Anant Singh
Advanced Hardware Design
October 29, 2021

Midterm

A1. High-Level FSM and Datapath

1.1 FSM

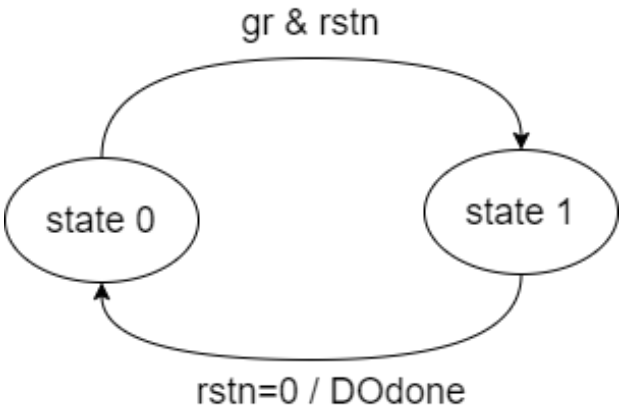

Image 1: Next State Machine 1

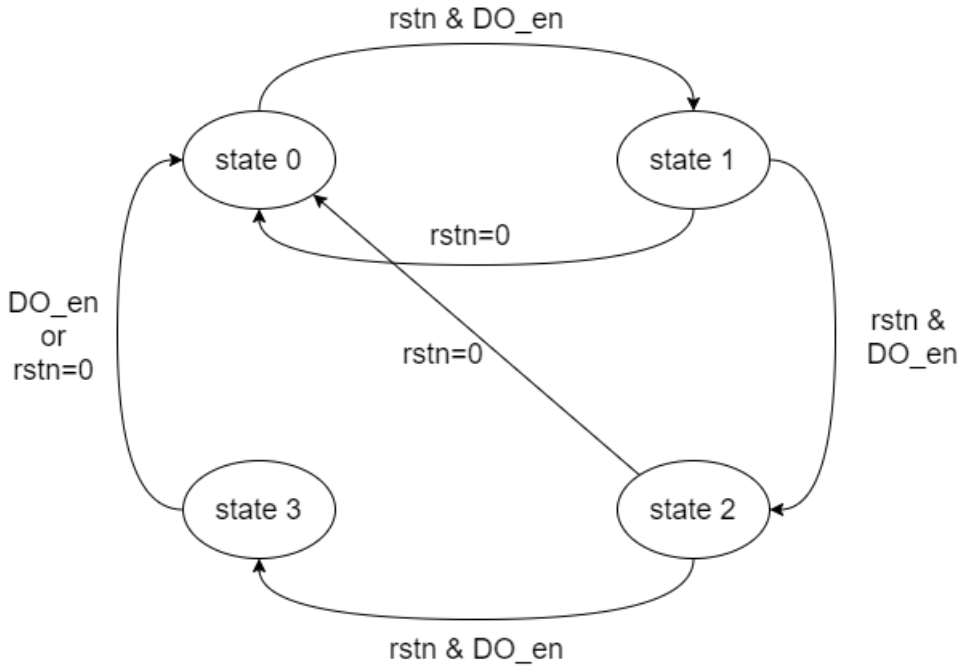

Image 2: Next State Machine 2

1.2 Datapath

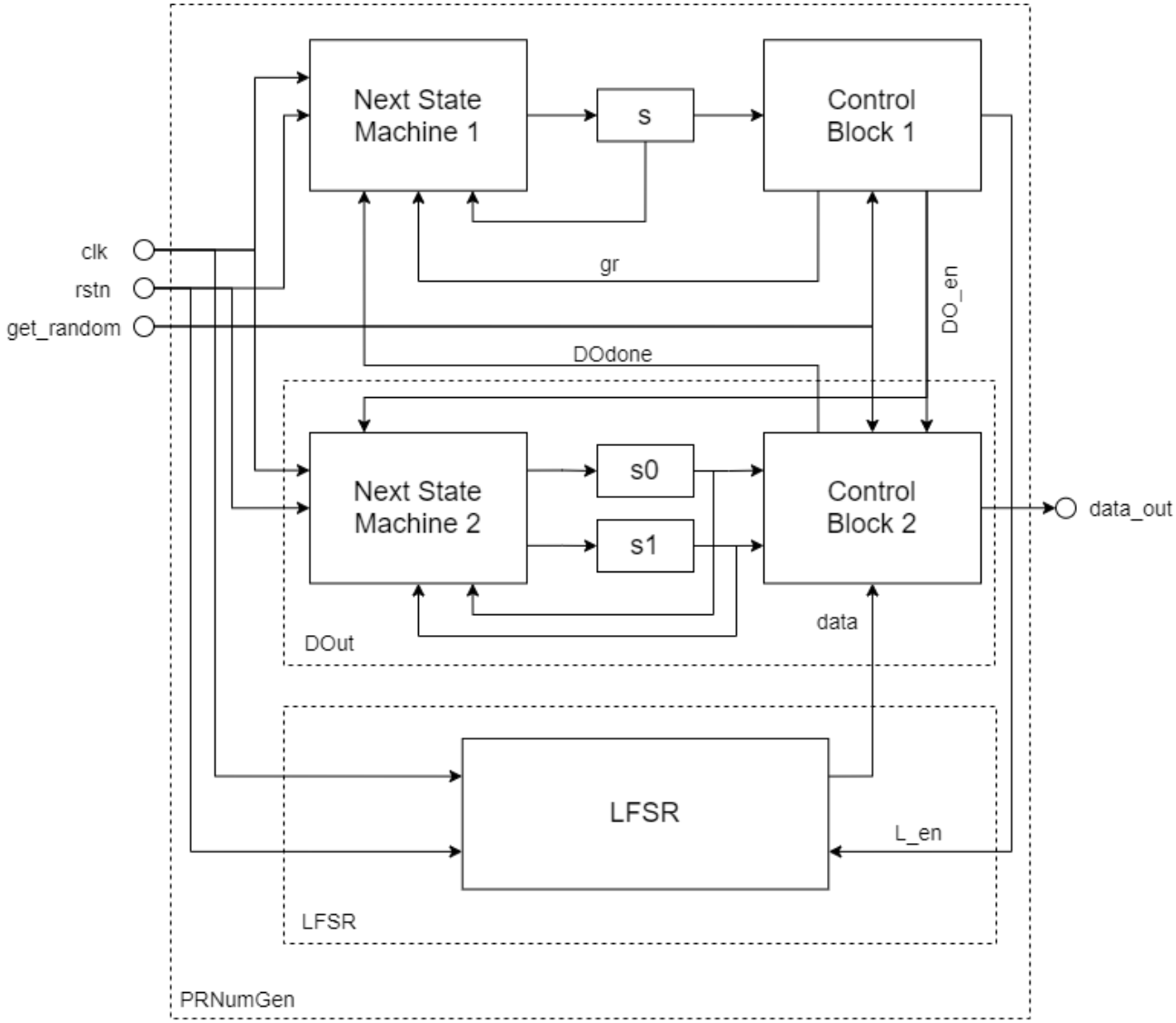

