FPGA Laboratory II

©2015 Kael HANSON
University of Wisconsin - Department of Physics
PHYS623 - Electronic Aids to Measurement

1 LED Snow: Simulation with ModelSim

1.1 Objectives

There is one tool you *should* still learn in firmware development: the simulator. Altera's Quartus contains a limited version of Mentor Graphics' simulator, ModelSim. We cover an advanced example of random number generation in hardware with the side-effect of illustrating the use of ModelSim. Covered in this exercise:

- Synthesizable generation of random bits using the Linear Feedback Shift Register techinique;
- Synthesizable generation of gaussian random numbers - how to create and fill ROM structures;
- Illustration of fixed-point arithmetic in hardware;
- Functional simulation with ModelSim;
- Visualization of simulation waveforms, including real-valued waveforms, *i.e.*, analog.
- Automated verification of designs via the VHDL `assert` facility.

1.2 LFSR Uniform Random Number Generation

A (Fibonacci) linear feedback shift register operates by shifting bits into the SR. The bits shifted in are derived by tapping elements in the middle of the shift register and feeding them to XORs (Figure 1). By properly choosing the taps, the LFSR can generate 0 and 1 bits with approximately equal probability and with the maximal length of sequence 2^m where m is the length of the register. By taking slices of the register, pseudorandom uniform integers can be generated. We will use this as the starting block for random number generation. The VHDL source listing is given in Section 3.2.

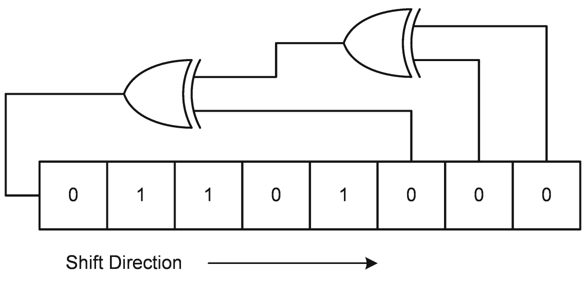


Figure 1: 8-bit linear feedback shift register with three taps.

1.3 Box-Muller Generation of Normal Random Numbers

Given 2 uniform random variables in the interval $(0, 1)$ u and v from the LFSR, they can be transformed into normal variables via the equations

$$z_1 = \sqrt{-2 \ln u} \cos 2\pi v \quad (1)$$
$$z_2 = \sqrt{-2 \ln u} \sin 2\pi v \quad (2)$$

The variables z_1 and z_2 are normally distributed with mean 0 and unit standard deviation.

1.4 Fixed Point Representation

We have uniform random numbers from the LFSR and a method to transform them into normally distributed numbers. Real number math is needed however. Without resorting to floating point math IP cores, we can represent numbers with limited precision using fixed point math. Bit vectors are simply interpreted differently. For example, if I want to represent real numbers from 0 to 10 and I know that I just need precsion to 0.01 then I can use unsigned 10-bit numbers and carry implicitly the scale factor of 1024. That is to say that a 10-bit binary number A represents the number:

$$r_A = \frac{A}{2^{10}}$$

Indeed, I can represent numbers up to 10.23 with this method. For example, the number 9.44 would be represented by the 10 binary bits `1110110000`. One can show that addition and subtraction work trivially just carrying along the implicit scale. Multiplication and division must be handled carefully propagating the scale factor.

Examining the `gaussian` entity (section 3.3), you can see that it contains two 512-element look-up ROM structures:

- One called `sine` (line 41) holding the values of the $\sin(2\pi v)$ part of the Box-Muller transform;
- One called `ln` (line 42) holding the values of the $\sqrt{-2 \ln u}$ part.

Thus, computing the value of the two halves of the transform has been reduced to looking up values in tables. As u and v are restricted by definition to the interval $(0, 1)$ it is trivial to map these onto 512 address values: the address i represents the real value

$$u_i = \frac{i + 0.5}{512}$$

The values at each address location in the ROMs are 9-bit values. The odd choice of 9-bits was driven by the Cyclone IV multiplier hardware - the DSP blocks contain 9×9 or 18×18 multipliers. The largest value that the sines and cosines can take is ± 1 so one could efficiently compress this range into a signed 9-bit integer (range -256 to +255) by multiplying by 255 Multiplying by a power of two will make things much easier for the hardware so we have to unfortunately throw away a bunch of dynamic range and use 128 as the largest possible multiplier.

The square root logarithm table is a bit more complex: the largest value in that table occurs at address 0 where the number being represented is $0.5/512 = 0.000977$. $\sqrt{-2 \ln 0.000977} = 3.7233$, thus the numbers range this 9-bit table must span is $(0, 3.7233)$. A scale factor of 64 works here.

