EECE 359 – Computer Communications and Networking **Spring 2020**

Lab #5 – Error Detection Performance of CRC Codes (Part 1)
(Pseudo-Random Bit Generator)

Objectives

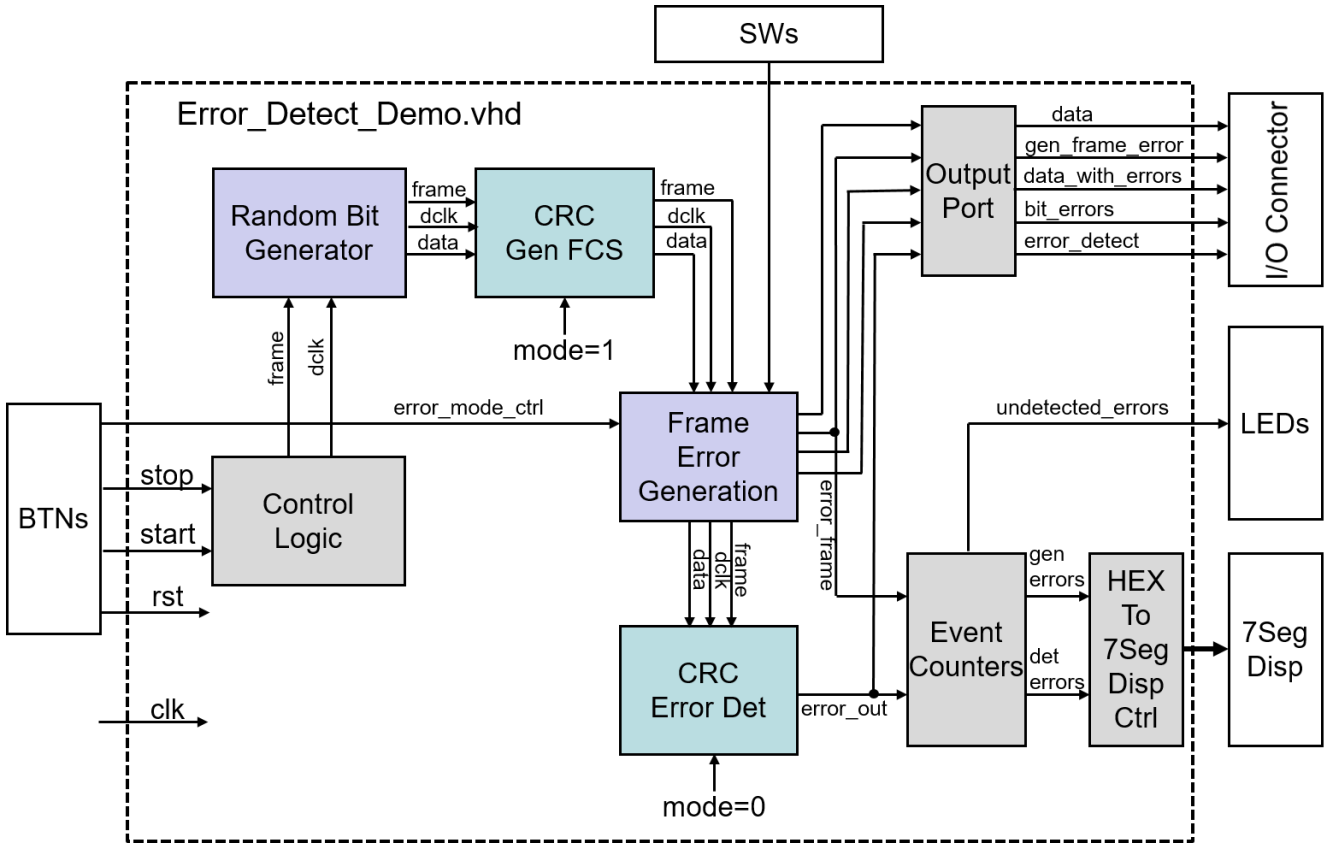
The objectives of Lab 5 and Lab 6, Error Detection Performance of CRC Codes, are

1. to develop a system for evaluating the performance of CRC error detection
2. to implement the system in hardware using the FPGA board, and
3. to use the system to study the performance of CRC error detection.

