

Final Fall 2021

C1. Code Description and Justification

a) Datapath

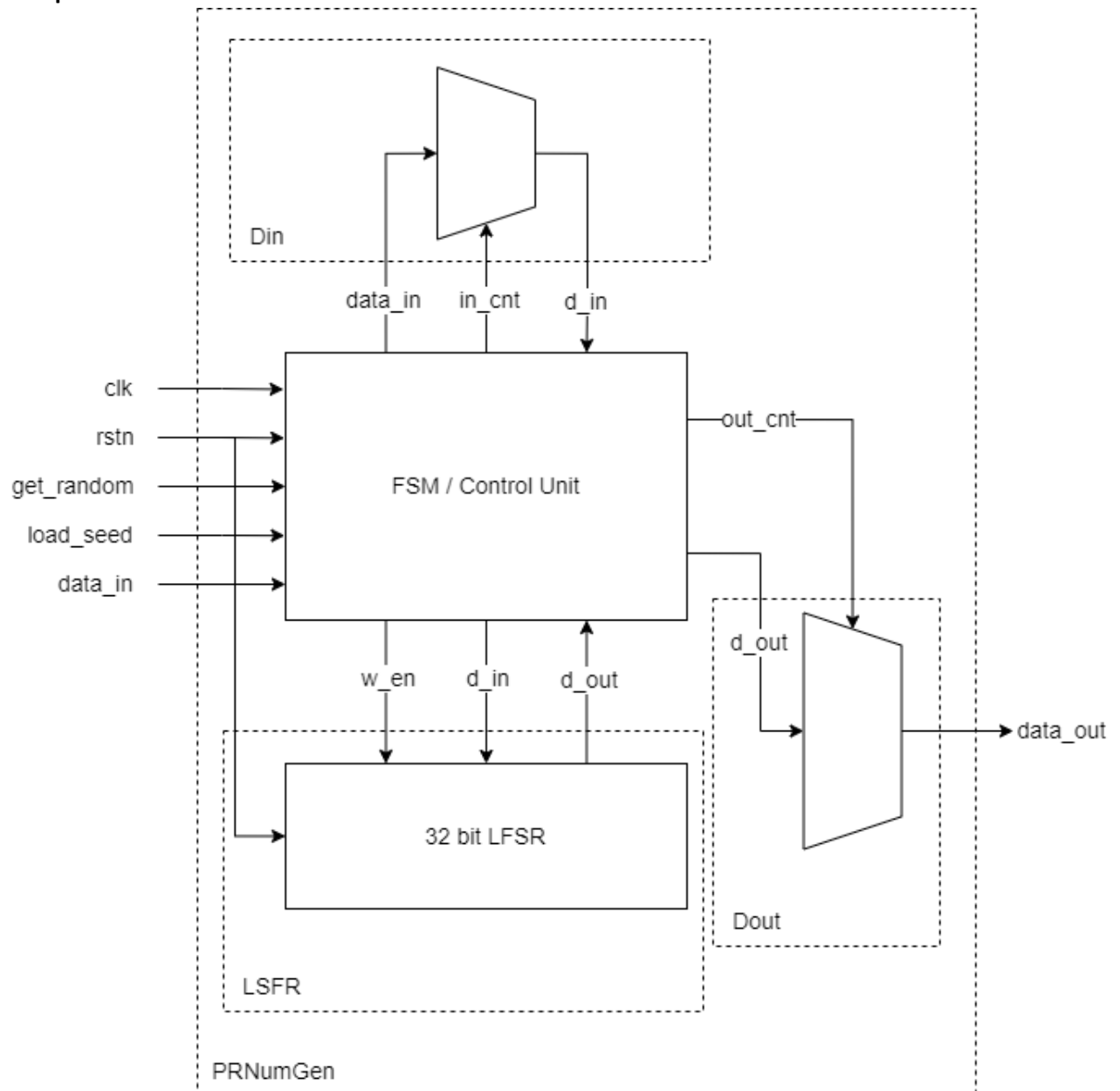


Image 1: Datapath

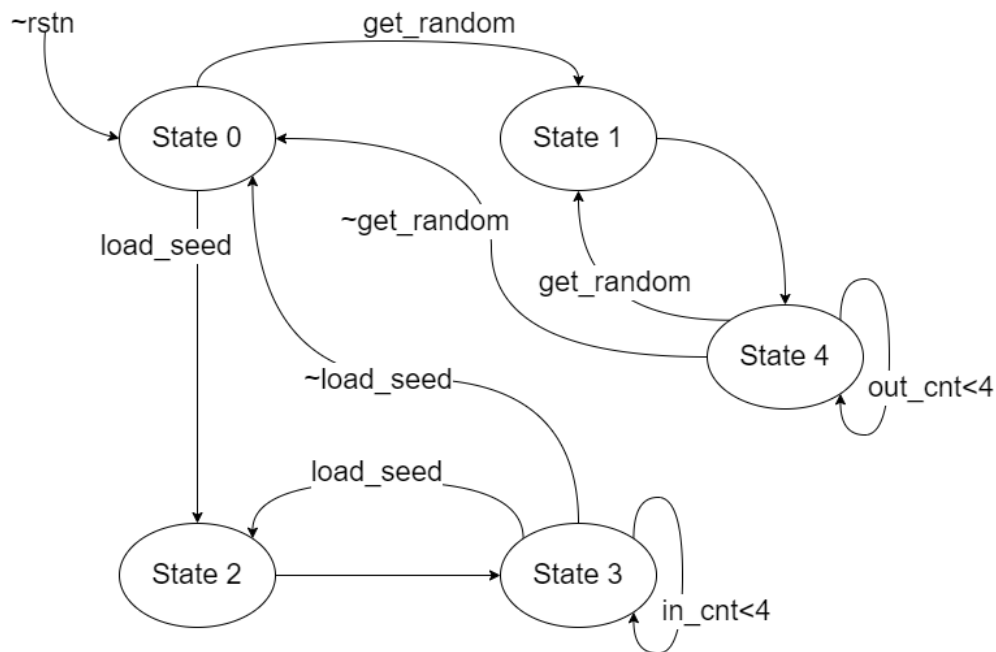
FSM

Image 2: FSM

Design Process – As per the design specifications given, there are 4 modules in the design. The top module PRNumGen calls 3 modules Din (for loading serial input seed from data_in), LFSR (to generate shifts from seed) and Dout (for data output to data_out port in serial manner). The design provide output/input if data is requested/loaded for more than 4 cycles repeatedly. The design uses ap_cint.h C library to use registers more effectively.

The **FSM** follows the following simplified rules, (Image 2)

It is a 5 state design. If reset (rstn) is low then state returns to state0. On reset output goes to 0 and seed to 0x02468ACD, until the reset is high nothing changes in the design. If no input is given i.e. load_seed, and get_random both are zero, then LFSR becomes active and keeps on generating sequences. If get_random becomes high state goes to state 1 and Data output sequence starts, State changes to State 4 on next cycle. State 4 remains current state until out_cnt<4 , i.e until the output sequence completes, after being in state 4 for 3 cycles FSM checks if get_random is high or not, if it is high then State goes back to State 1 and the cycle repeats, but if get_random is low then state goes to State 0 again. If load_seed becomes high state goes to state 2 and Data input sequence starts, State changes to State 3 on next cycle. State 3 remains current state until in_cnt<4 , i.e