COMPE 470 PROJECT REPORT

Generating Normally Distributed Random Numbers

Prof. Ken Arnold

Ethan Cua

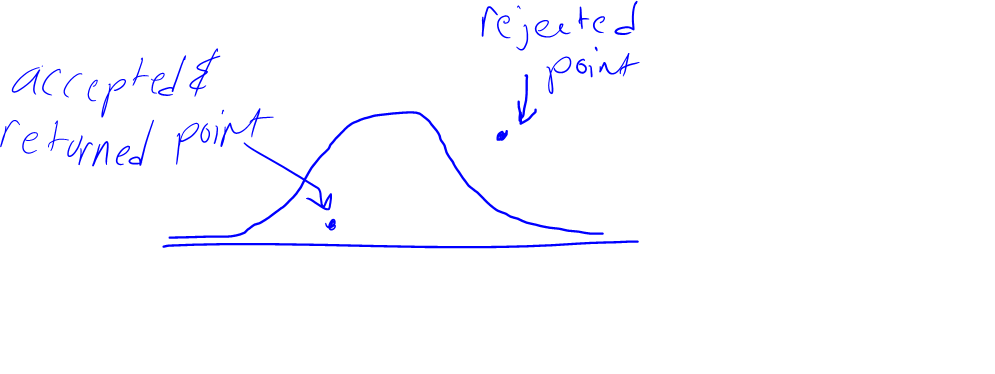
822636432

**Introduction**

The following program is designed to generate normally distributed random numbers according to the Gaussian distribution function modeled by the following equation, using a mean of 0 and a standard deviation of 1.

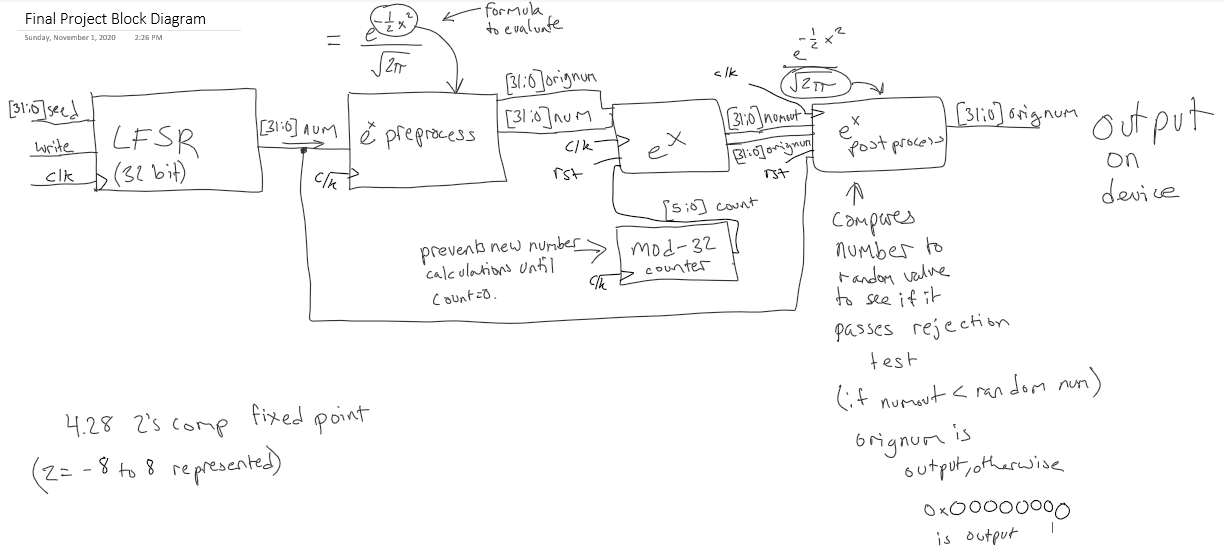
This standard distribution can be used to model other distributions with differing means and standard deviations, by using the following equation on the program’s output (z), given µ and σ of the desired distribution.

This will be done using a LFSR and an acceptance-rejection test. Each value the LFSR gives as a number will be paired, and an acceptance-rejection test will be used to create a final output. This final output, if sampled a sufficiently large number of times, should be normally distributed. This acceptance-rejection test is graphically shown below.



Effectively, points that are closer to the mean of the normal distribution are more likely to pass, as they are more likely to be paired with numbers that bring them “below” the normal curve in terms of height. Thus, the distribution of numbers returned will be roughly normal.

