder(Behavioral)
92     PORT MAP(
93       BCD => curr_rand_d2,
94       MASK => curr_rand_d2_mask
95     );
96
97   SSD3 : entity work.SevenSegmentDecoder(Behavioral)
98     PORT MAP(
99       BCD => curr_rand_d3,
100      MASK => curr_rand_d3_mask
101    );
102
103   SSD4 : entity work.SevenSegmentDecoder(Behavioral)
104     PORT MAP(
105       BCD => curr_rand_d4,
106       MASK => curr_rand_d4_mask
107     );
108
109   LFG : entity work.LFG(Behavioral)
110     PORT MAP(
111       CLK => CLK,
112       RUN => lfg_run,
113       DONE => lfg_done,
114       CURR => lfg_curr
115     );
116
117   process(CLK)
118     variable curr_int : integer range 0 to 2**13;
119   begin
120     if(rising_edge(CLK)) then
121       case State is
122         when s_init =>
123           curr_rand_int <= to_unsigned(0, 13);
124           lfg_run <= '0';
125           if(GET = '0') then
126             State <= s_init;
127           else
128             State <= s_start;
129           end if;
130         when s_start =>
131           lfg_run <= '1';
132           State <= s_acquire;
133         when s_acquire =>
134           lfg_run <= '0';
135           if(lfg_done = '1') then
136             State <= s_done;
137           else
138             State <= s_acquire;
139           end if;
140         when s_done =>
141           curr_int := to_integer(lfg_curr) mod 2**13;
142           curr_rand_int <= to_unsigned(curr_int, 13);
143           if(GET = '1') then
144             State <= s_done;
145           else
146             bcd_decoder_run <= '1';
147             State <= s_wait;
148           end if;
149         when s_wait =>
150           bcd_decoder_run <= '0';
151           if(GET = '0') then
152             State <= s_wait;
153           else
154             State <= s_start;
155           end if;
156         when others =>
157           State <= s_init;
158       end case;
159     end if;
160   end process;
161
162   process(CLK)
163     variable counter : integer range 0 to max := 0;
164   begin
165     if(rising_edge(CLK)) then
166       counter := counter + 1;
167       if(counter mod max = 0) then
168         anodes <= "11111110";
169         sevensegment <= not curr_rand_d4_mask;
170       elsif(counter mod max = max/4) then
171         anodes <= "11111101";
172         sevensegment <= not curr_rand_d3_mask;
173       elsif(counter mod max = 2*max/4) then
174         anodes <= "11111011";
175         sevensegment <= not curr_rand_d2_mask;
176       elsif(counter mod max = 3*max/4) then
177         anodes <= "11110111";
178         sevensegment <= not curr_rand_d1_mask;
179       end if;
180     end if;
181   end process;
182 end Behavioral;

```

Figure 6: Main.vhd

This is the part where we combine LFG, Binary Coded Decimal Converter and our Seven Segment Decoder. We basically acquire our randomly generated number with the help of the clock and then we use our Binary Coded Decimal to convert our 13-bit number into a BCD. Then we call our Seven Segment Decoder 4 times in order to get our BCD to work in the Seven Segment Display. We then use the rising edge of the clock once again to print out those BCDs in the Seven Segment Display.

Experimental Results

We have tested our implementation on simulations before testing it on the FPGA Board. The following are the simulations on LCG, SeedGen, LFG and FourDigit respectively. Our results depends on how long we wait before pressing the button. The simulations show the correctness of how we implemented our design and it also provides the chance of testing before using the FPGA Board.