Image 3: Datapath for PRNumGen

B2. Code Description and Justification

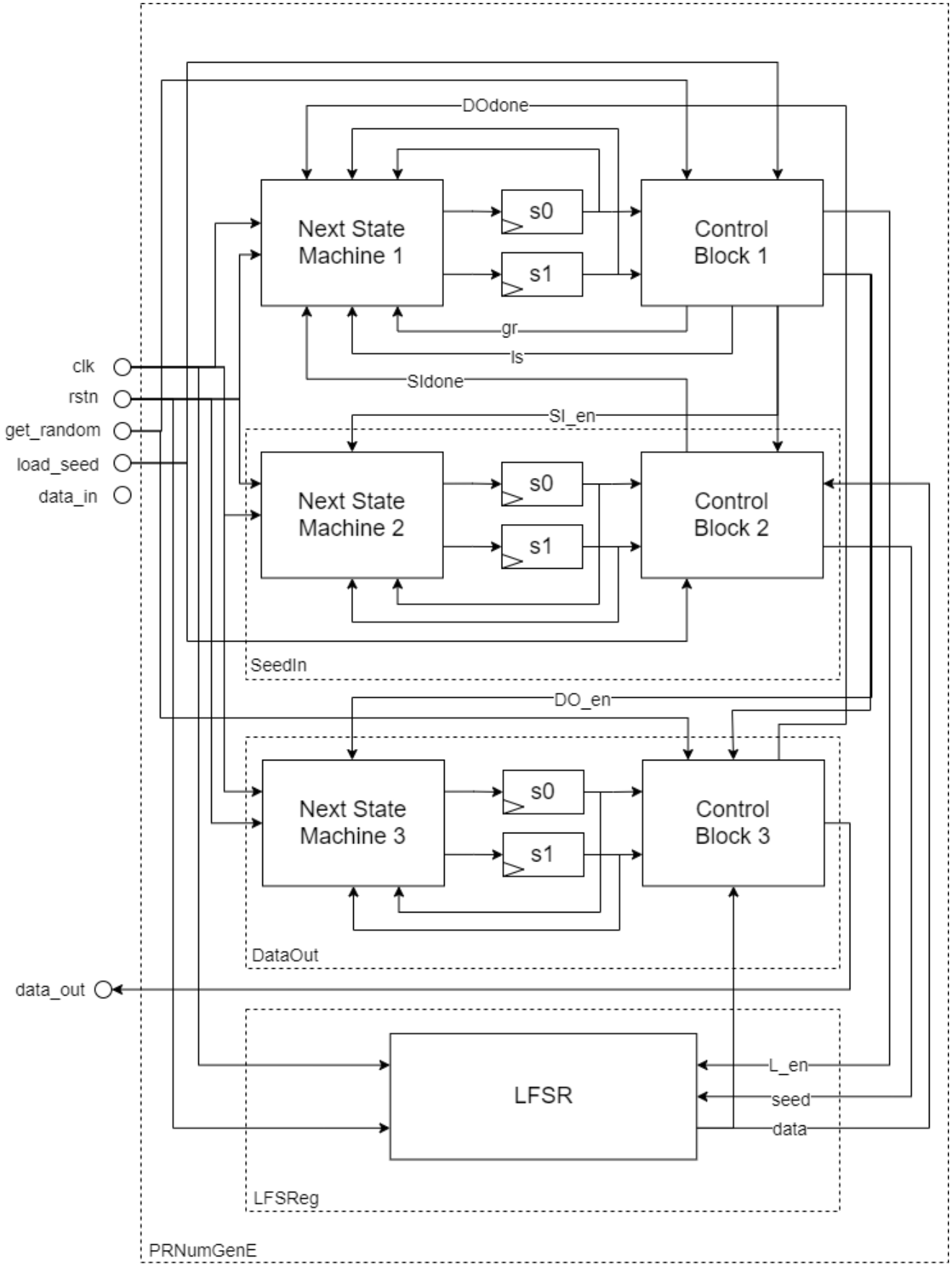

Image 4: Datapath for PRNumGenE

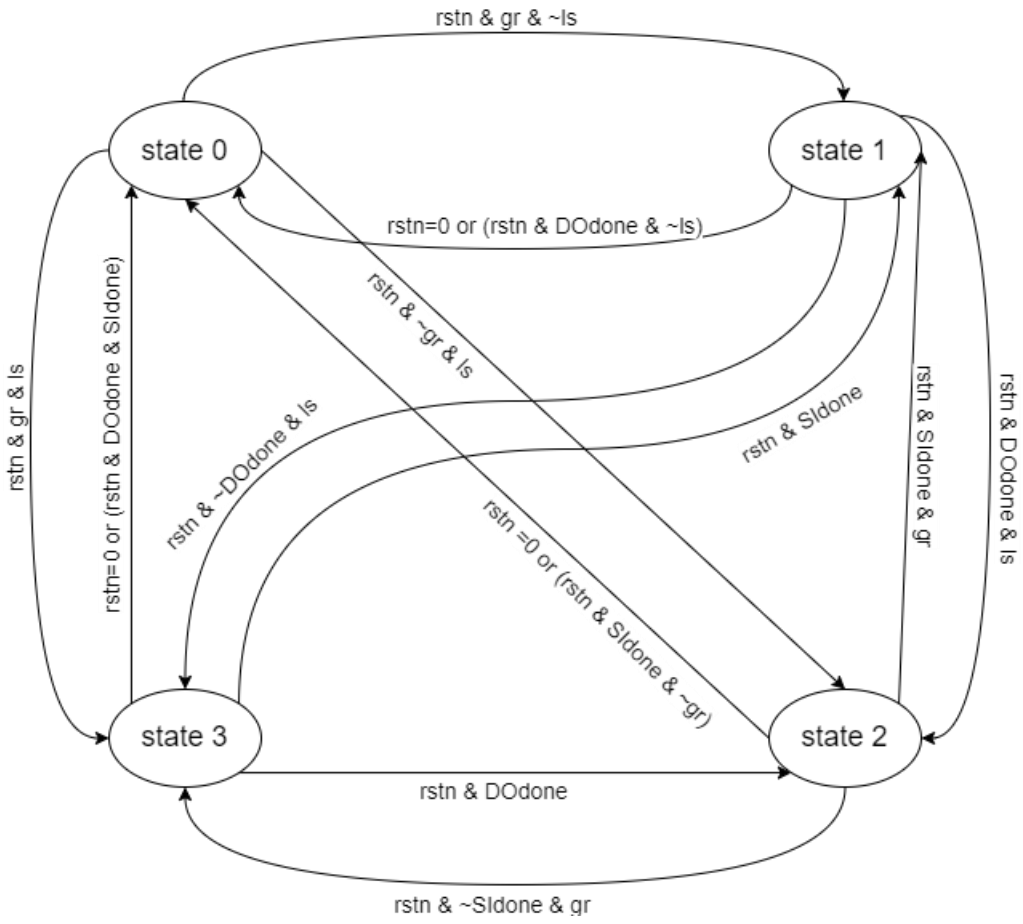

Image 5: Next State Machine 1

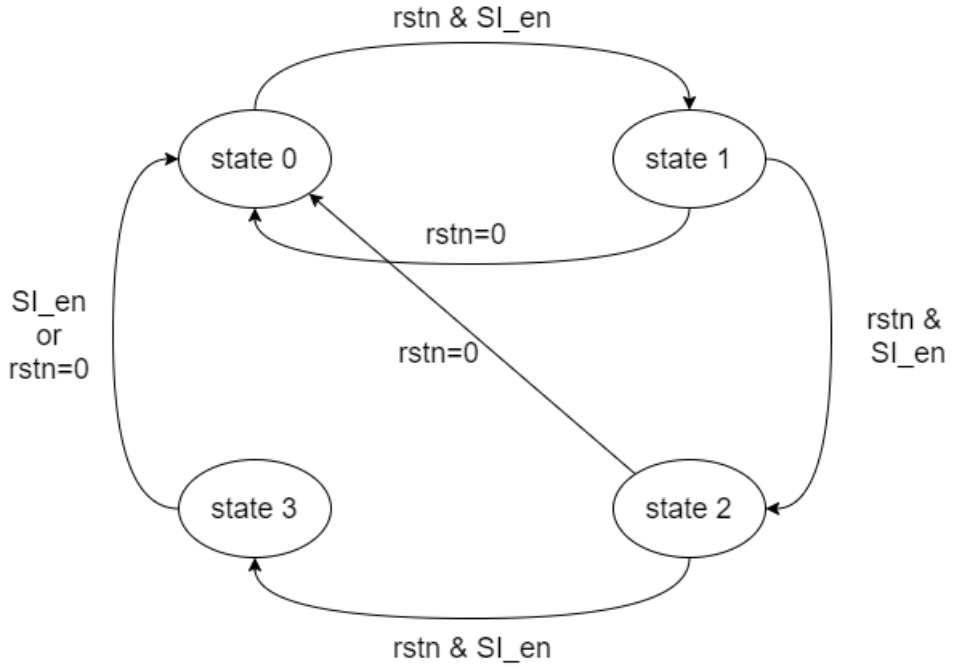

Image 6: Next State Machine 2

