Experiment 10. Reaction Timer: Part 1. Generating Random Delay

Masud ul Hasan · Aiman El-Maleh · Ahmad Khayyat – Version 151, 27 October 2015

Table of Contents

1. Objectives

- Learn how to generate random numbers using a *Linear Feedback Shift Register* (LFSR).
- Learn how to use counters to wait for specific amounts of time before performing an action.

2. Materials Required

- An FPGA prototyping board.
- Design and simulation software tools.
- The `onehz` module from previous experiments.

3. Background

3.1. Reaction Timer

The reaction timer is a circuit that measures human response time to a given event. We will develop the reaction timer as a game the goes as follows:

1. A player starts the game.
2. After a random delay, an LED will turn on.
3. As soon as the LED is on, the player should respond by pushing a button.
4. The circuit measures the time between the LED turning on and the player's response.
5. Depending on the response time, a message will be displayed, classifying the player's response as either *fast*, *good* (average), or *slow*.

To design this circuit, we need two components: a random time generator, and a response time calculator.

This experiment is divided into two parts. In the first part, you will build the random time generator, which will generate the random time delay between the start signal and the LED turning on. In the second part, you will build the response time calculator, which will measure the time between the LED turning on and the player’s response.

The Functional Block Diagram of the Final System figure shows the functional design of the system. The white area highlights the blocks you are going to build in this part of the experiment. The designer implementing this functional design, can either make each functional block as a stand-alone circuit, or merge two or more blocks into one circuit. This choice is left to the designer. Such decision can affect the complexity and the flexibility of the design.

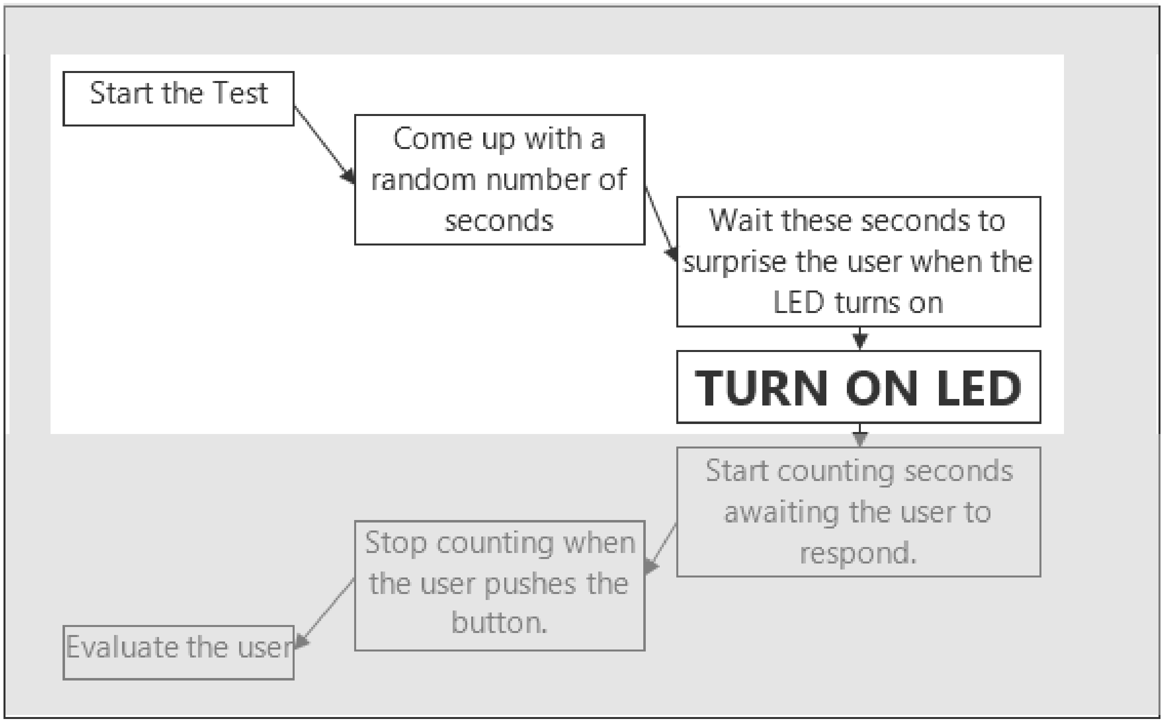
Start the Test
Come up with a random number of seconds
Wait these seconds to surprise the user when the LED turns on
TURN ON LED
Start counting seconds awaiting the user to respond.
Stop counting when the user pushes the button.
Evaluate the user

Figure 1. Functional Block Diagram of the Final System

3.2. Generating Random Numbers

Random numbers can be generated using a component called a *Linear Feedback Shift Register* (LFSR). An LFSR is a shift register with XOR feedback.