To check that you are still following me I list a few values of the two ROMs used in the Box-Muller hardware transformer (Table 1).

1.5 Pre-Lab Questions

Prior to coming to laboratory examine the `lfsr.vhdl` and `gaussian.vhdl` code listings given in Section 3.

i	u_i	$\sqrt{-2\ln u_i}$	LN ROM	$\sin(2\pi u_i)$	SINE ROM
0	0.0010	3.7233	238	0.0061	0
1	0.0029	3.4155	218	0.0184	2
2	0.0049	3.2625	208	0.0307	3
3	0.0068	3.1577	202	0.0429	5
...	...	...	...	...	...
45	0.0889	2.2003	140	0.5298	67
...	...	...	...	...	...
509	0.9951	0.0989	6	-0.0307	508
510	0.9971	0.0766	4	-0.0184	509
511	0.9990	0.0442	2	-0.0061	511

Table 1: Some elements of the lookup ROMs used in the firmware normal random number generator. Column 1 is the address, 2 is the number in the interval $(0, 1)$ that it represents. Column 3 is the real value of the first part of the Box-Muller transform. Column 4 is the integer representation. Columns 5 and 6 are the real and integer versions of the sine portion of Box-Muller.

1.5.1 Question A:

Fill in row 200 of the Table 1, above.

1.5.2 Question B:

What is the next state in the linear feedback shift register's sequence if the current state is (MSB on the left)

`0010 1010 1110 0100 1101 0001 1100 0011`?

Assume that the XOR taps are taken at 1, 5, 18, and 30 as they are in the `led_snow` driver.

1.5.3 Question C:

The `n1` and `n2` signals output by the `gaussian` entity are 18-bit values. Given the explanation of the fixed point representations in section 1.4 above, how can you interpret the 18-bit numbers which the entity gives as real-valued normal variates? To be explicit, how would you interpret the number `1110 0010 0111 1011` as a real number if it were produced by the entity? Note that this is a signed number so you must interpret it with 2's complement notation in mind.

1.6 Step-by-Step Instructions

Download the LED Snow Quartus project from:

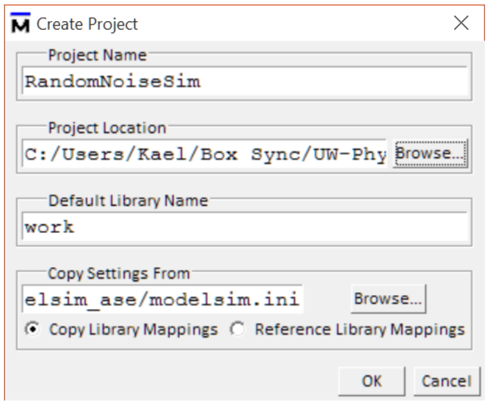
`https://www.physics.wisc.edu/undergrads/courses/fall2015/623/fpga-labs/led-snow-3.qar`

and unpack it into a directory of your choosing. You should have 5 files containing the design:

`lfsr.vhdl` Linear feedback shift register design. It implements an m-bit register with 4 XOR taps, all generic parameters;

`gaussian.vhdl` A combinational logic module which provides, on output, normal random variables given two uniform random variables from the LFSR. It uses the *Box-Muller* method. This entity *is* synthesizable and uses look-up tables to hold fixed-point representations of the two real-valued transformation functions employed by Box-Muller.

`rand_pack.vhdl` A VHDL package which holds definitions for the other packages in this example. It demonstrates how packages can be used to help organize code.


Create Project

Project Name: RandomNoiseSim

Project Location: C:/Users/Kael/Box Sync/UW-Phy

Default Library Name: work

Copy Settings From: elsim_ase/modelsim.ini

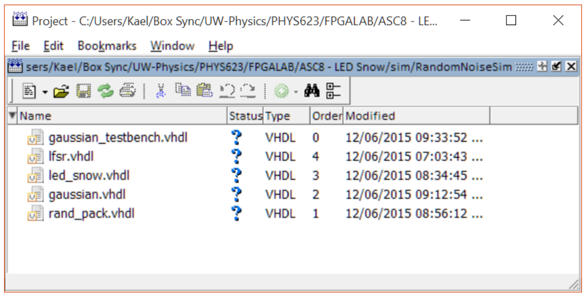
[✓ Copy Library Mappings] Reference Library Mappings

Figure 2: Dialog box filled in to create a new simulation project

`led-snow.vhdl` Top-level FPGA driver for the random generators. It contains the LFSR and the gaussian sub-modules and passes signals holding the uniform numbers from the shift register to the Box-Muller transformation. It then drives the random output of either the LFSR or the gaussian module out onto the LEDs to make a pleasing snow display (LFSR numbers) or a display based on normal random numbers.

