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: $Rn\text{-}1*a + b \pmod{M}$, 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

```
entity LCG is
    Generic ( a : integer := 29;
              b : integer := 7;
              m : integer := 65535 -- 2^16-1
            );
    Port ( CLK : in  STD_LOGIC;
           RUN : in  STD_LOGIC;
           RUNNING : out STD_LOGIC;
           DONE : out STD_LOGIC;
           CURR : inout  unsigned (0 to 15));
end LCG;

architecture Behavioral of LCG is
    -- fsm begin
    constant s_init : STD_LOGIC_VECTOR(0 to 1) := "00";
    constant s_calculate : STD_LOGIC_VECTOR(0 to 1) := "01";
    constant s_done : STD_LOGIC_VECTOR(0 to 1) := "11";
    signal State : STD_LOGIC_VECTOR(0 to 1) := s_init;
    -- fsm end

begin
    process(CLK)
        variable curr_int : integer range 0 to m := 30;
    begin
        if(rising_edge(CLK)) then
            case State is
                when s_init =>
                    DONE <= '0';
                    RUNNING <= '0';
                    curr_int := curr_int + 1;
                    CURR <= to_unsigned(0, 16);
                    if(RUN = '1') then
                        State <= s_calculate;
                    else
                        State <= s_init;
                    end if;
                when s_calculate =>
                    DONE <= '0';
                    RUNNING <= '1';
                    curr_int := (curr_int * a + b);
                    State <= s_done;
                when s_done =>
                    DONE <= '1';
                    CURR <= to_unsigned(curr_int, 16);
                    if(RUN = '1') then
                        State <= s_calculate;
                    else
                        State <= s_init;
                    end if;
                when others =>
                    State <= s_init;
            end case;
        end if;
    end process;
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: curr_int*a + b (Mod 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.

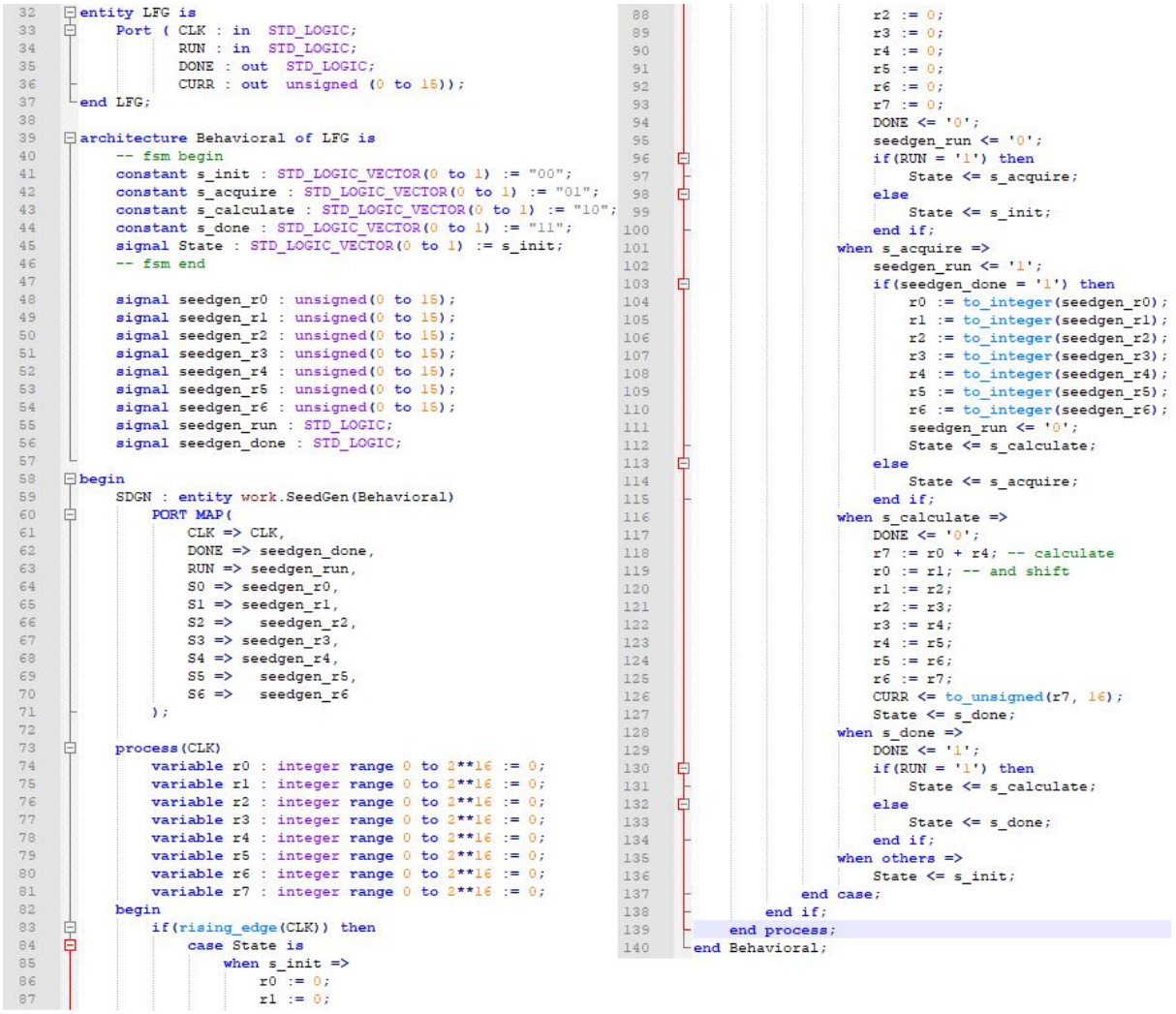
Seed Generator