until the input sequence completes, after being in state 3 for 3 cycles FSM checks if load_seed is high or not, if it is high then State goes back to State 2 and the cycle repeats, but if load_seed is low then state goes to State 0 again. State 1 and State 4 invokes Dout functions to write d_out to data_out serially. State 2 and State 3 invokes Din function to take serial input and load into d_in variable and on last input cycle LFSR is invoked to load the new seed. While in State 0 FSM invokes LSFR to generate new shifts and LFSR remains idle in all the other states.

Dout Module works in following way

This module of function is used for writing 32 bit d_out to 8 bit data_out output port serially. It is basically a multiplexer and sel pointer works as select line.

Din Module works in following way

This module of function is used for loading 8 bit data_in into 32 bit d_out serially. It is basically a multiplexer and sel pointer works as select line.

LFSR Module works in following way

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). The taps are stored in TAPS define directive as hex value. TAPS is basically 32 bit number in which all the bits are 0 except for tap locations. This hex number is used to and with Seed to generate the xor output which is then used to add to random number rnum. LFSR function is controlled by w_en signal from PRNumGen module. If w_en is high then new seed from input pointer d_in is loaded as new seed for shift generation. If w_en is low then shifts occur and loaded into d_out pointer. If rstn becomes low the seed is set to 0x02468ACD.