In this Part 1 of the lab exercise, you will develop a *pseudorandom bit generator* which will be used to generate data frames with randomized data for evaluating the performance of CRC error detection. It can also be used to simulate random errors in the data frame.

System Description

The block diagram below is a high-level view of the overall system.



An effective way for testing hardware functions, such as the CRC for error detection, is through the application of random data patterns. The data flow of this system begins with a Random Bit Generator, used to produce random data frames which are applied to the CRC FCS generator. The data frames that are output from the CRC FCS generator, which include

a frame check sequence appended to the end of the serial data, are passed through a Frame Error Generation block. This function simulates a transmission medium and inserts an error pattern into some of the data frames. The output of the Frame Error Generation function is then fed into the CRC Error Detection function designed to detect errors in the frames received.

Event Counters are used to count the number of frames with generated errors and the number of frames that are detected by CRC Error Detect. The counts are displayed on the 7-segment display. Two digits are used for each of the counts and they are displayed as hexadecimal numbers. The LEDs are used to display the binary difference between the two counters. Thus, it displays a count of the undetected errors.

The Control Logic is used to control the operation of the system using three of the board's momentary switches, one for a system reset, one to start operation, and one to stop operation. The board's slide switches can be used to provide some programmability of the error pattern to be injected in the data. The data frames with and without errors are brought out on external pins of the board. In addition, the signals indicating when an error frame is generated and when the CRC detects a frame with errors are brought out on pins.

Design Hierarchy/Files

For the entire evaluation system, the hierarchy will look like the following:

Test_Error_Detect_Demo – Behavior (Test_Error_Detect_Demo.vhd)

- uut – Error_Detect_Demo – Behavioral (Error_Detect_Demo.vhd)
  - HEXon7segDispA – HEXon7segDisp – Behavioral (HEXon7segDisp.vhd)
  - RST_debounce – debounce – Behavioral (debounce.vhd)
  - Start_debounce – debounce – Behavioral (debounce.vhd)
  - Stop_debounce – debounce – Behavioral (debounce.vhd)
  - Control – Control_Logic – Behavioral (Control_Logic.vhd)
  - RandBitGenA – RandBitGen – Behavioral (RandBitGen.vhd)
  - FCS_Gen – CRC – Behavioral (CRC.vhd)
  - Frame_Error – Frame_Error_Gen – Behavioral (Frame_Error_Gen.vhd)
    - RandBitGenB – RandBitGen – Behavioral(RandBitGen.vhd)
  - Error_Detect – CRC – Behavioral (CRC.vhd)
  - PMOD_JA – Output_Port – Behavioral (Output_Port.vhd)
  - Counters – Event_Counters – Behavioral (Event_Counters.vhd)
  - Lab6.ucf

For this lab exercise, the design hierarchy will look like the following:

Test_RandBitGenA – Test_RandBitGen – Behavior (Test_RandBitGen.vhd)

- RandBitGenA – RandBitGen – Behavioral (RandBitGen.vhd)