```
entity SeedGen is
    Port ( CLK : in  STD_LOGIC;
           RUN : in  STD_LOGIC;
           S0 : out  unsigned (0 to 15);
           S1 : out  unsigned (0 to 15);
           S2 : out  unsigned (0 to 15);
           S3 : out  unsigned (0 to 15);
           S4 : out  unsigned (0 to 15);
           S5 : out  unsigned (0 to 15);
           S6 : out  unsigned (0 to 15);
           DONE : out  STD_LOGIC
         );
end SeedGen;

architecture Behavioral of SeedGen is
    -- fsm begin
    constant s_init : STD_LOGIC_VECTOR(0 to 3) := "0000";
    constant s_start : STD_LOGIC_VECTOR(0 to 3) := "1011";
    constant s_0 : STD_LOGIC_VECTOR(0 to 3) := "0001";
    constant s_1 : STD_LOGIC_VECTOR(0 to 3) := "0010";
    constant s_2 : STD_LOGIC_VECTOR(0 to 3) := "0011";
    constant s_3 : STD_LOGIC_VECTOR(0 to 3) := "0100";
    constant s_4 : STD_LOGIC_VECTOR(0 to 3) := "0101";
    constant s_5 : STD_LOGIC_VECTOR(0 to 3) := "0110";
    constant s_6 : STD_LOGIC_VECTOR(0 to 3) := "0111";
    constant s_done : STD_LOGIC_VECTOR(0 to 3) := "1001";
    signal State : STD_LOGIC_VECTOR(0 to 3) := s_init;
    -- fsm end

    signal lcg_curr : unsigned(0 to 15);
    signal lcg_run : STD_LOGIC := '0';
    signal lcg_running : STD_LOGIC;
    signal lcg_done : STD_LOGIC;

begin

    LCG : entity work.LCG(Behavioral)
        PORT MAP(
            CLK => CLK,
            RUN => lcg_run,
            CURR => lcg_curr,
            RUNNING => lcg_running,
            DONE => lcg_done
        );