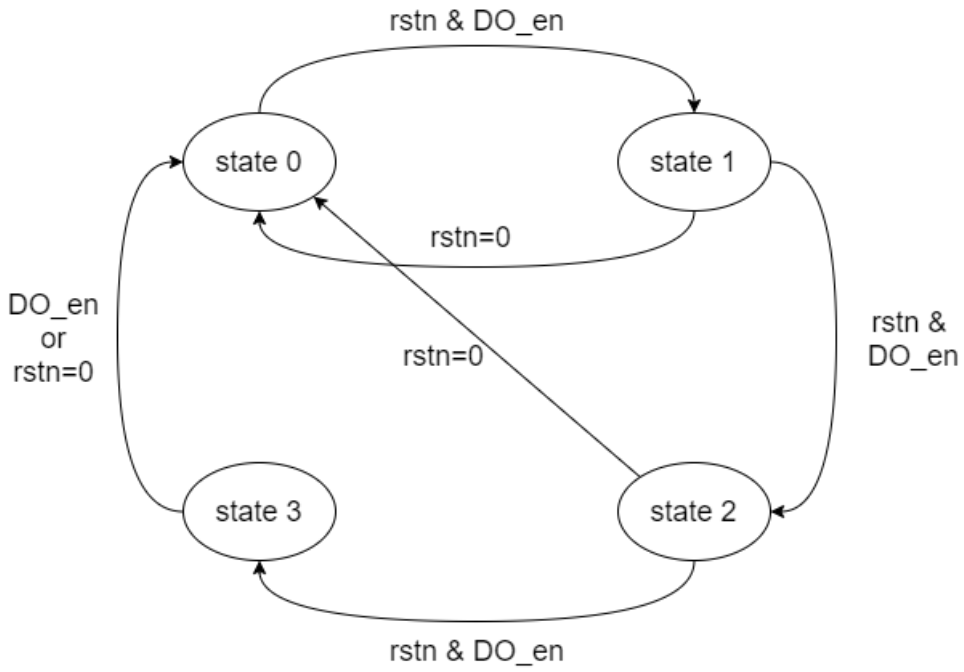

Image 7: Next State Machine 3

Design Process – As per the design specifications given, there are 3 modules in the design each having its own FSM and Control Block to generate outputs. The main module PRNumGen initiates 3 modules SeedIn (for loading serial input seed from data_in), LFSReg (to generate shifts from seed) and DataOut (for data output to data_out port in serial manner). The design treats 4 continuous cycles as more than 4 cycles as on the forth cycle we check for the condition id output/input is still required and at that point if the values is high then it repeats.