An *n*-bit LFSR can generate periodic sequences with a maximum period of $2^n - 1$. After the period is finished, the sequence will repeat. The placement of the XOR gates defines the period of the LFSR.

LFSRs in general can be described using polynomials that specify the specific bits used in constructing the XOR feedback.

An example of a maximal period LFSR is shown in the A 4-bit LFSR with Maximum Period Using Flip-Flops figure. The corresponding Verilog implementation is also shown in the Verilog Implementation of a 4-bit LFSR listing. The flip-flops used have an active-high preset to initialize the register to the `1111` state.

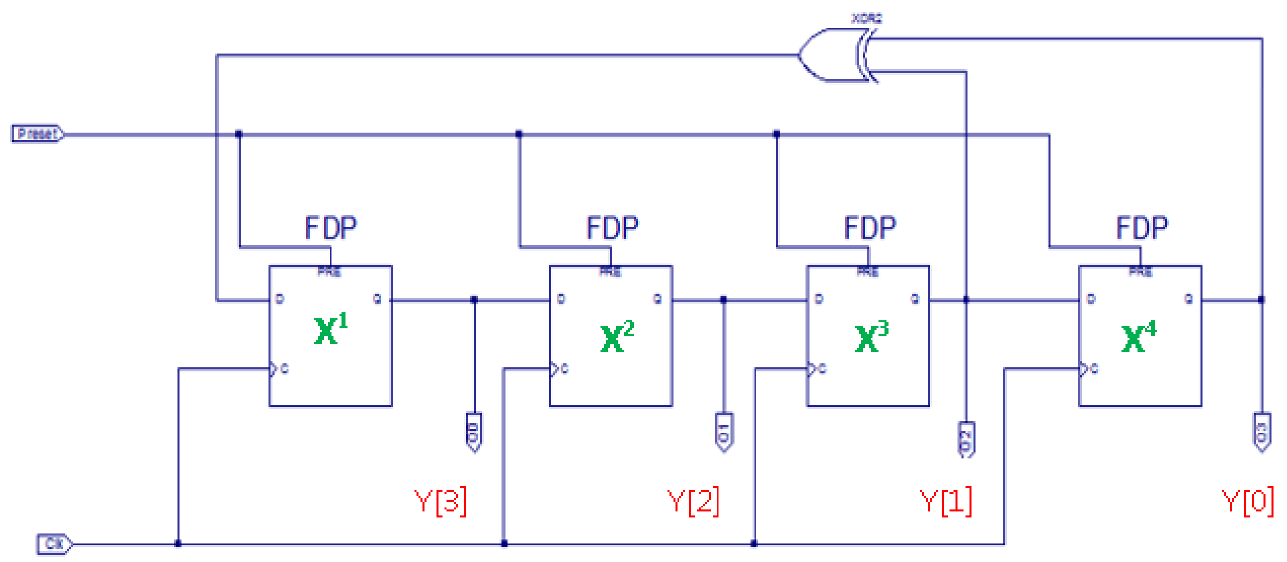

Figure 2. A 4-bit LFSR with Maximum Period Using Flip-Flops

Verilog Implementation of a 4-bit LFSR

```
module lfsr_4b (input clock, input preset, output reg [3:0] y);
    always @(posedge clock)
    begin
        if (preset)
            y <= 4'b1111;
        else
        begin
            y <= { y[1] ^ y[0], y[3:1] };
        end
    end
endmodule
```

The polynomial that describes this LFSR is:

$$1 + x^3 + x^4$$

As apparent, the bits mentioned in the polynomial [1] are the ones XOR'ed, and the result is used as the input to the shift register. That is, by knowing the polynomial, you can build a circuit that generates pseudo-random numbers.

For this LFSR, the state `0000` is a jam case: once you are in that state, you cannot leave it!