    process(CLK)
    begin
        if(rising_edge(CLK)) then
            case State is
                when s_init =>
                    DONE <= '0';
                    lcg_run <= '0';
                    S0 <= to_unsigned(0, 16);
                    S1 <= to_unsigned(0, 16);
                    S2 <= to_unsigned(0, 16);
                    S3 <= to_unsigned(0, 16);
                    S4 <= to_unsigned(0, 16);
                    S5 <= to_unsigned(0, 16);
                    S6 <= to_unsigned(0, 16);
                    if(RUN = '1') then
                        State <= s_start;
                    else
                        State <= s_init;
                    end if;
                when s_start =>
                    DONE <= '0';
                    lcg_run <= '1';
                    if(lcg_done = '1') then
                        State <= s_0;
                    else
                        State <= s_start;
                    end if;
                when s_0 =>
                    S0 <= lcg_curr;
                    if(lcg_done = '1') then
                        State <= s_1;
                    else
                        State <= s_0;
                    end if;
                when s_1 =>
                    S1 <= lcg_curr;
                    if(lcg_done = '1') then
                        State <= s_2;
                    else
                        State <= s_1;
                    end if;
                when s_2 =>
                    S2 <= lcg_curr;
                    if(lcg_done = '1') then
                        State <= s_3;
                    else
                        State <= s_2;
                    end if;
                when s_3 =>
                    S3 <= lcg_curr;
                    if(lcg_done = '1') then
                        State <= s_4;
                    else
                        State <= s_3;
                    end if;
                when s_4 =>
                    S4 <= lcg_curr;
                    if(lcg_done = '1') then
                        State <= s_5;
                    else
                        State <= s_4;
                    end if;
                when s_5 =>
                    S5 <= lcg_curr;
                    if(lcg_done = '1') then
                        State <= s_6;
                    else
                        State <= s_5;
                    end if;
                when s_6 =>
                    S6 <= lcg_curr;
                    if(lcg_done = '1') then
                        State <= s_done;
                    else
                        State <= s_6;
                    end if;
                when s_done =>
                    DONE <= '1';
                    lcg_run <= '0';
                    if(RUN = '1') then
                        State <= s_done;
                    else
                        State <= s_init;
                    end if;
                when others =>
                    State <= s_init;
            end case;
        end if;
    end process;

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



```vhdl
entity LFG is
    Port ( CLK : in  STD_LOGIC;
           RUN : in  STD_LOGIC;
           DONE : out  STD_LOGIC;
           CURR : out  unsigned (0 to 15));
end LFG;

architecture Behavioral of LFG is
    -- fsm begin
    constant s_init : STD_LOGIC_VECTOR(0 to 1) := "00";
    constant s_acquire : STD_LOGIC_VECTOR(0 to 1) := "01";
    constant s_calculate : STD_LOGIC_VECTOR(0 to 1) := "10";
    constant s_done : STD_LOGIC_VECTOR(0 to 1) := "11";
    signal State : STD_LOGIC_VECTOR(0 to 1) := s_init;
    -- fsm end

    signal seedgen_r0 : unsigned(0 to 15);
    signal seedgen_r1 : unsigned(0 to 15);
    signal seedgen_r2 : unsigned(0 to 15);
    signal seedgen_r3 : unsigned(0 to 15);
    signal seedgen_r4 : unsigned(0 to 15);
    signal seedgen_r5 : unsigned(0 to 15);
    signal seedgen_r6 : unsigned(0 to 15);
    signal seedgen_run : STD_LOGIC;
    signal seedgen_done : STD_LOGIC;

begin
    SDGN : entity work.SeedGen(Behavioral)
        PORT MAP(
            CLK => CLK,
            DONE => seedgen_done,
            RUN => seedgen_run,
            S0 => seedgen_r0,
            S1 => seedgen_r1,
            S2 =>   seedgen_r2,
            S3 => seedgen_r3,
            S4 => seedgen_r4,
            S5 =>   seedgen_r5,
            S6 =>   seedgen_r6
        );

    process(CLK)
        variable r0 : integer range 0 to 2**16 := 0;
        variable r1 : integer range 0 to 2**16 := 0;
        variable r2 : integer range 0 to 2**16 := 0;
        variable r3 : integer range 0 to 2**16 := 0;
        variable r4 : integer range 0 to 2**16 := 0;
        variable r5 : integer range 0 to 2**16 := 0;
        variable r6 : integer range 0 to 2**16 := 0;
        variable r7 : integer range 0 to 2**16 := 0;
    begin
        if(rising_edge(CLK)) then
            case State is
                when s_init =>
                    r0 := 0;
                    r1 := 0;
                    r2 := 0;
                    r3 := 0;
                    r4 := 0;
                    r5 := 0;
                    r6 := 0;
                    r7 := 0;
                    DONE <= '0';
                    seedgen_run <= '0';
                    if(RUN = '1') then
                        State <= s_acquire;
                    else
                        State <= s_init;
                    end if;
                when s_acquire =>
                    seedgen_run <= '1';
                    if(seedgen_done = '1') then
                        r0 := to_integer(seedgen_r0);
                        r1 := to_integer(seedgen_r1);
                        r2 := to_integer(seedgen_r2);
                        r3 := to_integer(seedgen_r3);
                        r4 := to_integer(seedgen_r4);
                        r5 := to_integer(seedgen_r5);
                        r6 := to_integer(seedgen_r6);
                        seedgen_run <= '0';
                        State <= s_calculate;
                    else
                        State <= s_acquire;
                    end if;
                when s_calculate =>
                    DONE <= '0';
                    r7 := r0 + r4; -- calculate
                    r0 := r1; -- and shift
                    r1 := r2;
                    r2 := r3;
                    r3 := r4;
                    r4 := r5;
                    r5 := r6;
                    r6 := r7;
                    CURR <= to_unsigned(r7, 16);
                    State <= s_done;
                when s_done =>
                    DONE <= '1';
                    if(RUN = '1') then
                        State <= s_calculate;
                    else
                        State <= s_done;
                    end if;
                when others =>
                    State <= s_init;
            end case;
        end if;
    end process;
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