The **Main FSM** follows the following simplified rules, (Image 4, Image 5)

It is a 4 state design. If reset is low then state returns to state0. All the modules have asynchronous reset and this resets the output to 0 and seed to 0x02468ACD, until the reset is high nothing changes in the design as soon as the reset signal is high again. If no input is given i.e. load_seed, and get_random both are zero, then LFSReg becomes active and keeps on generating sequences. Both load_seed and get_random are used to imitate semi asynchronous behavior to control Main FSM by generating gr and ls signals. If load_seed, becomes high and get_random remains low, then then state 2 comes into action and SeedIn module start working while LFSR module goes down. If load_seed is high for more than 4 cycles then the design will continue accepting seed value in a multiple of 4 cycles. If load_seed is high only for less than 4 cycles, then SeedIn module will run only for 4 cycles. As soon as the input is taken SIdone is turned high to signal main FSM to switch state back to state0.I f get_random, becomes high and load_seed remains low, then state 1 comes into action and DataOut module start working while LFSR module goes down. If get _random is high for more than 4 cycles then the design will continue outputting values in serial order as per the specs in a multiple of 4 cycles. If get_random is high for less than 4 cycles, then DatatOut module will only run for 4 cycles. As soon as the output is finished DOdone is turned high to signal main FSM to switch state back to state0. If at the same time both load_seed and get_random becomes high then which ever the system is in it goes to state 3 straight, state 3 continues until SIdone or DOdone signal is received if SIdone is received first then system goes to state1 and normal operation continue from there. If DOdone is received first, then system goes to state2 and normal operation continue from there. For case when both Sidone and DOdone becomes high simultaneously then system goes to state 0.

DataOut Module works in following way – (Image 4, Image 7)

The Next State Machine 3 start operation when DO_en signal becomes high from Control Block 1. State 0 to State 4 switches on each clock cycle, it Reset becomes low then system goes to state0. Control Block 3 outputs the lower 8 bits in the next cycle then next 8 bits then next 8bit then finally first 8 bits, at state 4 it is checked whether get_random is high not, if it is high then repeat 4 states again and output the values again, else DOdone turns high and control again goes to Main FSM and Control Block 1, which in-turn turns off DataOut module. No LFSR shifts occurs in this process as L_en becomes low. It takes data for output from LFSReg Module. For all other cases DataOut Module keeps out 0. Note: DOut Module from design PRNumGen works in same way (Image 2, Image 3)

SeedIn Module works in following way- (Image 4, Image 6)

The Next State Machine 2 start operation when SI_en signal becomes high from Control Block 1. State 0 to State 4 switches on each clock cycle, it Reset becomes low then system goes to state0. Control Block 3 accept the lower 8 bits in next clock cycle then next 8 bits then next 8bit then finally first 8 bits, at state 4 it is checked whether load_seed is high not, if it is high then repeat 4 states again and accept input the values again, else SIdone turns high and control again goes to Main FSM and Control Block 1, which in-turn turns off SeedIn module. No LFSR shifts occurs in this process as L_en becomes low. For all other cases SeedIn keep the data value from LFSR module.

LFSReg Module works in following way- (Image 4)

This module generates the shifts for taps as per my N number the taps are at [0,4,5,9,11,13,31] (zero indexed). At each clock cycle the data is updated to the next shift until L_en goes down from Control Block 1. At reset the seed value is set to 0x02468ACD. When SeedIn Module is active i.e SI_en is high then LFSReg Module takes seed value from seed signal from SeedIn module. No shift takes place if L_en is turned low. (Unless for reset and SI_en case) Note: LFSR from PRNumGen works in same way, only difference is absence of SeedIn conditions it does not takes seed in and hard set to 0x02468ACD everytime. (Image 3)

Main Module from PRNumGen works in following way- (Image 1, 3)

if reset is low then state returns to state0. All the modules have asynchronous reset and this resets the output to 0 and seed to 0x02468ACD, until the reset is high nothing changes in the design as soon as the reset signal is high again. if no input is given i.e. random both is low, then LFSR becomes active and keeps on generating sequences. get_random is used to imitate semi asynchronous behavior to control Main FSM by generating gr signal. If get_random, becomes high, then state 1 comes into action and DOut module start working while LFSR module goes down. If get _random is high for more than 4 cycles, then the design will continue outputting values in serial order as per the specs in a multiple of 4 cycles. If get_random is high for less than 4 cycles, then DOut module will only run for 4 cycles. As soon as the output is finished DOdone is turned high to signal main FSM to switch state back to state0.