`sim/gaussian_testbench.vhdl` A simulation driver to test the functionality of the `gaussian` entity. The simulator can be stepped and signals viewed in the wave window. Additionally, some VHDL features for automatically checking the output of the limited precision fixed point Box-Muller transform against a simulation-only implementation which uses 64-bit IEEE floating point numbers. Shows how one can use numerical math functions in VHDL simulations.

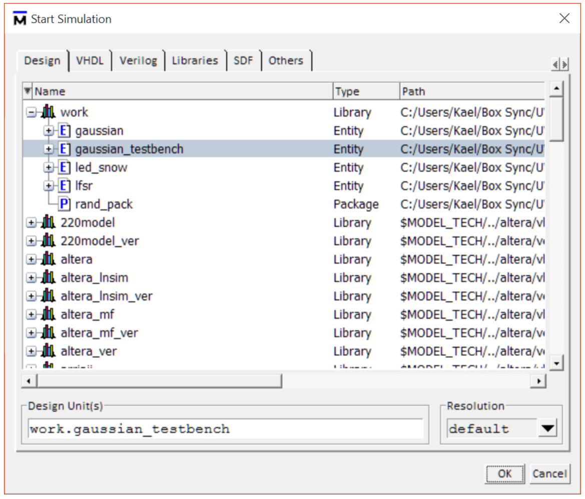
Now setup the simulation project. This happens outside of Quartus using the Altera version of ModelSim. Start the ModelSim application (e.g., search it from the Windows start menu). You will be prompted with a dialog box that contains the button, **Jumpstart**. Click that and click "Create a Project" on the following dialog. Name your simulation project and specify the project location as the `sim` subdirectory where you unpacked your Quartus archive (see Figure 2). You will see another dialog inviting you to create or add objects to the simulation project. The files are already there so click the "Add Existing File" icon and locate and add all 5 files mentioned above into your simulation project. Note that the simulation driver file `gaussian_testbench.vhdl` is in the `sim` subdirectory while the other 4 files are in the parent directory. You will need to add the files in two steps, first the file `gaussian_testbench.vhdl` by itself and then the other 4 files can be added all at once. You should see a window with the project files as in Figure 3 when you are ready to proceed to the next step. First, ensure that the `rand_pack.vhdl` file is compiled first. It contains definitions needed by the other files. Click the "Compile → Compile Order ..." menu item and either click the "AutoGenerate" button in the dialog that pops up or drag the `rand_pack.vhdl` file to the top of the list. Now, click the "Compile → Compile All" button or menu item. The question marks should turn into green checks if all is well.


Project - C:/Users/Kael/Box Sync/UW-Physics/PHYS623/FPGALAB/ASC8 - LE...

sers/Kael/Box Sync/UW-Physics/PHYS623/FPGALAB/ASC8 - LED Snow/sim/RandomNoiseSim

Name	Status	Type	Order	Modified
gaussian_testbench.vhdl	?	VHDL	0	12/06/2015 09:33:52 ...
lfsr.vhdl	?	VHDL	4	12/06/2015 07:03:43 ...
led_snow.vhdl	?	VHDL	3	12/06/2015 08:34:45 ...
gaussian.vhdl	?	VHDL	2	12/06/2015 09:12:54 ...
rand_pack.vhdl	?	VHDL	1	12/06/2015 08:56:12 ...

Figure 3: ModelSim project files.



Name	Type	Path
work	Library	C:/Users/Kael/Box Sync/U
gaussian	Entity	C:/Users/Kael/Box Sync/U
gaussian_testbench	Entity	C:/Users/Kael/Box Sync/U
led_snow	Entity	C:/Users/Kael/Box Sync/U
lfsr	Entity	C:/Users/Kael/Box Sync/U
rand_pack	Package	C:/Users/Kael/Box Sync/U
220model	Library	$MODEL_TECH/../altera/vl
220model_ver	Library	$MODEL_TECH/../altera/v
altera	Library	$MODEL_TECH/../altera/vl
altera_lnsim	Library	$MODEL_TECH/../altera/vl
altera_lnsim_ver	Library	$MODEL_TECH/../altera/v
altera_mf	Library	$MODEL_TECH/../altera/vl
altera_mf_ver	Library	$MODEL_TECH/../altera/v
altera_ver	Library	$MODEL_TECH/../altera/v

Design Unit(s): `work.gaussian_testbench`

Resolution: `default`

Figure 4: Selecting the design unit simulation driver.