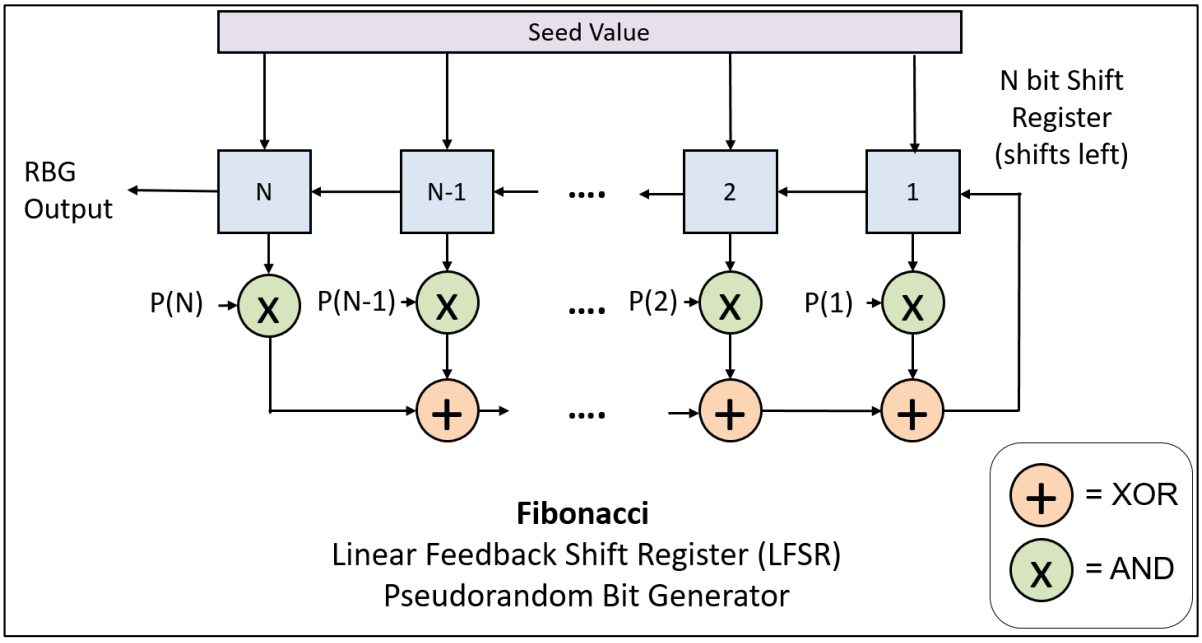
You will be developing the random bit generator (RandBitGen.vhd) and the test bench (Test_RandBitGen.vhd) for testing the random bit generator module using simulation. The RandBitGen.vhd file is provided as a template for your design.

Procedure:

- **Create a project named "Lab5" in the ISE Project Navigator**
- **Add the provided design template, RandBitGen.vhd, to your project**
- **Complete the design of the Random Bit Generator in RandBitGen.vhd**

  What we want to do is create a random data stream to feed into the FCS generator, which uses the CRC module developed in Lab4. A pseudorandom sequence can be produced using a linear feedback shift register (LFSR) and applying a primitive polynomial to define the feedback paths. It works in a manner similar to CRC generation.

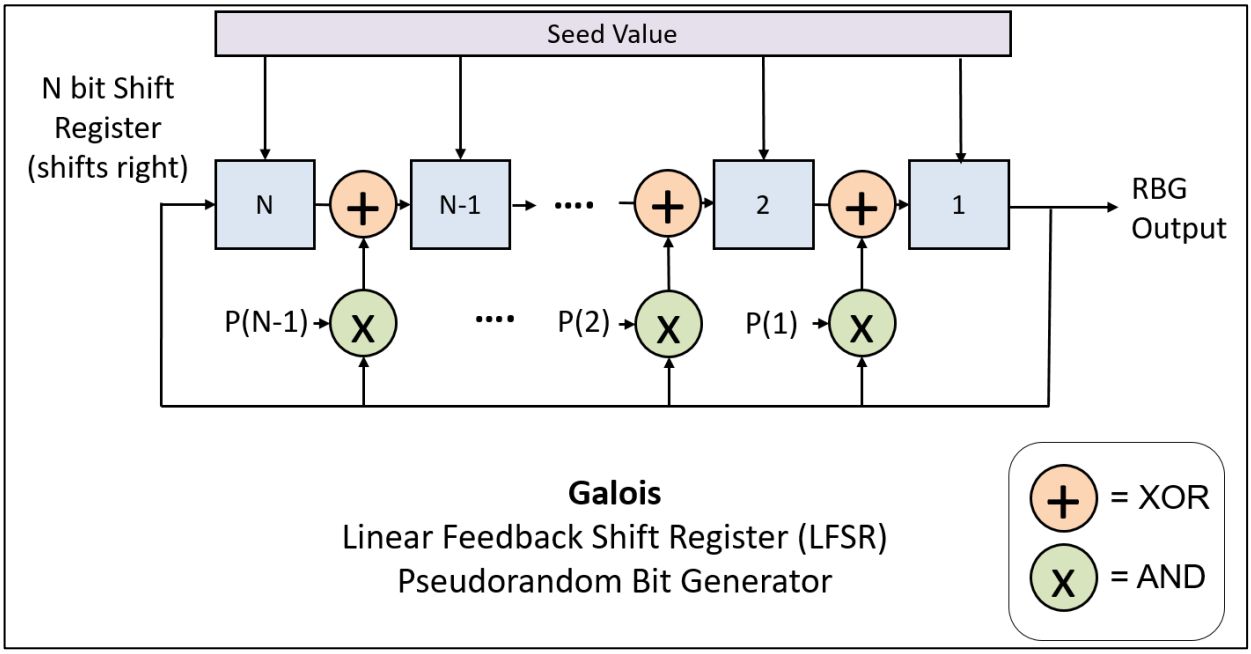
  For an implementation with an N-bit register, the sequence will cycle through all possible sequences of N bits (except all zeros) before the sequence repeats. Because the sequence repeats predictively, it is called a "pseudo" random sequence. The length of the sequence is $2^N - 1$. We will look at two different implementations. A good reference is http://en.wikipedia.org/wiki/Linear_feedback_shift_register. The first implementation is called the **Fibonacci LFSR**, shown in the figure below. It feeds back the exclusive OR of the register bits, selected according to the primitive polynomial expression, to the input of bit 1 of the shift register.