PRNumGen_testbench

1) The testbench is designed for PRNumGen.v file and tests it against inputs from a file Testcases.txt. Testcases-
   // Test 1 - check for get_random = 0, rstn = 0 // Test 2 - check for get_random = 1, rstn = 0
   // Test 3 - check if LFSR works against 1st output when get random is low for less than 4 cycles
   // Test 4 - test if get_random is high for more than 4 cycles
   // Test 5 - Reset in mid output cycle when get_random is high for more than 4 cycles
   // Test 6 - Reset in mid output cycle when get_random is high for less than 4 cycle
   // Test 7 – Extensive testing of LFSR for remaining shifts generated from python program for 100 values.
   Testbench tests for all the possible cases of importance. All other cases are sequential combination of these tests.

PRNumGenE_testbench

1) The testbench is designed for PRNumGenE.v file and tests it against inputs from a file Testcases.txt.
   Testcases-
   //--------- DataIn Loader Test ----------//
   // Test 1 - check for load_seed = 0, rstn = 0 // Test 2 - check for load_seed = 1, rstn = 0
   // Test 3 - check if LFSR works against 1st input seed when load_seed is high for less than 4 cycles
   // Test 4 - check if LFSR works against 1st input seed when load_seed is high for more than 4 cycles
   // Test 5 - Reset in mid input cycle when load_seed=1 for less than 4 cycles
   // Test 6 - Reset in mid input cycle when load_seed=1 for less than 4 cycles
   //--------- DataOut Test ----------//
   // Test 7 - check for get_random = 0, rstn = 0 // Test 8 - check for get_random = 1, rstn = 0
   // Test 9 - check if LFSR works against 1st output when get random is low for less than 4 cycles
   // Test 10 - test if get_random is high for more than 4 cycles
   // Test 11 - Reset in mid output cycle when get_random is high for more than 4 cycles
   // Test 12 - Reset in mid output cycle when get_random is high for less than 4 cycle
   //--------- DataOut and Dataload simultaneous Testing ----------//
   // Test 13 - Testing when get_random=1 after 1 cycle of load_seed=1, and get_random=1 both for less than 4 cycles
   // Test 14 - Testing when load_seed=1 after 1 cycle of and get_random=1, and get_random=1 both for less than 4 cycles
   // Test 15 - Testing when rstn =0 mid cycle get_random=1 load_seed=1, both for less than 4 cycles
   // Last Test - Extensive testing of LFSR for a custom seed and 100 values.

   Testbench tests for all the possible cases of importance. All other cases are sequential combination of these tests.
   Test values came from Python Program, an implementation of LFSR with same taps, included in project.

C1. Asynchronous Reset or Synchronous Reset

The design uses asynchronous reset. I choose asynchronous reset because it has the advantage of being able to reset flops without the need of a clock. I designed the module as if a human will interact with it, so if reset signal is given for any given time the circuit should return to default condition no matter what. If clock signal frequency is very low, then it may be possible that reset signal might not get registered in case of asynchronous reset design. Asynchronous reset design has another advantage in general cases over synchronous reset that it reduces additional Gate required for synchronous reset scheme at the input of Flipflops. Synchronous reset design has the advantage of avoiding glitches in the system and hence leads to a more reliable and simple design, this advantage is more useful when reset signal is generated by internal circuit, as it has path delays and Asynchronous reset design may get glitches.

C2. Requirement of registers

Registers are very important part of the design when there is a need to store some value to process it. Registers can store, transfer, and accept data quickly that can be used by other elements of the design. Without using registers there is no way that a data element can be stored for any further use or in other words there can be no memory element in the circuit, and it will be a memoryless design. Registers particularly differentiate from other types of memories due to closeness and compactness of registers to the processing unit, memory access from any other type of memory element will be laggy at best. The design for Problem A uses 47 register bits, Problem B uses 85 register bits.