Click the "Simulate → Start Simulation ..." menu item. You will get a list of all design files that ModelSim knows about (Figure 4). You must select a simulation top-level design unit. Your design files all go into the work library. Expand the '+' next to the library named work library and click gaussian_testbench, then click the "OK" button to launch the simulation.

The simulation should now start but pause before simulating anything. You will probably need to tell it to run for a longer amount of time - the default is 0.1 ns which is not long enough to observe the effects of this simulation. First, add some waves to your wave window so that you can snoop on what the simulation is doing. Drag all data objects from the `Objects` window (the slate blue one) to the wave window left gray area. These are the signals in your top level simulation driver. You can also add the process variables by activating the "Locals" window (menu item View →Locals should have a check mark next to it) and then clicking the gaussian_testbench/ver process in the structure window, the white window labeled "sim - Default" in the upper left of Figure 5). Select and drag all variables in the "Locals" windows below the "–ver–" line to the wave window too.

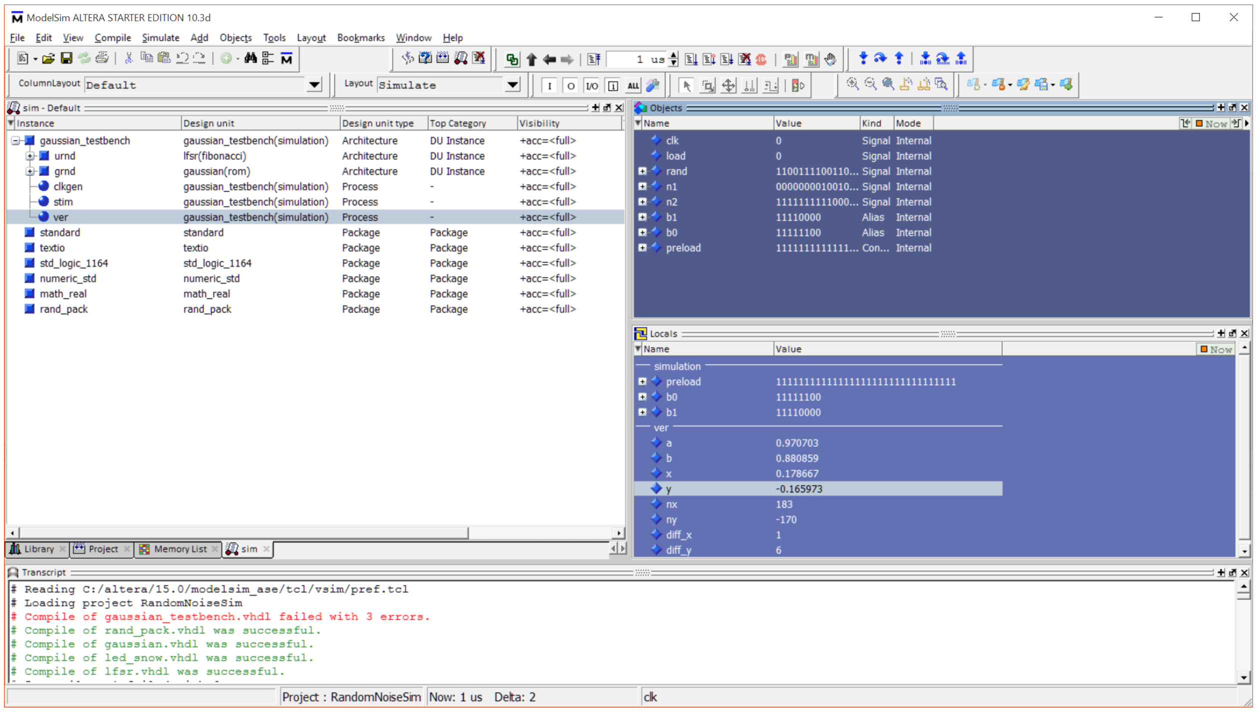
Tell ModelSim you want 20 μs of simulation time by entering "20 us" in the text box at the top where 100 ps is written (box with "1 us" in Figure 5). Run the simulation by clicking "Simulate → Run → Run 100" or typing F9. The simulator wave window will update but show you only the last few hundred ps of simulation time. Expand the window to view the full simulation by selecting the wave window and typing 'F' or use the zoom tools on the toolbar or use the "Wave → Zoom" submenu items.

ModelSim allows a pseudo-analog display of waveforms which is actually handy for this problem. Select the `n1` and `n2` waveforms and right click the mouse. In the context pop-up menu which appears select "Format → Analog (custom) ...". Set the waveform Height to 50 pixels, the Format to Analog interpolated, and the Max and Min to ±25000. Your display should now appear as in Figure 7.