Note that the bits are numbered 1 to N so as to line up with the orders of the polynomial terms. The register shifts to the left with the feedback going into bit 1. The RBG output is taken from bit N. The shift register bits are preloaded at reset with what is called a seed value. The seed value can be any value except all zeros and defines the starting point in the sequence.

The second implementation is called the **Galois LFSR**. In this case, the shift register shifts to the right with feedback coming from bit 1 and being distributed to exclusive OR gates between the bits of the shift register. The presence of the feedback path for a particular bit is determined by the primitive polynomial term corresponding to the bit. The RBG output is taken from bit 1 of the shift register.

In either implementation, the output can really come from any bit of the register. The only difference would be the phasing of the data sequence.



A few primitive polynomials are shown in the table below.

Bits	Primitive Polynomial	Period
n		$2^n - 1$
2	$x^2 + x + 1$	3
3	$x^3 + x^2 + 1$	7
4	$x^4 + x^3 + 1$	15
5	$x^5 + x^3 + 1$	31
6	$x^6 + x^5 + 1$	63
7	$x^7 + x^6 + 1$	127
8	$x^8 + x^6 + x^5 + x^4 + 1$	255

Both the Fibonacci and Galois implementations provide optimal length sequences ($2^N - 1$), but the sequences will be different. The Galois LFSR is more like the implementation of the CRC that you did in Lab4. For your implementation of the random bit generator, choose one of the two methods.

Implement the design in VHDL using the provided template. The "size" or number of bits used for the shift register is passed as a "generic" parameter. The primitive polynomial "P' and the value of the "seed" are passed to the module as inputs. The value of P indicates

the presence of all the terms except the lowest order or 0^{th} term. That term is always 1, but is not directly used in the LFSR implementations. The values of size, P, and seed are passed to the module from the next higher level of the design hierarchy (which for this lab is the test bench).

The “frame_in” input is used to enable shifting of the register and enabling the output data. The frame_out signal should “frame” the output data as was done in the CRC design.

The dclk_in signal is an input that establishes the data rate of the random bit generator output data stream. The dclk_out signal is the data clock for the output data stream. The rising edge of dclk_out should occur at the mid-point of each data bit of data_out.

The template includes code for a counter, “shift_ctr”, which counts the number of bits generated in a frame. The counter is not required for the RBG to function, but is helpful in simulation to check the size of your frames. The counter will be optimized out during synthesis.

Notice that because of its simplicity, the RBG appears to have no FSM. But actually, the shift register of the LFSR is the state register for the FSM. The feedback of the LFSR is the next state logic.

- **Create a simulation Test Bench to test operation of the RBG behavioral**

Generate a test bench by adding a new source to the project and selecting VHDL Test Bench.

The created test bench code will not have the GENERIC clause nor its application to the vector sizes in the PORT clause of the *RandBitGen* component. Nor will it be implemented in the instantiation template. You will have to add these into the test bench code manually.