**Diagram and Code Description**



This is a rough diagram of the process described above. A full PDF of this diagram will be attached with this report’s submission.

The first module in this process is the LFSR, which is a simple 32-bit shift register that can be written to. The numbers interpreted from the module in the rest of the program will be in a 4.28 fixed-point notation (4 bits before the mantissa, 28 bits after), with one exception discussed later. Disabling this module will effectively disable the overall module, as no new numbers will be pushed through the pipeline. To disable the module, simply write 0x00000000 into the state register using the seed bus. This module feeds pseudo-random numbers to the rest of the module. The taps for this LFSR are in bits 31, 29, 26, and 25, with the LFSR being shifted left and the new bit being inserted at bit position 0 (the LSB).

The next module to the right is a preprocessing module, which takes the LFSR output and modifies it so it can be passed to the module calculating ex. This modification entails squaring the number (multiplying it by itself), and then dividing it by 2 (a simple ASR). One of the quirks of the module is in that preventing the loss of information, the fixed-point format of this modified number is 8.24. This, in comparison to the general use of 4.28 fixed point notation is strange, but the bits before the mantissa are significant to the operation, and thus they must be kept, along with their place value. This module also outputs the original, unmodified number so it can be carried through and output if the rejection test is passed later in the module.

There is a mod-32 counter in this module, and this serves to stall the input to the ex module as it calculates the exponential of the argument passed to it, which requires 31 different multiply instructions from a lookup table containing precalculated values of ex. These LUT values are also accessed based on input from the mod-32 counter as well as bits of the original argument passed. This module ultimately outputs the original number and the value of e.5x^2, to be used in the final module.

The final module modifies the output of the previous one slightly before finally evaluating the output number against another random number. This modification is simply to adjust the number from the previous module to match the gaussian distribution. It is done by multiplying the number by a constant of

It should be noted that both numbers are evaluated as unsigned numbers. If the random number from the LFSR is less than the number received from the previous module, the rejection test has been passed and the original number is finally sent to the output of the whole module. Effectively, the larger the number is (on output from ex module) the higher the chance it has of passing and being output. If the rejection test is failed, 0x00000000 is output. This rejection test is performed 32 times on each number from the previous module (due to delays in changing input), and one oddity of the program is that it can output the same number more than once per overall cycle because of this.