1.7 Things to Do

Switch back to "Literal" view for the waveforms by right clicking them and selecting "Format →Literal". Unless you are skilled at reading binary you may also want to right click and select "Radix →Decimal" to display the waveforms n1, n2, b0, b1, nx, ny, and diff_x and diff_y as decimal numbers. Check that n1 and n2 are properly generated from the uniform integers b0 and b1 by computing the Box-Muller transform yourself for 1-2 clock cycles. Do the values differ from the values you obtain? By how much? Is this expected? The simulation driver code contains assert statements

The gaussian_testbench driver contains code to write out the signals to a text file which can be loaded into a program like Excel for analysis. The file is located in the ModelSim project directory (probably the `sim`


ModelSim ALTERA STARTER EDITION 10.3d
File Edit View Compile Simulate Add Objects Tools Layout Bookmarks Window Help
ColumnLayout Default
Layout Simulate
sim - Default
Objects
Locals
Library
Project
Memory List
sim
Transcript

Instance	Design unit	Design unit type	Top Category	Visibility
gaussian_testbench	gaussian_testbench(simulation)	Architecture	DU Instance	+acc=<full>
urnd	lfsr(fibonacci)	Architecture	DU Instance	+acc=<full>
grnd	gaussian(rom)	Architecture	DU Instance	+acc=<full>
clkgen	gaussian_testbench(simulation)	Process	-	+acc=<full>
stim	gaussian_testbench(simulation)	Process	-	+acc=<full>
ver	gaussian_testbench(simulation)	Process	-	+acc=<full>
standard	standard	Package	Package	+acc=<full>
textio	textio	Package	Package	+acc=<full>
std_logic_1164	std_logic_1164	Package	Package	+acc=<full>
numeric_std	numeric_std	Package	Package	+acc=<full>
math_real	math_real	Package	Package	+acc=<full>
rand_pack	rand_pack	Package	Package	+acc=<full>

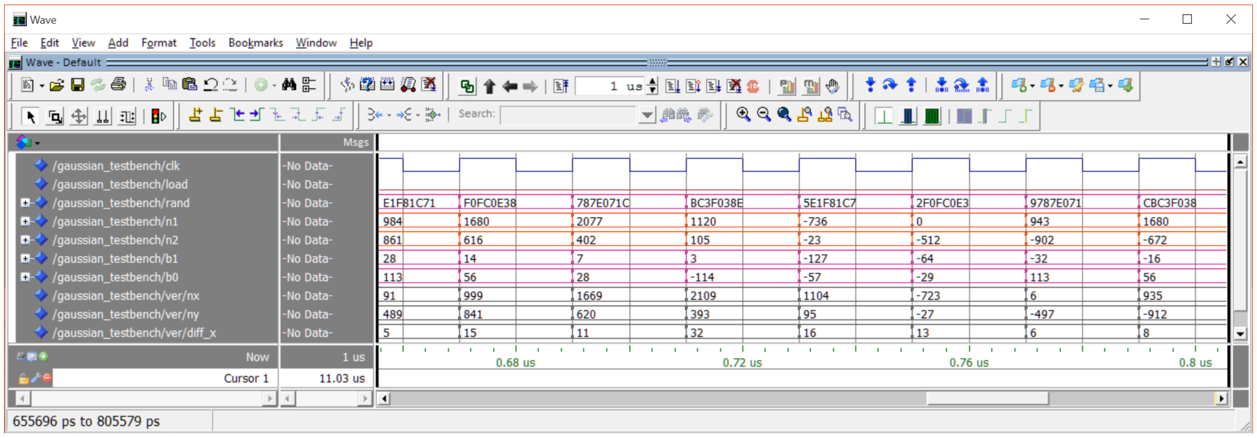
Name	Value	Kind	Mode
clk	0	Signal	Internal
load	0	Signal	Internal
rand	1100111100110...	Signal	Internal
n1	0000000010010...	Signal	Internal
n2	1111111111000...	Signal	Internal
b1	11110000	Alias	Internal
b0	11111100	Alias	Internal
preload	1111111111111...	Con...	Internal

Name	Value
simulation	
preload	11111111111111111111111111111111
b0	11111100
b1	11110000
ver	
a	0.970703
b	0.880859
x	0.178667
y	-0.165973
nx	183
ny	-170
diff_x	1
diff_y	6

```
# Reading C:/altera/15.0/modelsim_ase/tcl/vsim/pref.tcl
# Loading project RandomNoiseSim
# Compile of gaussian_testbench.vhdl failed with 3 errors.
# Compile of rand_pack.vhdl was successful.
# Compile of gaussian.vhdl was successful.
# Compile of led_snow.vhdl was successful.
# Compile of lfsr.vhdl was successful.
```