```
entity FourDigits is
    Port ( BIN : in  unsigned (0 to 12);
           RUN : in  STD_LOGIC;
           CLK : in STD_LOGIC;
           D1 : out  unsigned (0 to 3);
           D2 : out  unsigned (0 to 3);
           D3 : out  unsigned (0 to 3);
           D4 : out  unsigned (0 to 3);
           DONE : out  STD_LOGIC
           );
end FourDigits;

architecture Behavioral of FourDigits is
    signal current : unsigned(0 to 12) := to_unsigned(0, 13);
    signal digit1 : integer range 0 to 9 := 0;
    signal digit2 : integer range 0 to 9 := 0;
    signal digit3 : integer range 0 to 9 := 0;
    signal digit4 : integer range 0 to 9 := 0;

    -- fsm begin
    constant s_init : STD_LOGIC_VECTOR(0 to 2) := "000";
    constant s_dig1 : STD_LOGIC_VECTOR(0 to 2) := "001";
    constant s_dig2 : STD_LOGIC_VECTOR(0 to 2) := "010";
    constant s_dig3 : STD_LOGIC_VECTOR(0 to 2) := "100";
    constant s_dig4 : STD_LOGIC_VECTOR(0 to 2) := "101";
    constant s_done : STD_LOGIC_VECTOR(0 to 2) := "110";
    signal State : STD_LOGIC_VECTOR(0 to 2) := s_init;
    -- fsm end
begin
    process(CLK)
    begin
    if(rising_edge(CLK)) then
        case State is
            when s_init =>
                DONE <= '0';
                current <= unsigned(BIN);
                digit1 <= 0;
                digit2 <= 0;
                digit3 <= 0;
                digit4 <= 0;
                State <= s_dig1;
            when s_dig1 =>
                if(current > 999) then
                    digit1 <= digit1 + 1;
                    current <= current - 1000;
                end if;
                if(current < 1000) then
                    State <= s_dig2;
                else
                    State <= s_dig1;
                end if;
            when s_dig2 =>
                if(current > 99) then
                    digit2 <= digit2 + 1;
                    current <= current - 100;
                end if;
                if(current < 100) then
                    State <= s_dig3;
                else
                    State <= s_dig2;
                end if;
            when s_dig3 =>
                if(current > 9) then
                    digit3 <= digit3 + 1;
                    current <= current - 10;
                end if;
                if(current < 10) then
                    State <= s_dig4;
                else
                    State <= s_dig3;
                end if;
            when s_dig4 =>
                if(current > 0) then
                    digit4 <= digit4 + 1;
                    current <= current - 1;
                end if;
                if(current = 0) then
                    State <= s_done;
                else
                    State <= s_dig4;
                end if;
            when s_done =>
                DONE <= '1';
                -- report the digits
                D1 <= to_unsigned(digit1, 4);
                D2 <= to_unsigned(digit2, 4);
                D3 <= to_unsigned(digit3, 4);
                D4 <= to_unsigned(digit4, 4);
                if(RUN = '1') then
                    State <= s_init;
                else
                    State <= s_done;