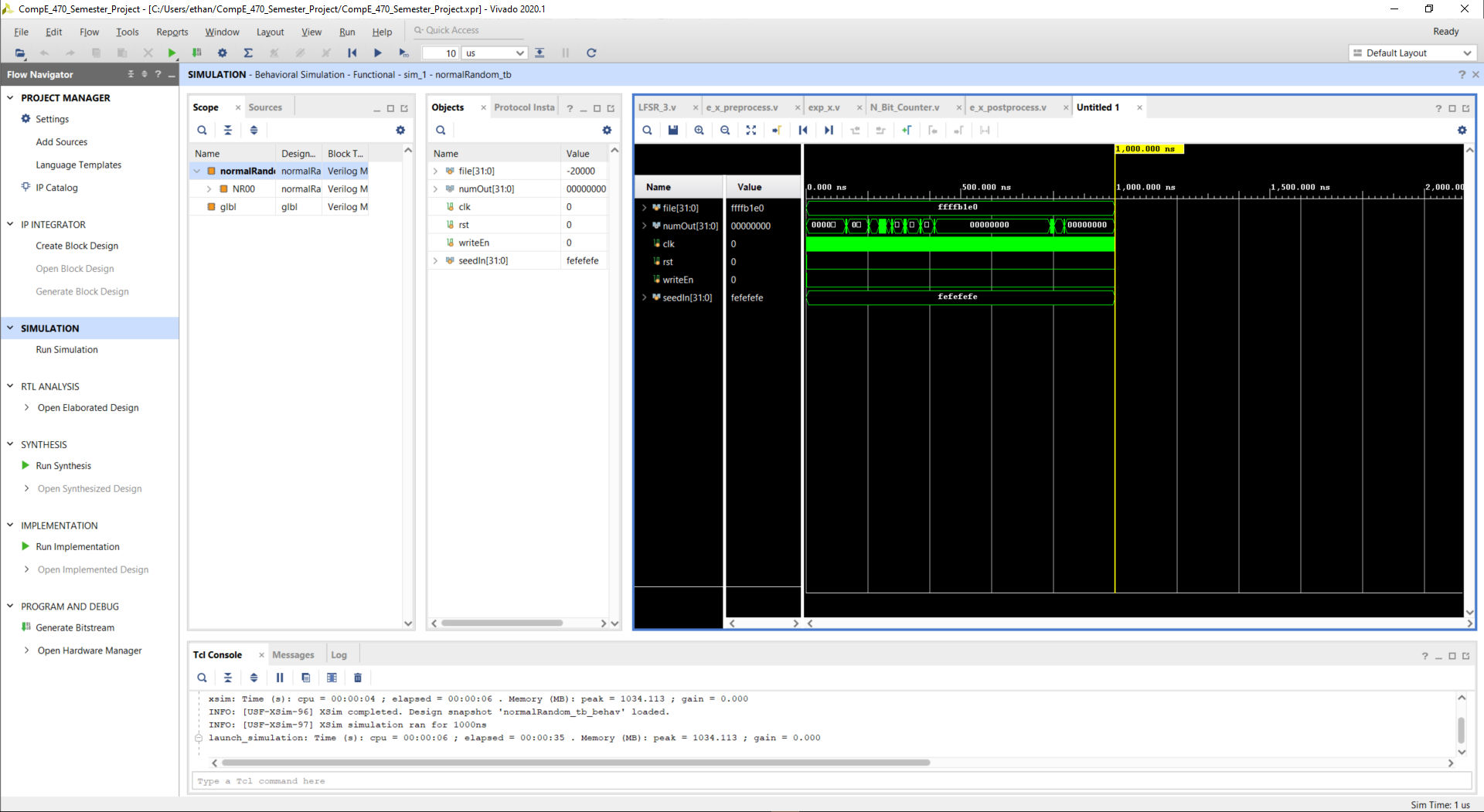
**Toolset used**

* Xilinx Vivado
* MATLAB
* Microsoft Excel

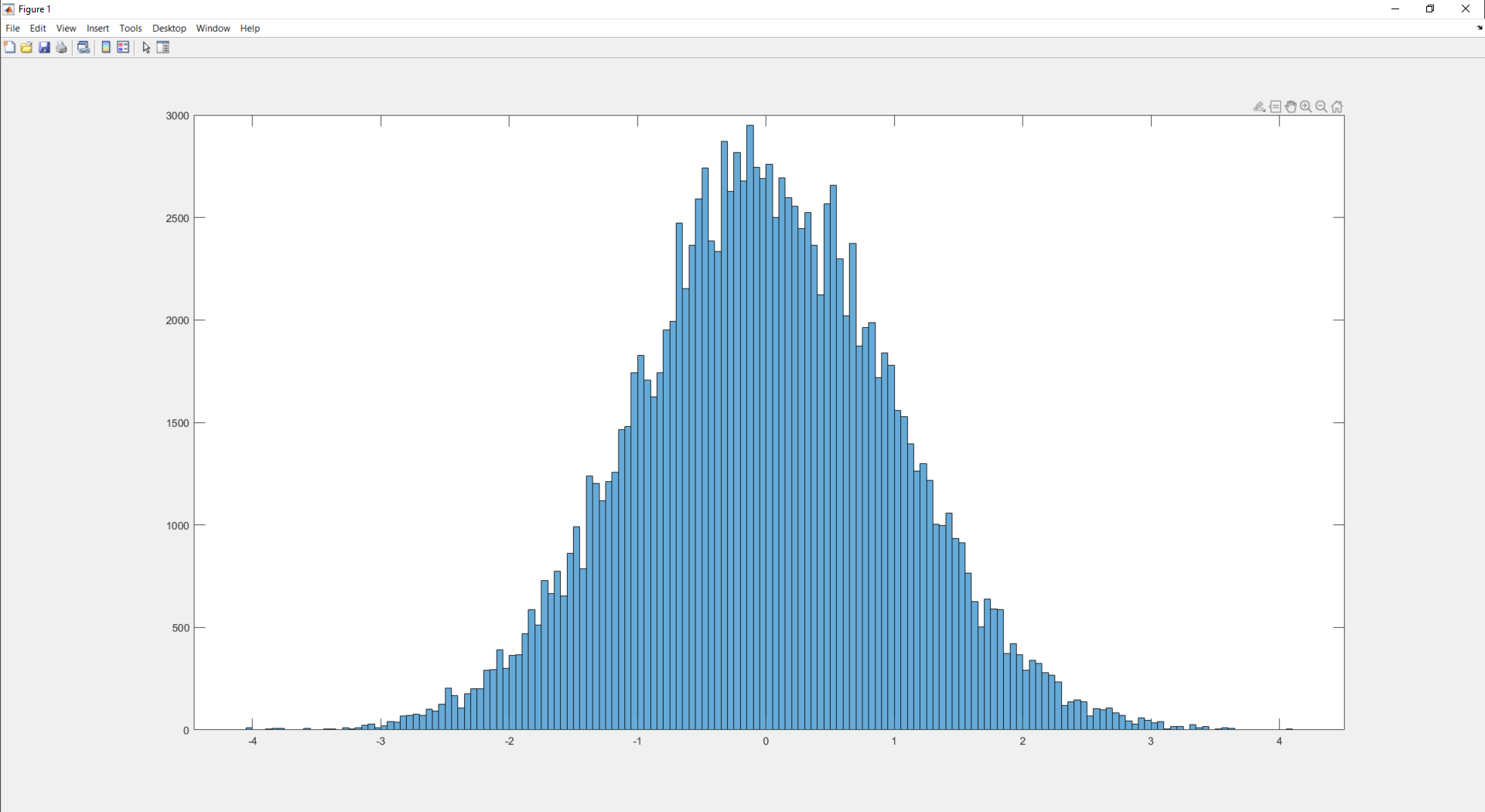
**Other notes**

* The output of the project is ultimately a 4.28 signed fixed point number, and thus still needs to be converted to a standard floating-point number for accurate interpretation.
* Nonzero outputs are written to testNumbers.txt, you will need to do the following to plot them.
  + Change the directory in the randomNormal\_tb.v file to one of your choosing with an empty testNumbers.txt file contained within.
  + Change the file extension of testNumbers.txt to .csv, for ease of reading and plotting.

**Test results in Vivado and Matlab**



This is the testbench for the project in Vivado. The seed used for this testbench is 0xfefefefe. Note the high clock rate, used to obtain as many samples as possible in as little time as possible. Also note the brief changes from 0 to other numbers in the plot, which represent the output of a valid number. This simulation was left to run for a little longer (~15ms in simulated time, or 7.5 million clock cycles @ 500 MHz) to gather more data. This data is shown on the following histogram produced in MATLAB.



This is the histogram output from the MATLAB code used to process the module’s output. The distribution is roughly normal, with some noise breaking the distribution in areas.