Modify the input signal declarations for “*P*” and “*seed*” to use the generic parameter.

In the test bench, set the system clock period in the *clk_process* to what your FPGA board uses.

Add a process, like the *clk_process*, to generate the *dclk_in* signal. Set its period to 32 times the system clock period and with a 50% duty cycle.

You are to run a separate simulation for each of the following parameter sets:

TEST NUMBER	RBG Size	RBG P	RBG Seed	Number of Frames	Frame Length
1	4	1100	1001	2	$2^4 - 1 = 15$
2	5	10100	10001	2	$2^5 - 1 = 31$
3	8	10111000	10000001	6	8

To make the Test Bench flexible so that it can be easily switched between the different parameter cases, add the following code to your test bench:

- Declare an integer constant named *TEST_NUMBER*, and initially set it to 1. This is the only value that will need to be changed to run a different set of parameters.
- Declare another integer constant, this one with the name *LARGEST_SIZE*, and set it to 8. We will be saving the parameter values in an array. In order to have different sizes of *P* and *Seed* values in the array, the size of its vector needs to be declared with the length of the longest one. The shorter values will have leading zeros to fill the extra bits of the vector.
- Declare a record type where the records are declarations for the parameters from the above table, given as follows:

```
type param_set is record
      RBGsize : integer;
      RBG_P : std_logic_vector(LARGEST_SIZE downto 1);
      RBGseed : std_logic_vector(LARGEST_SIZE downto 1);
      FRAME_LENGTH : integer;
      NUM_FRAMES : integer;
end record;
```

- Declare an array, of the type just defined, containing the parameter values from the table, given as follows:

```
type param_array_type is array (positive range <>) of param_set;
constant PARAM_ARRAY: param_array_type :=
      (
            (4, "00001100", "00001001", 15, 2),
            (5, "00010100", "00010001", 31, 2),
            (8, "10111000", "10000001",  8, 6)
      );
```

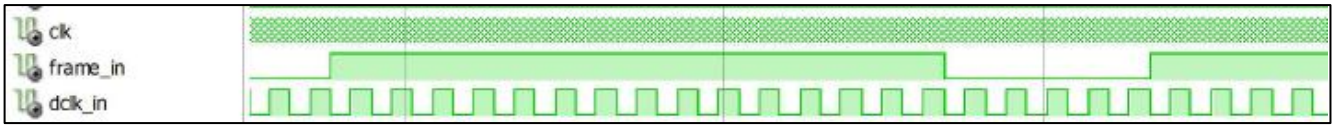
- At the beginning of the Stimulus Process, add the following code which will use the parameter array to set the values of *P* and *Seed*:

```
-- Stimulus process
stim_proc: process
begin
      P <= std_logic_vector(resize(
            unsigned(PARAM_ARRAY(TEST_NUMBER).RBG_P),
            PARAM_ARRAY(TEST_NUMBER).RBGsize));
      seed <= std_logic_vector(resize(
            unsigned(PARAM_ARRAY(TEST_NUMBER).RBGseed),
            PARAM_ARRAY(TEST_NUMBER).RBGsize));
```

This code uses *TEST_NUMBER* for the index of the parameter array and separately selects the P and Seed values for that index. The *std_logic_vector* values are then type converted to *unsigned* and then resized to the desired

RBGsize value. This removes any leading zeros that were added to make the vectors of a record the same length.

- Generate a reset signal to be a pulse for the first 50 ns of simulation and wait for five clk periods after the pulse.
- Using the appropriate parameters from the parameter array, generate a series of frames by setting the *frame_in* signal to '1' for the specified frame length parameter and with the number of frames as specified by the number of frames parameter. Hold *frame_in* at a '0' level for five periods of the *dclk_in* signal between frames. Synchronize the beginning of each frame with the falling edge of *dclk_in*. The desired timing for *RBGsize* = 4 is shown below.



For the first two test cases, we are setting the frame size in the simulation to be the length of the LFSR sequence. This is done so that we can easily see the complete sequence of the LFSR in one frame. But, this makes every frame identical, which is not accomplishing the goal of having pseudorandom data. Thus, we need to ensure that when applying the RBG, a frame is never the same length as the LFSR sequence. For increased randomness we want the length of the LFSR sequence to be much greater than the largest frame length. For the third parameter set, we will do exactly that.