                end if;
            when others =>
                State <= s_init;
            end case;
        end if;
    end process;
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

```
entity SevenSegmentDecoder is
    Port ( BCD : unsigned (0 to 3);
           MASK : out  STD_LOGIC_VECTOR (0 to 6));
end SevenSegmentDecoder;

architecture Behavioral of SevenSegmentDecoder is
    signal ISZERO : STD_LOGIC;
    signal ISONE : STD_LOGIC;
    signal ISTWO : STD_LOGIC;
    signal ISTHREE : STD_LOGIC;
    signal ISFOUR : STD_LOGIC;
    signal ISFIVE : STD_LOGIC;
    signal ISSIX : STD_LOGIC;
    signal ISSEVEN : STD_LOGIC;
    signal ISEIGHT : STD_LOGIC;
    signal ISNINE : STD_LOGIC;
begin
    ISZERO <= (BCD(0) xnor '0') and (BCD(1) xnor '0') and (BCD(2) xnor '0') and (BCD(3) xnor '0');
    ISONE <= (BCD(0) xnor '0') and (BCD(1) xnor '0') and (BCD(2) xnor '0') and (BCD(3) xnor '1');
    ISTWO <= (BCD(0) xnor '0') and (BCD(1) xnor '0') and (BCD(2) xnor '1') and (BCD(3) xnor '0');
    ISTHREE <= (BCD(0) xnor '0') and (BCD(1) xnor '0') and (BCD(2) xnor '1') and (BCD(3) xnor '1');
    ISFOUR <= (BCD(0) xnor '0') and (BCD(1) xnor '1') and (BCD(2) xnor '0') and (BCD(3) xnor '0');
    ISFIVE <= (BCD(0) xnor '0') and (BCD(1) xnor '1') and (BCD(2) xnor '0') and (BCD(3) xnor '1');
    ISSIX <= (BCD(0) xnor '0') and (BCD(1) xnor '1') and (BCD(2) xnor '1') and (BCD(3) xnor '0');
    ISSEVEN <= (BCD(0) xnor '0') and (BCD(1) xnor '1') and (BCD(2) xnor '1') and (BCD(3) xnor '1');
    ISEIGHT <= (BCD(0) xnor '1') and (BCD(1) xnor '0') and (BCD(2) xnor '0') and (BCD(3) xnor '0');
    ISNINE <= (BCD(0) xnor '1') and (BCD(1) xnor '0') and (BCD(2) xnor '0') and (BCD(3) xnor '1');
    MASK(0) <= ISZERO OR ISONE OR ISTWO OR ISTHREE OR ISFOUR OR ISSEVEN OR ISEIGHT OR ISNINE;
    MASK(1) <= ISZERO OR ISONE OR ISTHREE OR ISFOUR OR ISFIVE OR ISSIX OR ISSEVEN OR ISEIGHT OR ISNINE;
    MASK(2) <= ISZERO OR ISTWO OR ISTHREE OR ISFIVE OR ISSIX OR ISEIGHT OR ISNINE;
    MASK(3) <= ISZERO OR ISTWO OR ISSIX OR ISEIGHT;
    MASK(4) <= ISZERO OR ISFOUR OR ISFIVE OR ISSIX OR ISEIGHT OR ISNINE;
    MASK(5) <= ISZERO OR ISTWO OR ISTHREE OR ISFIVE OR ISSIX OR ISSEVEN OR ISEIGHT OR ISNINE;
    MASK(6) <= ISTWO OR ISTHREE OR ISFOUR OR ISFIVE OR ISSIX OR ISEIGHT OR ISNINE;
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)

```vhdl
entity Main is
    Generic( MAX : INTEGER := 2**16);
    Port ( CLK : in  STD_LOGIC;
           GET : in  STD_LOGIC;
           ANODES : out STD_LOGIC_VECTOR(0 to 7);
           SEVENSEGMENT : out STD_LOGIC_VECTOR(0 to 6)
           );
end Main;

architecture Behavioral of Main is
    -- fsm begin
    constant s_init : STD_LOGIC_VECTOR(0 to 2) := "000";
    constant s_start : STD_LOGIC_VECTOR(0 to 2) := "001";
    constant s_acquire : STD_LOGIC_VECTOR(0 to 2) := "010";
    constant s_done : STD_LOGIC_VECTOR(0 to 2) := "100";
    constant s_wait : STD_LOGIC_VECTOR(0 to 2) := "101";
    signal State : STD_LOGIC_VECTOR(0 to 2) := s_init;
    -- fsm end

    signal lfg_run : STD_LOGIC := '0';
    signal lfg_done : STD_LOGIC;
    signal lfg_curr : unsigned(0 to 15);