**Improvements with time**

The LFSR, using a 500 MHz clock, will go through all of its states in ~8.5 simulated seconds, meaning any noise from the first cycle of states will get amplified by the cyclic nature. If left to run, the spikes in the graph would grow disproportionately, meaning the distribution is not as close to the gaussian distribution as possible. A 64-bit LFSR would be recommended in the future, to reduce noise in the distribution. The code for the LFSR is also very… strange, but has been kept due to its functionality. It could be edited for clarity in the future.

The second issue is with the constant switching on the output bus, due to a test against 32 different numbers. It is recommended to only test the number once, at a specific clock cycle defined by the mod-32 counter, to avoid this issue, so that states can be held longer on the output.

Finally, implementing proper reset and enable wires would be in order, as the inclusion of reset wires is inconsistent, and enable wires are also nonexistent through the project.

**Hours spent (estimate)**

* LFSR – 2
* E\_x\_preprocess – 1.5
* Exp\_x – 3
* N\_Bit\_Counter – 0 (taken from homework 2)
* E\_x\_postprocess – 1
* normalRandom – 1
* Data analysis with MATLAB – 1
* Fixing various bugs – 2

Total time spent: 11.5 hours

**Conclusion**

This project was a marginal success, generating a roughly normal distribution from a number generator generating values with equal likelihood in the long run. The methods used to do this are suboptimal, but ultimately functional.