Project : RandomNoiseSim   Now: 1 us   Delta: 2   clk

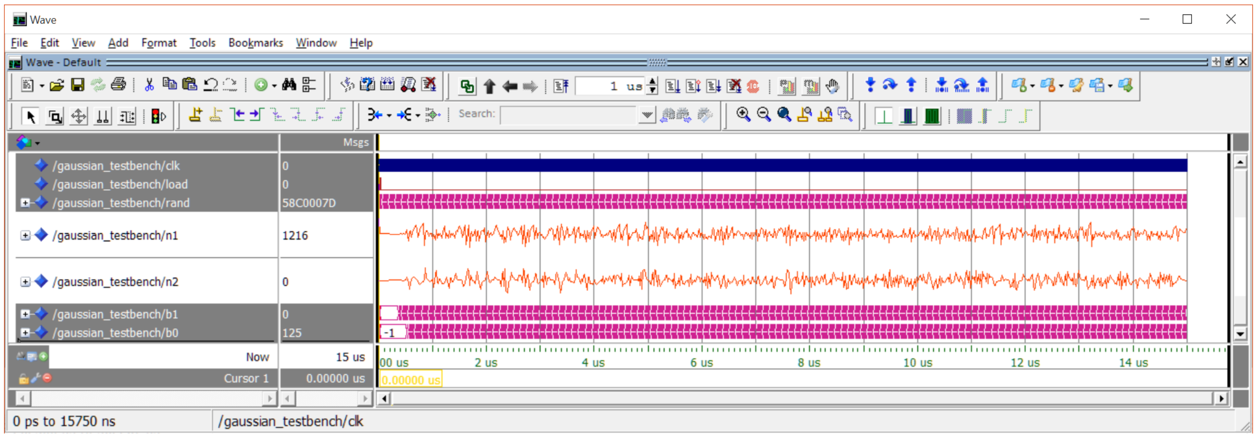
Figure 5: ModelSim workspace with elements illustrating how to access signal data objects as well as process variables.



	Msgs								
/gaussian_testbench/clk	-No Data-								
/gaussian_testbench/load	-No Data-								
/gaussian_testbench/rand	-No Data-	E1F81C71	F0FC0E38	787E071C	BC3F038E	5E1F81C7	2F0FC0E3	9787E071	CBC3F038
/gaussian_testbench/n1	-No Data-	984	1680	2077	1120	-736	0	943	1680
/gaussian_testbench/n2	-No Data-	861	616	402	105	-23	-512	-902	-672
/gaussian_testbench/b1	-No Data-	28	14	7	3	-127	-64	-32	-16
/gaussian_testbench/b0	-No Data-	113	56	28	-114	-57	-29	113	56
/gaussian_testbench/ver/nx	-No Data-	91	999	1669	2109	1104	-723	6	935
/gaussian_testbench/ver/ny	-No Data-	489	841	620	393	95	-27	-497	-912
/gaussian_testbench/ver/diff_x	-No Data-	5	15	11	32	16	13	6	8
Now	1 us								
Cursor 1	11.03 us								

655696 ps to 805579 ps

Figure 6: Basic waveform display.



```
/gaussian_testbench/clk    0
/gaussian_testbench/load   0
/gaussian_testbench/rand   58C0007D
/gaussian_testbench/n1     1216
/gaussian_testbench/n2     0
/gaussian_testbench/b1     0
/gaussian_testbench/b0     125
Now                        15 us
Cursor 1                   0.00000 us
0 ps to 15750 ns           /gaussian_testbench/clk
```

Figure 7: Modelsim analog waveform display.

directory under where you unpackaged the project archive) and it is called `random-out.txt`. Simulate some tens of microseconds and then stop the simulator. Read the file into your favorite data analysis package. Do the random variables follow their expected distributions?

1.8 Behavior on the DE0 Board

Compile the led_snow project and program the DE0. What do you see? Use the KEY0 button to switch between the snow display and the gaussian jitter displays. What else can you do with a fast hardware source of uniform and normally distributed random numbers? Can you think of good way to seed the random numbers so that the same sequence doesn't repeat itself? What about the question of repeating sequence? Can you determine, using the hardware, the length of the LFSR random sequence? Is it 2^{32}?

2 Optional Exerises

Now that you have the basic tools for FPGA development, you are encouraged to sink your teeth into a problem of your choice. Choose one of the problems below, or develop your own with the instructors.

