

Lab 3 Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

Understand how to implement a pseudo-random number generator on the FPGA development board. Learn how to design an appropriately-sized LFSR. Choose taps correctly to maximize the number sequence. Find a way to allow for "zero" to be shown as a random number. Show hexadecimal digits on the seven segment display.

Procedure

We started out by reading the lab-3 requirements for the random number generator project. We then used the System Builder tool to initialize the Quartus Prime project. We determined which I/O would be needed by this lab, those being: the system clock, push buttons, seven segment displays, and LEDs (so they don't glow dimly). We started by drawing up a basic block diagram for the random number generator. We determined we would only need 2 modules, the seven segment module and the LFSR. Instead of hard-coding the seed, we added the switches to be able to set the seed, making it much easier to try out different combinations.

For our LFSR, we decided to make it 10 bits wide since we had 10 switches to set the seed. The 8 bits in the middle were used to show the actual random number. We picked mostly prime numbers for the taps to ensure that they were not co-primes. We also designed the LFSR to use 4 taps. Just like the C program, we calculated the new random bit, shifted the bits over, and appended the new bit.

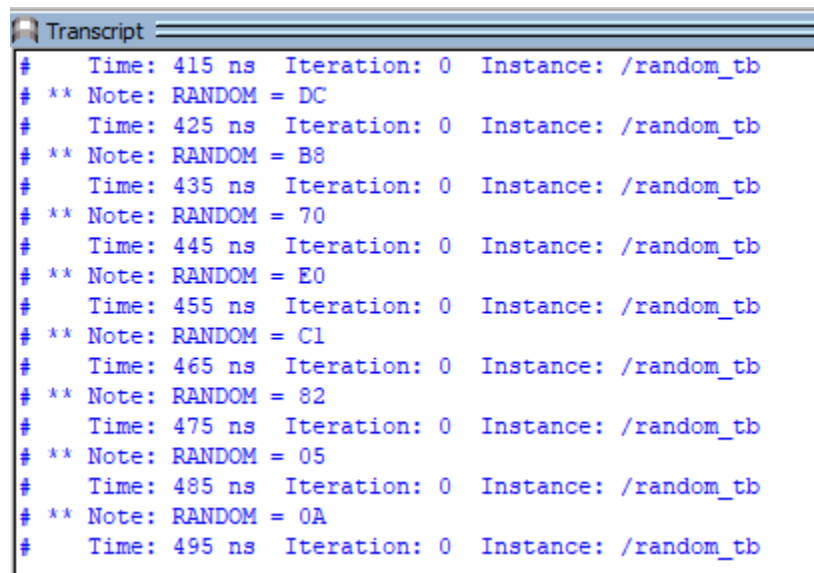
Lastly, for the testbench, we wanted an easy way to quantify and verify the results of our random number generator. We designed our testbench to print the value of the random number out to ModelSim's console each clock cycle. We then created a Python script to parse the output and count how many unique numbers appear before repeating the sequence. We were able to test the tap positions and see which taps worked best.

Results

Initially, we had problems getting the simulation to work with the print statements for the console output. We wrote a function to print the `std_logic_vector` to hex. The function

worked, but it only worked in the device under test code. We wanted to put the function into the testbench itself since putting the function in the device under test code is not good practice. The problem was that VHDL did not have a way to pull out signals buried under submodules, or so we thought. We then found that VHDL 2008 did have such functionality.

We found that changing the compile version in the ModelSim project file did not work, but eventually we found that the compile version could be set on a per file basis in the GUI. After making those changes, the function halfway worked. ModelSim correctly found the sub-signal and graphed it on the waveform, but the sub-signal showed up as “0xXX” in the terminal. For some reason the aliases in ModelSim did not work correctly, so instead of using an alias we supplied the print statement with the full signal path and that worked. Figure 1 shows an example of what our function outputs to ModelSim’s console.