**Reference websites used**

<https://www.binaryhexconverter.com/decimal-to-hex-converter>

https://www.fullchipdesign.com/readmemh.htm

<https://www.chipverify.com/verilog/verilog-file-io-operations#how-to-write-files>

<https://stackoverflow.com/questions/16468114/verilog-access-specific-bits>

<https://people.sc.fsu.edu/~jburkardt/m_src/cordic/cordic.html>

<http://www.ganssle.com/item/approximations-c-code-exponentiation-log.htm>

***END OF REPORT: PROJECT CODE FOLLOWS BELOW***

Table of Code Contents

LFSR\_3.v – 8

LFSR\_tb.v – 9

e\_x\_preprocess.v – 10 - 11

e\_x\_preprocess\_tb.v – 12 - 13

exp\_x.v – 14 - 16

exp\_x\_tb.v – 17

N\_Bit\_Counter.v – 18

e\_x\_postprocess.v – 19 - 20

e\_x\_postprocess\_tb.v – 21

normalRandom.v – 22 - 23

normalRandom\_tb.v – 24 – 25

randomNormalAnalysis.m – 26

LFSR\_3.v

`timescale 1ns / 100ps

module LFSR\_3(

output [31:0] numout,

input [31:0] seed,

input write,

input clk

);

reg [31:0] numout;

reg [31:0] state;

reg xoroutcome; //legacy variable from previous iterations, I left it here because I'm lazy

always @(posedge clk)

begin

numout[31:1] = state[30:0]; //essentially performs a left shift operation without actually using the <<

xoroutcome = state[31] ^ state[29] ^ state[26] ^ state[25]; //new bit inserted using these bit 'taps'

numout[0] = xoroutcome;

if(write == 1'b1) state = seed; //if the write signal is high, a new seed is written in instead of using numout

else state = numout; //updates the shfit register if write is low

end

endmodule

LFSR\_tb.v

`timescale 1ns / 100ps

module LFSR\_tb();

wire [31:0] test;

reg clkSig;

reg [31:0] seedIn = 32'h12\_32\_4a\_6f;

reg writeIn = 1'b0;

initial

begin

clkSig = 1'b0;

writeIn = 1'b1;

#55 writeIn = 1'b0; seedIn = 32'h00000000;

end

always begin

#50 clkSig = ~clkSig;

end

LFSR\_3 myReg(.numout(test), .seed(seedIn), .write(writeIn), .clk(clkSig));

endmodule

e\_x\_preprocess.v

`timescale 1ns / 100ps

//this module takes a number x from the LFSR in 32 bit signed format and returns -1/2 x^2 for use in calculating e^x

//Note that inputs are interpreted in a 4.28 fixed point format in 2's complement.

module e\_x\_preprocess(

input clk,

input [31:0] x,

output reg [31:0] y,

output reg [31:0] origNum

);

always @(posedge clk) begin

//stores original number in output reg to be modified

y = x;

//Since I'm only concerned with the magnitude of the number (the sign gets carried over in origNum),

//all arguments will be turned into positive numbers. (2's comp)

if(y[31] == 1'b1) begin

y = y - 1;

y = ~y;

end

//square result

y = {{32'd0, y} \* y} >> 32;//A trick I got from https://stackoverflow.com/questions/16468114/verilog-access-specific-bits

//gets the 32 msb's of the result to be put into y, rather than the 32 lsb's

//NOTE: while the original number is given in 4.28 fixed point, the number squared is now in 8.24 fixed point, because of the

//process of squaring it.

//finally, divide result by 2

y = y >> 1;

origNum = x;

end

endmodule

e\_x\_preprocess\_tb.v

`timescale 1ns / 100ps

module e\_x\_preprocess\_tb();

reg clk;

reg [31:0] in;

wire [31:0] out;

e\_x\_preprocess EXP\_00(.x(in), .clk(clk), .y(out));

always begin

#10 clk = ~clk;

end

initial begin

clk = 1'b0;

in = 32'hffffffff;

#40;

in = 32'h10000000;

#40;

in = 32'h00000000;

#40;

in = 32'h08000000;

#40;

in = 32'h00000001;

#40;

$finish;

end

endmodule

exp\_x.v

`timescale 1ns / 100ps

module exp\_x(

input rst,

input clk,

input [31:0] arg,

input [4:0] num, //number from counter to obtain values from the lookup table.

input [31:0] origNum,

output reg [31:0] origNumOut,

output reg [31:0] numOut

);

//Note: arg takes the form of a 8.24 fixed point number, as I only want to generate z-scores between 0 and 8 (signed 2's comp)