If this LFSR is initialized to Q_3 Q_2 Q_1 Q_0 = `1111`, the maximal period will be 15, and the output sequence will be as shown in the Function Table of the 4-bit LFSR and the Sequence of the 4-bit LFSR figure.

Table 1. Function Table of the 4-bit LFSR

Cycle	$Q_3^+ = Q_0 \oplus Q_1$	Q_2^+	Q_1^+	Q_0^+	Value
1	1	1	1	1	15
2	0	1	1	1	7
3	0	0	1	1	3
4	0	0	0	1	1

Cycle	Q = Q ⊕ Q	Q	Q	Q	Value
5	1	0	0	0	8
6	0	1	0	0	4
7	0	0	1	0	2
8	1	0	0	1	9
9	1	1	0	0	12
10	0	1	1	0	6
11	1	0	1	1	11
12	0	1	0	1	5
13	1	0	1	0	10
14	1	1	0	1	13
15	1	1	1	0	14
16	1	1	1	1	15

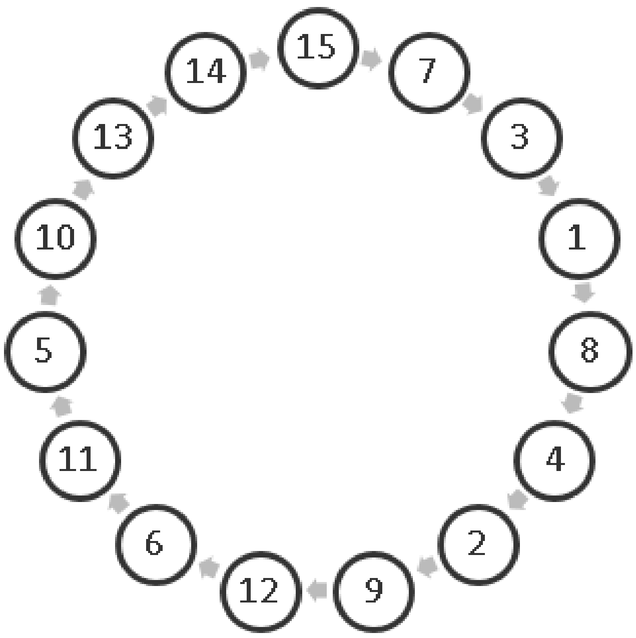

Figure 3. Sequence of the 4-bit LFSR

3.3. From Random Numbers to Random Delays

We would like to use the randomly generated number to determine the length of time to wait for user input. We can do that using a counter that counts up to the random number.

Counters, Revisited

Recall from previous experiments that a counter counts the edges of an input signal, typically the `clock` signal. It is is a sequential circuit that is composed of a combinational unit and a register. The register holds the value that the counter has reached, and the combinational unit will compute the next counter value, which depends on whether the counter is counting up or down.

In experiment 8 (Clock), you learned how to model a parameterized counter with configurable width. There are many other variations of counters. For example, synchronous vs. asynchronous clear, up counter vs. down counter, whether the counter can be loaded with an initial value or not, and so on.

The Verilog Description of a 2-bit Up Counter with Load Capability shows another example of counter implementation with parallel load: it can set the counter value to a user-provided value that can be loaded, in parallel, using a dedicated input pin for each bit. A value is loaded using the `data` input when the `load` input is set to `1`.

Verilog Description of a 2-bit Up Counter with Load Capability

```
module up_counter_2b (
    input clock, reset, enable, load, [1:0] data,
    output reg [1:0] count);

    always @(posedge clock)
    begin
        if (reset)
            count <= 0;
        else if (load)
            count <= data;
        else if (enable)
            count <= count + 1;
    end
endmodule
```

4. Tasks

4.1. Design a 5-bit LFSR

1. Use the example 4-bit LFSR described in the “Generating Random Numbers” section as a guide. Analyze its circuit and propose a design for a 5-bit LFSR based on the polynomial:

$$1 + x^2 + x^5$$

2. Write a Verilog module for your 5-bit LFSR design, named `lfsr_5b`. Simulate your design and verify that the sequence is maximal, i.e., period = 31.

