LAB 7: Random Number Generator and Primality Testing

**Objective:** This lab consists of the design of a system that generates a random 32-bit number and determines whether that number is prime (“primality”).

**Prelab**

Complete the following exercises

1. Look at the pseudo code for the primality testing and modulus algorithms. Assuming a 50 MHz clock, estimate how many seconds it will take for the primality testing algorithm to run. State any assumptions you made.
2. Propose several optimizations that will bring the running time of this algorithm down to a reasonable time. These solutions must be reasonable, i.e. you must be able to implement these solutions in your hardware module.

**Project Description**

In this lab, you will design and implement a system that generates a random 32-bit unsigned integer, determines if that integer is prime, stores the integer in a small RAM, and displays the integer one byte at a time to the 7-Segment displays on the DE10-LITE board. The high level architecture of the system is shown in Figure 1.

![Figure 1; High level system architecture for the random number generator. Note that clock connections are not shown, but should be used where necessary.](image)

A detailed description of each block follows.

**Synchronizer**

Using asynchronous external IO signals directly in a digital design can introduce functional errors due to metastability if the external signals arrive too close in time to a clock edge which would violate the setup or hold time requirements. Synchronization is done by simply chaining together several flip flops as a shift
register as shown in Figure 2. In this lab, you must synchronize all external inputs using at least three flip flops per signal.

![Simple input synchronizer](image)

**Figure 2: Simple input synchronizer.**

**NumberGenerator**

This module will generate a random number using a 16-bit counter and the input sample (which shall be connected to KEY[0] in the top level design.) This module shall have a free running 16-bit counter that increments each clock cycle. When sample transitions from high to low (KEY[0] is pressed), the contents of the counter should be saved to the lower 16-bits of a 32-bit register. When sample then transitions from low to high (KEY[0] is released), the contents of the counter should be saved to the upper 16-bits of the 32-bit register. Based off the timing of the pressing and releasing KEY[0], a random 32-bit number will be generated in the register.

This module also has a debug[1:0] input which selects between routing the random number to the output, or known prime/composite numbers. This will be helpful for debugging your system. The expected behavior of this is given in the following Table.

<table>
<thead>
<tr>
<th>debug[1:0]</th>
<th>value[31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Output value[31:0] should be assigned the random 32-bit number stored in the internal register.</td>
</tr>
<tr>
<td>01</td>
<td>Output value[31:0] should be assigned a constant 32-bit non-prime number.</td>
</tr>
<tr>
<td>10</td>
<td>Output value[31:0] should be assigned a constant 32-bit prime number.</td>
</tr>
<tr>
<td>11</td>
<td>Output value[31:0] should be assigned the large 32-bit prime number, (2^31)–1</td>
</tr>
</tbody>
</table>

**PrimeTester**

This module will determine if the random number generated by NumberGenerator is prime. When start (connected to KEY[1] in the top level design) transitions from high to low, PrimeTester should register its input dividend[31:0] and set its output done = 0. When it has determined whether dividend[31:0] is prime or composite, it should set done = 1. If dividend[31:0] is prime, then the output is_prime should be assigned 1. Otherwise, is_prime should be zero. Signal done should be routed to LEDR[1] and is_prime should be connected to LEDR[0].

The simplest way to test for primality is given in the following pseudo-code. For notation, we will use

\[ N/D = (Q,R) \]

Where N is the dividend, D is the divisor, Q is the quotient, and R is the remainder.
**Algorithm:** PrimeTester

**INPUT:** N

For D = 2, 3, ..., N – 1
   If remainder(N, D) == 0  // divisor evenly divides dividend
      Return false
   End
End
Return true

**End Algorithm**

This can be implemented easily using a state machine. Pseudo code for the “remainder” operation is given below.

**Algorithm:** remainder for N÷D using 32-bit numbers (n=32)

**INPUT:** N[(n-1):0], D[(n-1):0]

R = 0

For i = n-1, n-2, ... , 1, 0
   R = R << 1
   R[0] = N[i]
   If R >= D
      R = R – D
   End
End
Return R

**End Algorithm**

**Pseudo-code instructions**

<table>
<thead>
<tr>
<th>For i = 5, 4, 3, 2, 1, 0</th>
<th>i = 5</th>
<th>i = 4</th>
<th>i = 3</th>
<th>i = 2</th>
<th>i = 1</th>
<th>i = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R = R &lt;&lt; 1</td>
<td>R = 000000</td>
<td>R = 000010</td>
<td>R = 000100</td>
<td>R = 000000</td>
<td>R = 000000</td>
<td>R = 000010</td>
</tr>
<tr>
<td>R[0] = N[i]</td>
<td>R = 000001</td>
<td>R = 000010</td>
<td>R = 000101</td>
<td>R = 000000</td>
<td>R = 000000</td>
<td>R = 000011</td>
</tr>
<tr>
<td>If R &gt;= D, R = R – D</td>
<td>false</td>
<td>false</td>
<td>R = 000000</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

**Example 3.** Calculation of the remainder of 43÷5 with 6-bit numbers. Loop iterations are shown in columns 2–7.

n=6 6-bit numbers, N = 43₁₀ = 6'b101011₁₂, D = 5₁₀ = 6'b000101

Note that these algorithms have not been optimized at all. In order to meet the time-requirements for this lab, you will have to think about optimizations and implement them.