//The following code creates a LUT filled with predefined values of e^x

reg [31:0] lut [30:0]; //31 values of e^-x in 4.28 fixed point approximations

reg [31:0] argState;

reg [31:0] state;

reg [31:0] origNumState;

initial begin

lut[0] = 32'h00000000; //e^-64

lut[1] = 32'h00000000;

lut[2] = 32'h0000001E;

lut[3] = 32'h00015FC2;

lut[4] = 32'h004B0556; //e^-4

lut[5] = 32'h022A5554;

lut[6] = 32'h05E2D58D;

lut[7] = 32'h09B4597E;

lut[8] = 32'h0C75F7CF;

lut[9] = 32'h0E1EB512;

lut[10] = 32'h0F07D5FD;

lut[11] = 32'h0F81FAB5;

lut[12] = 32'h0FC07F55;

lut[13] = 32'h0FE01FEA;

lut[14] = 32'h0FF007FD;

lut[15] = 32'h0FF801FF;

lut[16] = 32'h0FFC007F;

lut[17] = 32'h0FFE0020;

lut[18] = 32'h0FFF8001;

lut[19] = 32'h0FFF8001;

lut[20] = 32'h0FFFC000;

lut[21] = 32'h0FFFE000;

lut[22] = 32'h0FFFEFFF;

lut[23] = 32'h0FFFF800;

lut[24] = 32'h0FFFFBFF;

lut[25] = 32'h0FFFFE00;

lut[26] = 32'h0FFFFEFF;

lut[27] = 32'h0FFFFF7F;

lut[28] = 32'h0FFFFFC0;

lut[29] = 32'h0FFFFFE0;

lut[30] = 32'h0FFFFFEF; //e^-2^24

end

always @(posedge clk) begin

if(rst == 1'b1) begin

state = 32'h00000000;

numOut = 32'h00000000;

argState = 32'h00000000;

origNumState = 32'h00000000;

end

//before the counter resets, the finished number shows up on the output and then the new number gets shuffled in

//this is done because 31 is the only number not used by the LUT

if(num == 5'b11111) begin

numOut <= state;

state <= 32'h10000000; //this is 1 in 4.28 fixed point notation. Everything else gets multiplied to this

argState = arg;

origNumOut <= origNumState;

origNumState <= origNum;

end

else begin

if(argState[num - 1] == 1'b1) begin //the multiply only executes if there is a

state = {{32'd0, state} \* lut[31 - num]} >> 28; //accumulated multiply, gets 32 msb's (see e\_x\_preprocess for why this line looks like this)

end

end

end

endmodule

exp\_x\_tb.v

`timescale 1ns / 100ps

module exp\_x\_tb();

reg clk, rst;

reg [31:0] arg;

wire [4:0] num;

wire [31:0] out;

wire [31:0] state;

exp\_x EXP00 (.rst(rst), .clk(clk), .arg(arg), .num(num), .numOut(out), .state(state));

N\_Bit\_Counter #(.M(32), .N(5)) NBC00 (.clk(clk), .rst(rst), .number(num));

initial begin

rst = 1; clk = 0; arg = 32'h01000000; //this should (hopefully) return e^-1/2 = 32'h022A5554; after enough cycles

#6;

rst = 0;

#14;

arg = 32'h0FF0B671; //just some random number, hoping to get a sensible result after the second cycle

#750;

$finish;

end

always begin

#5;

clk = ~clk;

end

endmodule

N\_Bit\_Counter.v

`timescale 1ns / 100ps

//This is an N bit counter from Homework 2, but instead of counting up, this number counts down.

module N\_Bit\_Counter

#(

parameter M = 4,

parameter N = 2

)

(

input clk,

input rst,

output reg [N - 1:0] number

);

always @(posedge clk) begin

if(number == 0 || rst == 1'b1) begin

number = M - 1;

end

else begin

number = number - 1;

end

end

endmodule

e\_x\_postprocess.v

`timescale 1ns / 100ps

//This module takes the orignal number, its value after going through e^x module, and a random number from the LFSR

//If the argument from the e^x module is greater than the randomly generated value, then the original number is output.

module e\_x\_postprocess(

input [31:0] arg,

input [31:0] origNum,

input [31:0] testNum,

input clk,

input rst,

output [31:0] number

);

reg [31:0] number;

reg [31:0] modifiedArg;

reg [31:0] modifiedTestNum; //comes from LFSR, converted to a value between 0 and 1/sqrt(2pi), the max height of the function

always @(posedge clk) begin

//The rest of the formula normalizes the value of e^x so the area under the gaussian curve is 1. This is done by