4.2. Design a Down Counter

1. Use the 2-bit up counter described in the “From Random Numbers to Random Delays” section as a guide to design a *2-bit down counter with load.*

   The down counter should have a reset input that sets the counter to the value `2'b11` when set. In addition, the down counter should be loadable, which means that once a load input (`load`) is set to `1`, the counter is loaded with a value through the 2-bit `data` input. When the counter reaches 0, it should set an output signal `zero` to `1`.

2. Write a Verilog module for your 2-bit down counter design with the given requirements.

   Use th interface below for your design:

```
module down_counter_2b (
    input clock, reset, enable, load, [1:0] data,
    output zero, reg [1:0] count);
```

3. Simulate your design and verify that the `down_counter_2b` module counts correctly from a loadable value n, i.e., outputs = n, $n-1$, $n-2$, ... , 1, 0.

4.3. Counting Down Seconds

1. Use your 2-bit down counter to count seconds.

   Use your counter with a clock input that has a frequency of 1 Hz.

   Use the `onehz` module that you built in experiment 9 (Building a Digital Timer). Alternatively, you can use the following implementation.

```
module onehz (
    input  clock, reset,
    output clock_1hz);

    reg [26:0] counter;

    assign clock_1hz = (counter == 27'h5f5e0ff);

    always @(posedge clock)
    begin
        if (reset || clock_1hz)
            counter <= 0;
        else
            counter <= counter + 1;
    end
endmodule
```

2. Propose a design that loads some fixed number of seconds and starts counting down.

3. Write a Verilog module for your design. Name it `fixed_delay_counter`. Use the following module declaration:

```
module fixed_delay_counter (
    input  clock, reset, enable, load, [1:0] data,
    output zero, [1:0] count);
```

4. Implement your `fixed_delay_counter` module and verify its correct operation.

4.4. Counting Down a Random Number of Seconds

1. The following Verilog module, `delay_counter`, finalizes the design by integrating all the required components.

```
module delay_counter (
    input  clock, reset, load_rand,
    output led);

    wire       enable_rand;
    wire [1:0] count;
    wire [4:0] random;
    wire       zero;
    wire       load_rand_q;

    assign led = zero & ~load_rand;
    assign enable_rand = ~led & load_rand_q;

    dff load_rand_ff (clock, reset, load_rand, 1'b1, load_rand_q);
    lfsr_5b lfsr (clock, reset, random);
    fixed_delay_counter second_counter (clock, reset, enable_rand, load_rand,
                                        {random[3], random[1]}, zero, count);
endmodule
```

Here, `lfsr_5b` and `fixed_delay_counter` are your modules that you implemented in the previous tasks. `dff` is a D flip flop (see below).

The user should first press the `reset` button, which will reset the LFSR. Then, he presses the `load_rand` button, which will load the counter with a random number of seconds obtained from the LFSR. The 2-bit random number is composed of bit 1 and bit 3 from the 5-bit LFSR output. Note that when the user presses the `load_rand` button, the `onehz` module is reset. Once the counter reaches zero, the counter stops counting and the `led` signal should be `1`. The `led` signal remains `1` until the user presses the `load_rand` or `reset` buttons.

For the D flip flop, use the following Verilog module:

D Flip Flop Verilog Module

```
module dff (
    input  clock, reset, enable, d,
    output reg q);

    always @(posedge clock)
        if (reset)
            q <= 0;
        else if (enable)
            q <= d;
endmodule
```

2. Implement your `delay_counter` module on an FPGA by connecting the `clock` signal to the system clock (FPGA pin `V10`), and other signals to the necessary buttons and LEDs, and test it for correct functionality. You may want to declare `count[1]` and `count[0]` as output signals to see the value loaded in the counter when the `load_rand` button is pressed.
3. Answer the following questions:
   a. Why is the `led` output signal defined as `led = zero & ~load_rand`?
   b. Why is the `load_rand` signal stored in a D flip flop to obtain `load_rand_q`?
   c. Why is the `enable_rand` signal defined as `enable_rand = ~led & load_rand_q`?

5. Grading Sheet

Task	Points
Design and simulate a 5-bit LFSR	20
Design and simulate a 2-bit down counter with load	20
Design and test your `fixed_delay_counter`	20

Task	Points
Design and demonstrate your `delay_counter` on the FPGA	20
Lab notebook and discussion	20

1. More accurately, the bits whose positions correspond to the powers with non-zero coefficients in the polynomial.

Version 151
Last updated 2015-11-19 11:56:13 AST