2.1 Reading out the GSensor

An example entity which reads out the ADXL345 accelerometer and provides the X/Y acceleration values as 12-bit quantities is included in the project archive at `https://www.physics.wisc.edu/undergrads/courses/fall2015/623/fpga-labs/GSExplore.qar` Can you think of an interesting way to build on top of this?

3 Program Listings

3.1 VHDL Package

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

package rand_pack is

    -- non-synthesizable generation of normal random variates
    procedure box_muller(r1, r2 : in real; z1, z2 : out real);

    -- synthesizable block for normal random variates
    component gaussian is
        port (
            u, v    : in  std_logic_vector(8 downto 0);
            n1, n2  : out std_logic_vector(17 downto 0)
        );
    end component gaussian;

    -- synthesizable uniform random from linear feedback shift register
    component lfsr is
        generic (
            M : integer := 32;
            TAP1 : integer := 1;
            TAP2 : integer := 2;
            TAP3 : integer := 17;
            TAP4 : integer := 29
        );
        port (
            clk  : in  std_logic;
            u0   : in  std_logic_vector(M-1 downto 0);
            pre  : in  std_logic;
            u    : out std_logic_vector
        );
    end component lfsr;

end package rand_pack;

package body rand_pack is

    procedure box_muller(r1, r2 : in real; z1, z2 : out real) is
        variable x : real;
    begin
        x := sqrt(-2.0 * log(r1));
        z1 := x * cos(2.0 * MATH_PI * r2);
        z2 := x * sin(2.0 * MATH_PI * r2);
    end procedure box_muller;

end package body rand_pack;
```

3.2 Linear Feedback Shift Register

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

entity lfsr is
    generic (
        M : integer := 32;
        TAP1 : integer := 1;
        TAP2 : integer := 2;
        TAP3 : integer := 17;
        TAP4 : integer := 29
    );
    port (
        clk : in  std_logic;
        u0  : in  std_logic_vector(M-1 downto 0);
        pre : in  std_logic;
        u   : out std_logic_vector
    );
end entity lfsr;

architecture fibonacci of lfsr is
    signal utmp : std_logic_vector(31 downto 0);
begin
    u <= utmp;

    gen: process (clk)
    begin
        if rising_edge(clk) then
            if pre = '1' then
                utmp <= u0;
            else
                utmp(30 downto 0) <= utmp(31 downto 1);
                utmp(31) <= utmp(TAP1-1) xor utmp(TAP2-1) xor utmp(TAP3-1) xor utmp(TAP4-1);
            end if;
        end if;
    end process;

end architecture fibonacci;
```

3.3 Gaussian Random Number Generator

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

entity gaussian is
    port (
        u, v   : in  std_logic_vector(8 downto 0);
        n1, n2 : out std_logic_vector(17 downto 0)
    );
end entity gaussian;

architecture rom of gaussian is
    subtype word_t is signed(8 downto 0);
    type mem_t is array(0 to 511) of word_t;

    function init_logrom return mem_t is
        variable r, u : real;
                variable tmp : mem_t := (others => (others => '0'));
        begin
        for i in 0 to 511 loop
            u := (real(i)+0.5) / 512.0;
            r := sqrt(-2.0*log(u));
            tmp(i) := to_signed(integer(r*64.0), 9);
        end loop;
        return tmp;
        end init_logrom;

        function init_sinerom return mem_t is
        variable r, v : real;
                variable tmp : mem_t := (others => (others => '0'));
        begin
        for i in 0 to 511 loop
            v := (real(i)+0.5) / 512.0;
            r := sin(2.0*MATH_PI*v);
            tmp(i) := to_signed(integer(r*128.0), 9);
        end loop;
        return tmp;
        end init_sinerom;