    -- RANDOM output begin
    signal curr_rand_int : unsigned(0 to 12);
    -- bcd decoded version
    signal curr_rand_d1 : unsigned(0 to 3);
    signal curr_rand_d2 : unsigned(0 to 3);
    signal curr_rand_d3 : unsigned(0 to 3);
    signal curr_rand_d4 : unsigned(0 to 3);
    -- 7-segment masks
    signal curr_rand_d1_mask : std_logic_vector(0 to 6);
    signal curr_rand_d2_mask : std_logic_vector(0 to 6);
    signal curr_rand_d3_mask : std_logic_vector(0 to 6);
    signal curr_rand_d4_mask : std_logic_vector(0 to 6);
    -- RANDOM output end

    signal bcd_decoder_run : STD_LOGIC := '0';
    signal bcd_decoder_done : STD_LOGIC;

begin

    FDG : entity work.FourDigits(Behavioral)
        PORT MAP(
            CLK => CLK,
            BIN => curr_rand_int,
            RUN => bcd_decoder_run,
            DONE => bcd_decoder_done,
            D1 => curr_rand_d1,
            D2 => curr_rand_d2,
            D3 => curr_rand_d3,
            D4 => curr_rand_d4
        );

    SSD1 : entity work.SevenSegmentDecoder(Behavioral)
        PORT MAP(
            BCD => curr_rand_d1,
            MASK => curr_rand_d1_mask
        );

    SSD2 : entity work.SevenSegmentDecoder(Behavioral)
        PORT MAP(
            BCD => curr_rand_d2,
            MASK => curr_rand_d2_mask
        );

    SSD3 : entity work.SevenSegmentDecoder(Behavioral)
        PORT MAP(
            BCD => curr_rand_d3,
            MASK => curr_rand_d3_mask
        );

    SSD4 : entity work.SevenSegmentDecoder(Behavioral)
        PORT MAP(
            BCD => curr_rand_d4,
            MASK => curr_rand_d4_mask
        );

    LFG : entity work.LFG(Behavioral)
        PORT MAP(
            CLK => CLK,
            RUN => lfg_run,
            DONE => lfg_done,
            CURR => lfg_curr
        );

    process(CLK)
        variable curr_int : integer range 0 to 2**13;
    begin
        if(rising_edge(CLK)) then
            case State is
                when s_init =>
                    curr_rand_int <= to_unsigned(0, 13);
                    lfg_run <= '0';
                    if(GET = '0') then
                        State <= s_init;
                    else
                        State <= s_start;
                    end if;
                when s_start =>
                    lfg_run <= '1';
                    State <= s_acquire;
                when s_acquire =>
                    lfg_run <= '0';
                    if(lfg_done = '1') then
                        State <= s_done;
                    else
                        State <= s_acquire;
                    end if;
                when s_done =>
                    curr_int := to_integer(lfg_curr) mod 2**13;
                    curr_rand_int <= to_unsigned(curr_int, 13);
                    if(GET = '1') then
                        State <= s_done;
                    else
                        bcd_decoder_run <= '1';
                        State <= s_wait;
                    end if;
                when s_wait =>
                    bcd_decoder_run <= '0';
                    if(GET = '0') then
                        State <= s_wait;
                    else
                        State <= s_start;
                    end if;
                when others =>
                    State <= s_init;
            end case;
        end if;
    end process;

    process(CLK)
            variable counter : integer range 0 to max := 0;
        begin
            if(rising_edge(CLK)) then
                counter := counter + 1;
                if(counter mod max = 0) then
                    anodes <= "11111110";
                    sevensegment <= not curr_rand_d4_mask;
                elsif(counter mod max = max/4) then
                    anodes <= "11111101";
                    sevensegment <= not curr_rand_d3_mask;
                elsif(counter mod max = 2*max/4) then
                    anodes <= "11111011";
                    sevensegment <= not curr_rand_d2_mask;
                elsif(counter mod max = 3*max/4) then
                    anodes <= "11110111";
                    sevensegment <= not curr_rand_d1_mask;
                end if;
            end if;
        end process;
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.