The image shows a screenshot of the ModelSim Transcript window. The window has a title bar with a small icon and the word "Transcript". The text inside is as follows:

```
# Time: 415 ns Iteration: 0 Instance: /random_tb
# ** Note: RANDOM = DC
# Time: 425 ns Iteration: 0 Instance: /random_tb
# ** Note: RANDOM = B8
# Time: 435 ns Iteration: 0 Instance: /random_tb
# ** Note: RANDOM = 70
# Time: 445 ns Iteration: 0 Instance: /random_tb
# ** Note: RANDOM = E0
# Time: 455 ns Iteration: 0 Instance: /random_tb
# ** Note: RANDOM = C1
# Time: 465 ns Iteration: 0 Instance: /random_tb
# ** Note: RANDOM = 82
# Time: 475 ns Iteration: 0 Instance: /random_tb
# ** Note: RANDOM = 05
# Time: 485 ns Iteration: 0 Instance: /random_tb
# ** Note: RANDOM = 0A
# Time: 495 ns Iteration: 0 Instance: /random_tb
```

Figure 2. Console Output

We wrote a Python script to count the number of unique numbers we generated. As expected, since we are actually using 10-bits, it will take 1024 iterations to get our 256 unique random numbers. Thus, during the 1024 iterations we do get duplicates in between, as all 256 are not generated in the first iterations. With our original tap positions, we got around 234 unique numbers. After making changes to the taps, we then got our desired result of 256 unique random numbers.

Figure 2 shows an example of part of the wave generated by our test bench. The two 7-segment displays change every clock cycle while the generate button is pressed and

the reset button is not pressed. The value of the switches sets the seed for the random number generator whenever the reset button is pressed.

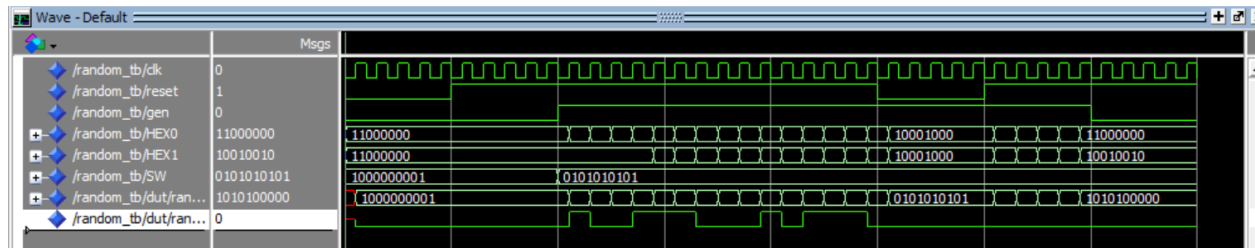


Figure 2. Simulation Waveform

As expected, this design uses 11 registers, 10 for the LFSR and one to calculate the new random bit (see Figure 3). Putting the design on the board, worked as shown in simulation and the demo during class. We even got lucky and landed on zero after 5 tries. Though we could start it at zero by using the 10 switches on the board. The last thing we tested was putting all of the switches to zero to ensure that the LFSR is stuck at zero.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Thu Sep 30 01:49:58 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	Random
Top-level Entity Name	Random
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	28 / 49,760 (< 1 %)
Total registers	11
Total pins	73 / 360 (20 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 3. Flow Summary

Conclusion

The purpose of this lab was to design a pseudo-random number generator using an appropriately sized LFSR to obtain hexadecimal values between 0x00 and 0xFF. We learned how to use an LFSR to generate pseudo-random numbers upon button presses. We also went above and beyond the requirements of this lab by designing the 10 switches to set the seed upon reset button presses. We learned how to print out sub-signals to ModelSim's console. We used a 10 bit seed, so we were able to use the middle 8 bits for our random number to obtain the values 0x00 - 0xFF. Overall, we fulfilled the requirements of this lab and gained additional experience with VHDL, Quartus, and ModelSim.