    signal sine    : mem_t := init_sinerom;
    signal ln      : mem_t := init_logrom;
    signal x, y, z : word_t;
begin
    x <= ln(to_integer(unsigned(u)));
    y <= sine(to_integer(unsigned(v)));
    z <= sine((to_integer(unsigned(v))+128) mod 512);
    n1 <= std_logic_vector(x*z);
    n2 <= std_logic_vector(x*y);
end architecture rom;
```

3.4 Simulation Driver

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use std.textio.all;
use work.rand_pack.all;

entity gaussian_testbench is
end entity gaussian_testbench;

architecture simulation of gaussian_testbench is
    signal clk           : std_logic;
    constant preload     : std_logic_vector(31 downto 0) := x"FFFFFFFF";
    signal load          : std_logic;
    signal rand          : std_logic_vector(31 downto 0);
    alias  b0 is rand(8 downto 0);
    alias  b1 is rand(17 downto 9);
    signal n1, n2        : std_logic_vector(17 downto 0);
begin

    urnd: lfsr generic map (M=>32, TAP1=>1, TAP2=>5, TAP3=>18, TAP4=>30)
        port map (clk=>clk, pre=>load, u0=>preload, u=>rand);

    grnd: gaussian port map (u=>b0, v=>b1, n1=>n1, n2=>n2);

    clkgen: process
    begin
        clk <= '0';
        wait for 10 ns;
        clk <= '1';
        wait for 10 ns;
    end process clkgen;

    stim: process
    begin
        load <= '0', '1' after 25 ns, '0' after 50 ns;
        wait;
    end process stim;

    ver: process (clk)
        variable a, b, x, y : real;
        variable nx, ny : integer;
        variable diff_x, diff_y : natural;

        file save_file : text open write_mode is "random-out.txt";
        variable L : line;
    begin
        if rising_edge(clk) then
            a := (real(to_integer(unsigned(b0)))+0.5)/512.0;
            b := (real(to_integer(unsigned(b1)))+0.5)/512.0;
            box_muller(a, b, x, y);
            nx := integer(2.0**13 * x);
            ny := integer(2.0**13 * y);
            diff_x := abs(nx - to_integer(signed(n1)));
            diff_y := abs(ny - to_integer(signed(n2)));
            assert diff_x < 144 report "loss␣of␣precision␣in␣x:␣" & natural'image(diff_x) severity warning;
            assert diff_y < 144 report "loss␣of␣precision␣in␣y:␣" & natural'image(diff_y) severity warning;
            write(L, to_integer(signed(b0))); write(L, string'(",␣"));
            write(L, to_integer(signed(b1))); write(L, string'(",␣"));
            write(L, to_integer(signed(n1))); write(L, string'(",␣"));
            write(L, to_integer(signed(n2))); write(L, string'(",␣"));
            writeline(save_file, L);
        end if;
    end process ver;

end architecture simulation;
```

3.5 LED Snow

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.rand_pack.all;

entity led_snow is
    port (
        clk : in  std_logic;
        key : in  std_logic_vector(1 downto 0);
        led : out std_logic_vector(7 downto 0)
    );
end entity led_snow;

architecture behavioral of led_snow is
    signal rand : std_logic_vector(31 downto 0);
    signal load : std_logic;
    signal ce   : std_logic;
    constant CLK_DIV : integer := 2000000;
    alias b0 is rand(8 downto 0);   -- for readability, define alias to
    alias b1 is rand(17 downto 9);  -- low bytes of the 32-bit random
    signal n1, n2 : std_logic_vector(17 downto 0);

    -- the function below illustrates the use of
    -- signal attributes to allow unconstrained
    -- array type handling in functions
    function thresh(x : std_logic_vector; level : real) return bit is
        constant N : integer := x'length;
    begin
        if unsigned(x) < to_unsigned(integer(level * (2.0**N)), N) then
            return '1';
        else
            return '0';
        end if;
    end function thresh;

begin
    r0: lfsr generic map (M=>32, TAP1=>1, TAP2=>5, TAP3=>18, TAP4=>30)
        port map(clk=>clk, pre=>load, u0=>(others=> '1'), u=>rand);

    g0: gaussian port map (u=>b0, v=>b1, n1=>n1, n2=>n2);

    -- hold the load line high for 10 clocks
    -- 1 clock would be fine - this is to
    -- demonstrate reset generators
    rst_gen: process (clk)
        variable init : integer range 0 to 15 := 0;
    begin
        if rising_edge(clk) then
            if init < 10 then
                load <= '1';
                init := init + 1;
            else
                load <= '0';
            end if;
        end if;
    end process rst_gen;

    div: process (clk)
        variable count : integer range 0 to CLK_DIV-1 := 0;
    begin
        if rising_edge(clk) then
            if count = CLK_DIV-1 then
                count := 0;
                ce <= '1';
            else
                count := count + 1;
                ce <= '0';
            end if;
        end if;
    end process div;

    display: process (clk)
        variable iled : integer range 0 to 7;
    begin
        if rising_edge(clk) and ce = '1' then
            if key(0) = '1' then
                -- use the 8 4-bit nybbles of the 32-bit
                -- random register to determine whether
                -- the 8 LEDs should be turned on.  It
                -- looks more like snow if the fraction
                -- of bits is much less than 50%.
                for i in 0 to 7 loop
                    led(i) <= to_X01(thresh(rand(4*i+3 downto 4*i), 0.15));
                end loop;
            else
                led(7 downto 0) <= (others => '0');
                iled := to_integer(signed(n1(15 downto 13))) + 4;
                led(iled) <= '1';
            end if;
        end if;
    end process display;
end architecture behavioral;
```