C3. Real world application of LFSR

LFSR are used in Cryptography as Pseudo Random Number Generator for use in stream ciphers. These can be used in combination and a Boolean function to generate the keystream. As LFSR is a linear system a combination of several LFSR are used and outputs are forwarded into a Boolean function or a summation function to make the key generator non-linear. The initial seed determines the output of keystreams and works as secret keys. The key generator generates a 128 bit or 256 bit key and then these are used to encrypt and decrypt the given data. Important LFSR based streams ciphers include A5/1, A5/2 in GSM cell phones, E0 used in Bluetooth.

Sources – https://www.rocq.inria.fr/secret/Anne.Canteaut/MPRI/chapter3.pdf

https://en.wikipedia.org/wiki/Linear-feedback_shift_register

C4. Cycle for repeating values

It will take 33554430 cycles + additional initial setup cycles (depending on test scenario) for this design to start repeating values. I used a python program to count the number of cycles using same taps. The program works same as a LFSR and the loop runs until the input value is the generated output value by following the shifts using taps. This is not so good of a count for a 32 bit LFSR as it can be as high as 4294967295 if taps are adjusted to their optimum indices. The period can be increased by adjusting the taps to their optimum indices or by increasing the data bits to more than 32. The optimum indices for 32 bit are [1,5,6,31] and the repeat period for these taps is 4294967295 cycles.