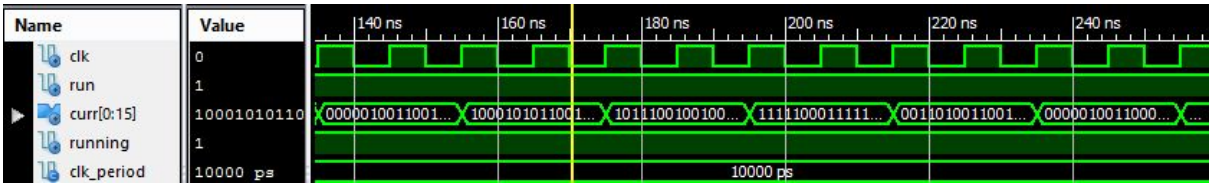


Figure 7: LCG Simulation

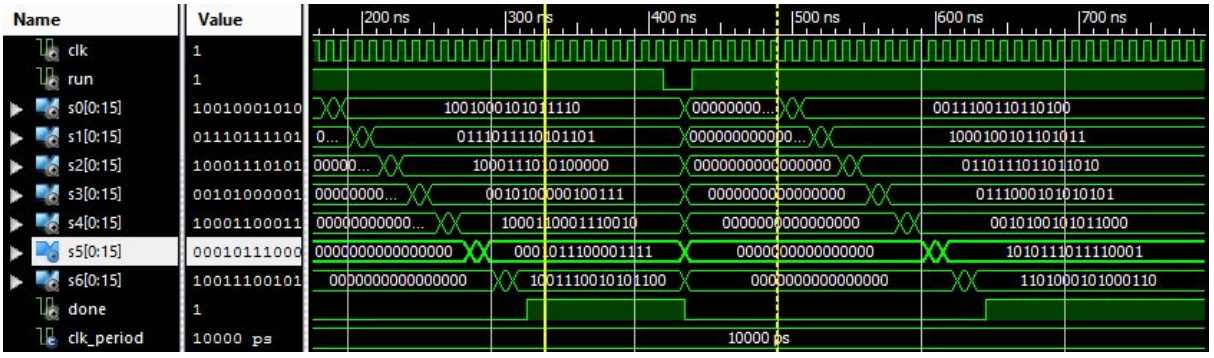


Figure 8: SeedGen Simulation

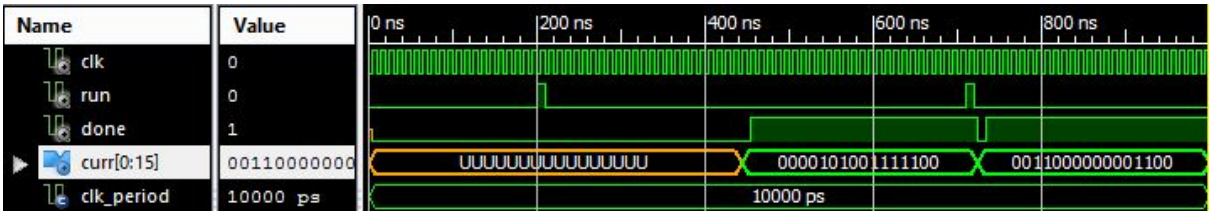


Figure 9: LFG Simulation

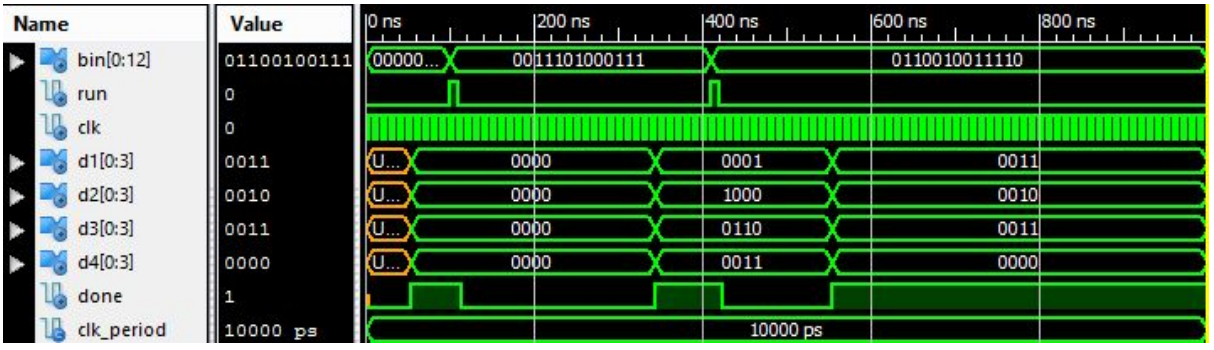


Figure 10: FourDigit Simulation

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.