//dividing by sqrt(2pi), which is approximately the same as multiplying by 32'h06621101

modifiedArg = {{32'd0, arg} \* 32'h06621101} >> 28;

modifiedTestNum = {{32'd0, testNum} \* 32'h06621101} >> 28;

if(rst == 1'b1) begin

number = 32'h00000000;

end

else if(arg > testNum) begin

number = origNum;

end

else begin

number = 32'h00000000;

end

end

endmodule

e\_x\_postprocess\_tb.v

`timescale 1ns / 100ps

module e\_x\_postprocess\_tb( );

reg [31:0] arg, origNum, randomNum;

wire [31:0] out;

reg rst, clk;

e\_x\_postprocess EXPP00 (.arg(arg), .origNum(origNum), .testNum(randomNum), .number(out), .clk(clk), .rst(rst));

initial begin

clk = 0; rst = 1; arg = 32'h01234567; origNum = 32'h10000000; randomNum = 32'h02000000;

#15;

rst = 0;

#30;

randomNum = 32'h00000000;

#30;

$finish;

end

always begin

clk = ~clk;

#5;

end

endmodule

normalRandom.v

`timescale 1ns / 100ps

//This is the top-level module that implements this whole process as one module.

module normalRandom(

input clk,

input rst,

input writeEnable,

input [31:0] seed,

output [31:0] number

);

wire [31:0] lfsrOut;

wire [31:0] expreOut;

wire [31:0] expreOrigOut;

wire [4:0] counterNum;

wire [31:0] origNumExpx;

wire [31:0] expxOut;

//These are instantiations of all the modules, operating in one data pipeline.

//this collection is made to simplify testing of the entire project by supplying only a few signals

LFSR\_3 LFSR00 (.seed(seed), .write(writeEnable), .clk(clk), .numout(lfsrOut));

e\_x\_preprocess EXPRE00 (.clk(clk), .x(lfsrOut), .y(expreOut), .origNum(expreOrigOut));

exp\_x EXPX00 (.clk(clk), .rst(rst), .origNum(expreOrigOut), .arg(expreOut), .num(counterNum), .origNumOut(origNumExpx), .numOut(expxOut));

N\_Bit\_Counter #(.N(5), .M(32)) NBC00 (.clk(clk), .rst(rst), .number(counterNum));

e\_x\_postprocess EXPOST00 (.clk(clk), .rst(rst), .number(number), .arg(expxOut), .origNum(origNumExpx), .testNum(lfsrOut));

endmodule

normalRandom\_tb.v

`timescale 1ns / 100ps

module normalRandom\_tb();

integer file;

wire signed [31:0] numOut;

reg clk, rst, writeEn;

reg [31:0] seedIn;

normalRandom NR00(.clk(clk), .rst(rst), .writeEnable(writeEn), .seed(seedIn), .number(numOut));

always begin

clk = ~clk;

#1;

if(numOut != 0) begin

$fwrite(file, numOut);

$fwrite(file, ", ");

end

end

initial begin

file = $fopen ("C:\\Users\\ethan\\Documents\\SDSU\\Fall 2020\\CompE 470\\Semester Project\\testNumbers.txt", "w");//file extension here.

rst = 1; clk = 0; seedIn = 32'hFEFEFEFE; writeEn = 1;

#2;

rst = 0;

writeEn = 0;

end

endmodule

randomNormalAnalysis.m

%this code is intended to read waveform data from the randomNormal module

data = readmatrix('testNumbers.csv');

dataSize = size(data, 2);

modifiedData = zeros(size(data, 2), 1);

%the matrix read from testNumbers.csv is read in integers, but are defined

%through my verilog program as 4.28 2's comp fixed point. This section

%converts each 32 bit "integer" to a floating point number.

for i = 1 : size(data, 2) - 1

temp = data(1, i);

neg = 0;

%the following section converts this number into a positive magnitude

%it also raises a neg flag for the final part of the conversion.

if(temp < 0)

temp = temp \* -1;

neg = 1;

end

%does the fixed to floating point conversion

temp2 = 0.0;

for j = 0 : 30

if(bitand(int32(temp), int32(2^j)) ~= 0)

number = 2^(j - 28);

temp2 = temp2 + number;

end

end

modifiedData(i, 1) = temp2;

if(neg == 1)

modifiedData(i,1) = modifiedData(i, 1) \* -1;

end

end

%plots the data in a histogram for verification

histogram(modifiedData);