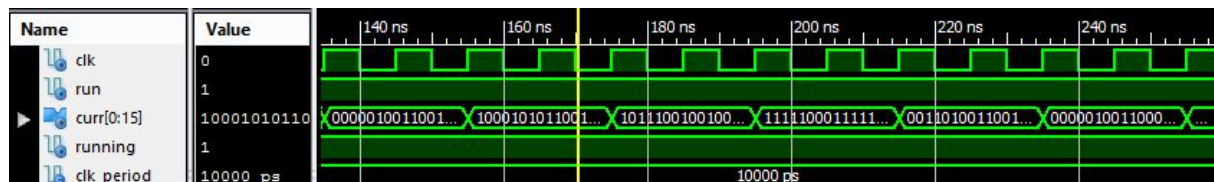


Figure 7: LCG Simulation

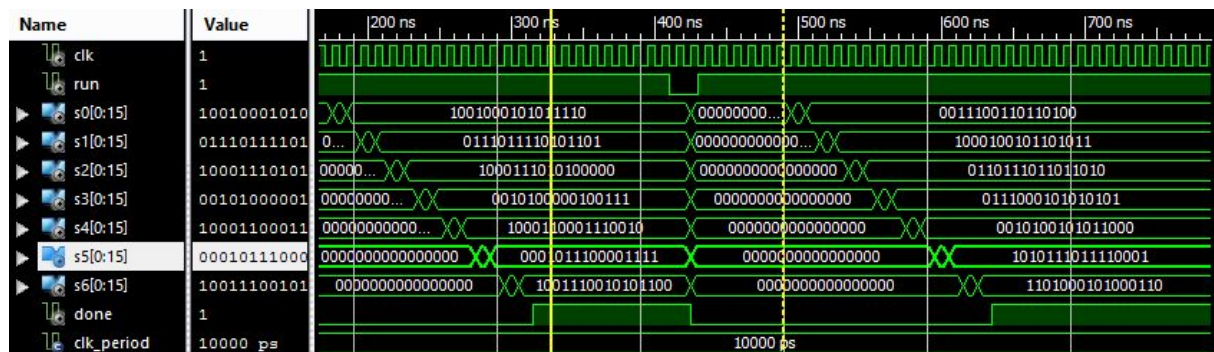


Figure 8: SeedGen Simulation

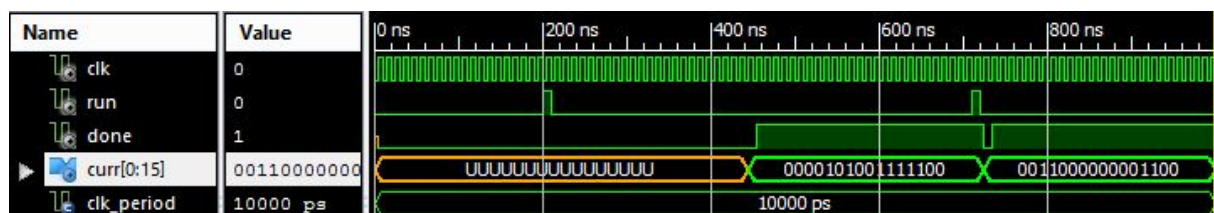


Figure 9: LFG Simulation

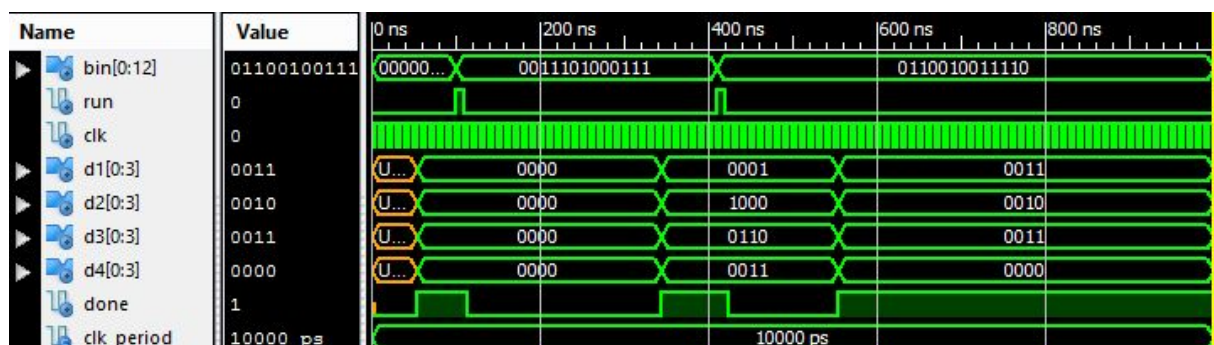


Figure 10: FourDigit Simulation

Conclusion

In this lab project, we have designed a random number generator using Linear Congruential Generator and Lagged-Fibonacci Generator. We took extra care to make sure the numbers that we are acquiring are uniform and have a high period. We initially used the simplest possible random generator (using a counter) and used it in the seed of LCG (which has low period), then we used this random generator to generate the seed of our LFG (which has both higher period and generates more uniform numbers). So, it could be said that we have gone from simple random generator to a more complex one.

We have made use of many interacting finite-state machines, which allowed us to divide our implementation into many small, easy to understand/debug modules. We did have certain difficulties while trying to make sure that the SeedGen was taking new outputs from LCG continuously, but we have solved it by adding a "Done" output to LCG and checking this output in SeedGen before acquiring the next random number. Overall, we can safely say that we have created a trustable random generator using both mathematical and digital design concepts.