**OutputRam**

This is a simple RAM implemented in flip-flops that will store the randomly generated number. The RAM should have four addresses with an entry width of 8-bits (32-bits of total memory). It will need separate read and write addresses. The write signals (write_data[7:0], write_enable, write_addr[1:0]) come from the Serializer module. Connect the read address read_addr[1:0] to switches SW[3:2]. Display the output value read_data[7:0] in hexadecimal on the 7-segment displays.

**Serializer**

This module will store the 32-bit signal from NumberGenerator into the output RAM one byte at a time. That is, when it detects a transition on its input save, it will store data_in[7:0] to address 0 of the RAM, data_in[15:8] to address 1 etc. Note that only one address in OutputRam may be written at a time, so this write operation will have to take place over at least 4 clock cycles.
Inputs and Outputs

The table below summarizes the top level inputs and outputs of the lab.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Connection in Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY[0]</td>
<td>Input sample on NumberGenerator.</td>
</tr>
<tr>
<td>KEY[1]</td>
<td>Input start on PrimeTester</td>
</tr>
<tr>
<td>SW[1:0]</td>
<td>Input debug[1:0] on NumberGenerator</td>
</tr>
<tr>
<td>SW[3:2]</td>
<td>Input read_addr[1:0] on OutputRam</td>
</tr>
<tr>
<td>SW[9]</td>
<td>Global Reset</td>
</tr>
<tr>
<td>LEDR[0]</td>
<td>Output is_prime from PrimeTester</td>
</tr>
<tr>
<td>LEDR[1]</td>
<td>Output done from PrimeTester</td>
</tr>
<tr>
<td>HEX0[7:0]</td>
<td>Display lower 4 bits in hex of the output of OutputRam</td>
</tr>
<tr>
<td>HEX1[7:0]</td>
<td>Display upper 4 bits in hex of the output of OutputRam</td>
</tr>
</tbody>
</table>

Design Requirements

- Write a module and a testbench for each of the blocks NumberGenerator, PrimeTester, OutputRam, and Serializer.
- Make optimizations to the primality testing and modulus algorithms so your module determines the primality of ANY 32-bit number in less than one second. Full credit cannot be given if your design takes longer than one second. Methods to speed up a system include:
  - Increasing the clock frequency of your design using a PLL
    See Lab 3 for a reminder of how to set a target frequency for a design and to find its actual maximum operating frequency.
    See Instantiating and using a PLL on the DE10-LITE board on the course web page
  - Parallelizing the prime testing
  - Performing more intelligent testing
- Connect LEDs LEDR[2] to LEDR[9] to signals of your choice to show system status

Checkoff: Show the working system on the DE10-LITE board to your TA. They should be able to verify the debug operations and repeatedly generate random numbers until a prime number is found.

Helpful Tips

- Carefully think about the interaction of signals between modules. The more time you spend thinking about the requirements of the system as a whole, the easier it will be to implement the design as a whole.
- You may use the website https://primes.utm.edu/curios/includes/primetest.php to check the primality of 32-bit hexadecimal numbers. For example, you can type in 0x7FFFFFFF to verify that this number is prime.
- Approximately 5% of 32-bit numbers are prime so you will need to test approximately 20 random numbers on average to find a prime number.
Submitted Work

[20 pts] Prelab – Solutions to prelab exercises.

[70 pts] Lab Checkoff – Demonstrate the working system to your TA.

  [+10 points Extra Credit] Extend your design to automatically search for candidate prime numbers by starting with a single random value (NumberGenerator), making small increments for new candidate numbers, and stopping when a prime number is found.

  [+10 points Extra Credit] Extend your design to work with 64-bit numbers and demonstrate.

[10 pts] Lab Report

Submit all Verilog hardware and testbench code that you wrote. Do not include any code that you did not write. Uploading your verilog is essential to receive credit for the entire lab.

A) [5 points] Print and submit during your lab session.

B) [5 points] Upload to Smartsite using the following steps by the end of your lab session:

  1. Make a folder on your computer
  2. Copy all verilog files you wrote into the folder—only the ones you wrote
  3. “zip” the folder into a single .zip file
  4. Log onto Smartsite, click Assignments on the left side of the page, click on the correct lab number
  5. Upload the .zip file as instructed