Appendix A

Simulation screenshots-

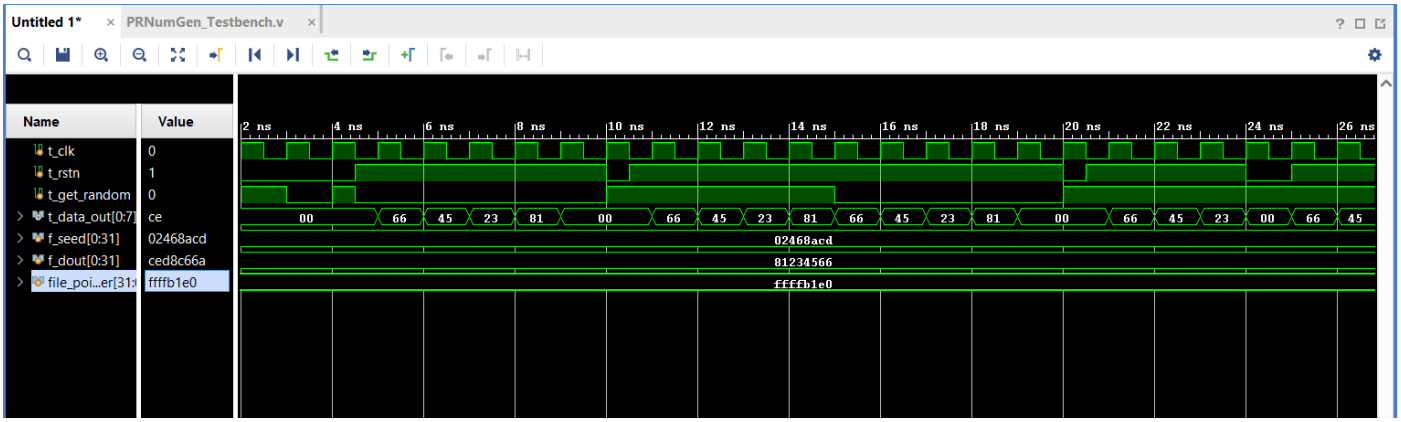

Image 8: Simulation of PRNumGen_Testbench

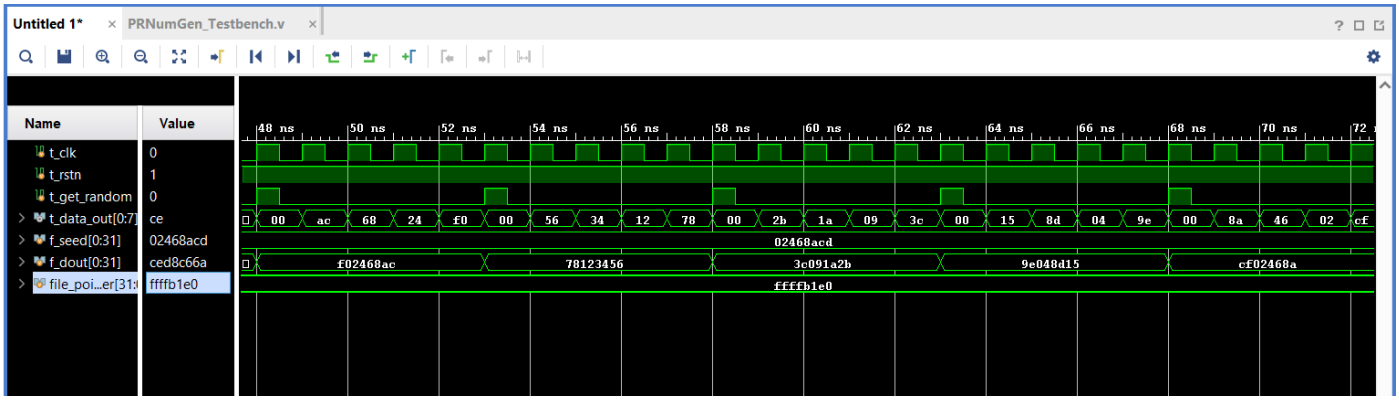

Image 9: Simulation of PRNumGen_Testbench

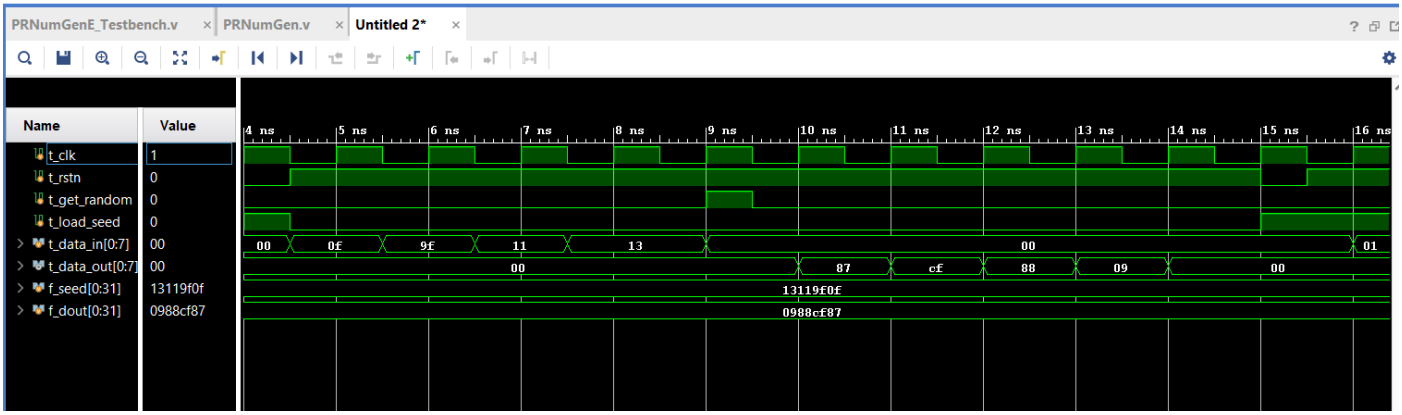

Image 10: Simulation of PRNumGenE_Testbench

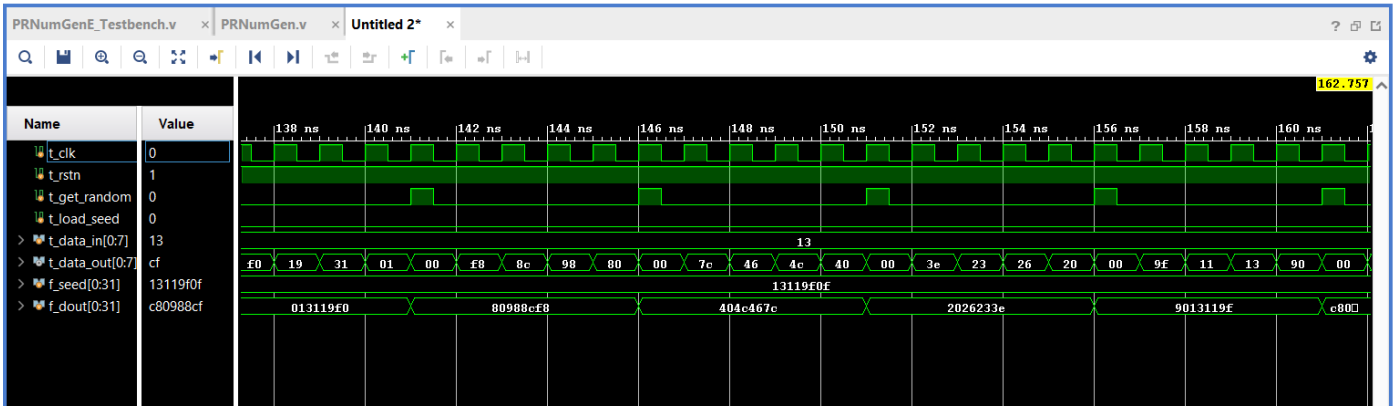

Image 10: Simulation of PRNumGenE_Testbench