New this year:

Add the following code to the test bench immediately before the stimulus process:

```
-- frame_out/dclk_out timing check
frame_out_timing: process
      variable time1 : time;
      variable time2 : time;
      variable time3 : time;
begin
      report "BEGIN TEST";
      for i in 1 to PARAM_ARRAY(TEST_NUMBER).NUM_FRAMES loop
            -- frame_out rise to dclk_out rise timing check
            wait until frame_out'event and frame_out = '1';
            time1 := now;
            wait until dclk_out'event and dclk_out = '1';
            time2 := now;
            time3 := time2 - time1;
            assert time3 = 500000 ps
                  report "Delay from rise of frame_out to rise of
                  dclk_out is " & time'image(time3) & ".
                  Should be 500000 ps"  severity error;
            -- frame_out fall to dclk_out rise timing check
            wait until frame_out'event and frame_out = '0';
```

```
            time1 := now;
            wait until dclk_out'event and dclk_out = '1';
            time2 := now;
            time3 := time2 - time1;
            assert time3 = 500000 ps
                  report "Delay from fall of frame_out to rise of
                  dclk_out is " & time'image(time3) & ".
                  Should be 500000 ps" severity error;
      end loop;
      report "TEST IS FINISHED";
      wait;
end process;

-- data_out/dclk_out timing check
data_out_timing: process
      variable time1 : time;
      variable time2 : time;
      variable time3 : time;
begin
      wait until rst'event and rst = '0';
      loop
      -- data_out rise or fall to dclk_out rise timing check
      wait until data_out'event; time1 := now;
      wait until dclk_out'event and dclk_out = '1'; time2 := now;
      time3 := time2 - time1;
      assert time3 = 500000 ps
            report "Delay from data_out transition to rise of dclk_out
            is " & time'image(time3) & ". Should be 500000 ps"
            severity error;
      end loop;
end process;
```

This code will check the *frame_out* and *data_out* signals relative to the *dclk_out* signal for correct timing. Timing errors will be reported in the simulator console.

- **Use simulation with iSim to verify correct operation of the RBG**

  Run the test bench with TEST_NUMBER set to 1. Run for 40 µs.

  In the iSim waveform viewer:

  - Group the test bench signals with name “Test Bench”.
  - Add the UUT signals and group them with the name “RBG”.
  - Select all of the UUT signals except shift_reg_f/g and shift_ctr and group them with the name “RBG_Signals”. Collapse this group.
  - Change the radix of the shift_reg_f/g and shift_ctr signals to “unsigned decimal.” Then expand both to show the individual bits.
  - Save your waveform configuration.

- Verify that the length of each frame is correct.
- Verify that the first frame has every possible value (except 0) with no value repeated.
- Verify that the sequence in the first frame is exactly repeated in the second frame.

Create a PDF of the simulation waveforms for test #1, configured as described above.

Repeat the simulation with TEST_NUMBER set to 2. Run for 70 µs. Verify that the frames have the correct length, both frames have exactly the same sequence, and no value is repeated in the sequence.

Create a PDF of the simulation waveforms for test #2, configured the same as for test #1.

Repeat the simulation with TEST_NUMBER set to 3. Run for 80 µs. For this test case, verify that the sequences in the six frames are all different.

Create a PDF of the simulation waveforms for test #3, configured as before.

Submit PDFs of the simulation waveforms for each of the three simulations in the lab assignment.

Demonstrate the operation of your RBG in the simulator to a lab proctor and get checked off on their sheet

- **Synthesize the RBG**

  There should be no synthesis errors or warnings.

  Debug any problems causing errors or warnings

- **Generate the RTL Schematic for the RBG module**

  Generate the RTL schematic.

  Submit an PDF of your RTL Schematic in the lab assignment.

- **Submit your VHDL Code**

  Submit your Random Bit Generator code (RandBitGen.vhd) in the lab assignment.

  Submit your Test Bench code (Test_RandBitGen.vhd) in the lab assignment.