PRNumGen Module from PRNumGen works in following way-

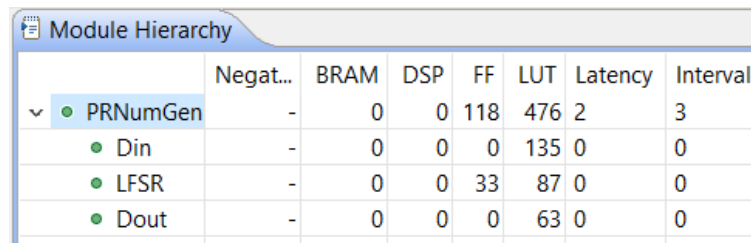
This module invokes all the other modules/functions in the program and works as top module for the design. If reset (rstn) is low then it resets the values to default. On reset output goes to 0 and seed to 0x02468ACD, until the reset is high nothing changes in the design. If no input is given i.e. load_seed, and get_random both are zero, then LFSR becomes active and keeps on generating sequences. If get_random becomes high state data output sequence starts until out_cnt<4 , i.e until the output sequence completes, after 3 cycle it checks if get_random is high or not, if it is high then output cycle repeats, but if get_random is low then shifts start again. If load_seed becomes high state then data input sequence starts until in_cnt<4 , i.e until the input sequence completes, after 3 cycles it checks if load_seed is high or not, if it is high then input cycle repeats, but if load_seed is low then shifts starts again. It invokes Dout function to write d_out to data_out serially. And it invokes Din function to take serial input and load into d_in variable and on last input cycle LFSR is invoked to load the new seed. While if not input or output request is present then it invokes LFSR to generate new shifts and LFSR remains idle in all the other cases.

C2. Resource Utilization

The design used near 0 % of the resources for Nexys 4 DDR FPGA board. 112 register bits were used. The largest component is top module and after that Din is the second largest module with 135 LUTs. The utilization is shown below in the screenshot-

Utilization Estimates				
Summary				
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	63
FIFO	-	-	-	-
Instance	-	-	33	285
Memory	-	-	-	-
Multiplexer	-	-	-	122
Register	-	-	79	-
Total	0	0	112	470
Available	270	240	126800	63400
Utilization (%)	0	0	~0	~0

Image 3: Utilization summary

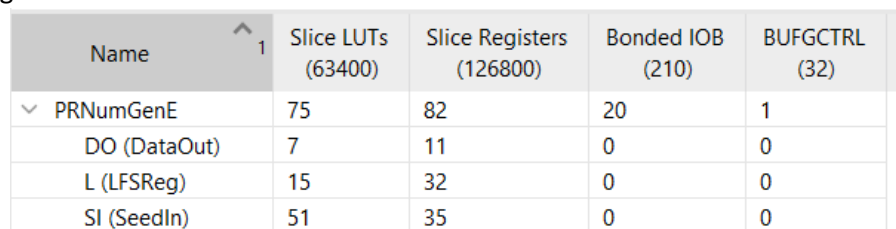


	Negat...	BRAM	DSP	FF	LUT	Latency	Interval
PRNumGen	-	0	0	118	476	2	3
Din	-	0	0	0	135	0	0
LFSR	-	0	0	33	87	0	0
Dout	-	0	0	0	63	0	0

Image 3b: Hierarchical Resource Utilization

C3. Differences in Pure Verilog and HLC C Design

The main difference was in resource utilization, almost like difference in Assembly code and a High level Language code. Same thing took almost 7 times less resources to implement. The utilization screenshot from Pure Verilog design is shown below –



Name	Slice LUTs (63400)	Slice Registers (126800)	Bonded IOB (210)	BUFGCTRL (32)
PRNumGenE	75	82	20	1
DO (DataOut)	7	11	0	0
L (LFSReg)	15	32	0	0
SI (SeedIn)	51	35	0	0

Image 4: Utilization summary of Pure Verilog Design

Another major difference was in the amount of control over design. We have more control over design while coding in pure HDL. Also The HLS design takes more cycles to do the same.

C4. HLS C Simulation and Testbench Justification

PRNumGen_TB:

The testbench reads a file named Testcases.txt for verification of the working of the design. It tests PRNumGen.c file for the following testcases

```
// Test 1 - Test for zero output if get_random = 0, load_seed = 0
// Test 2 - Reset Test
// Test 3 - get_random=1 more than 4 cycle
// Test 4 - Reset in between of a cycle get_random cycle
// Test 5 - load_seed=1 less than 4 cycle
// Test 6 - load_seed=1 more than 4 cycle
// Test 7 - Reset in between load_seed cycle
// Test 8 - Extensive testing against 1000 test vectors
```

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.

C5. C / RTL Co-simulation

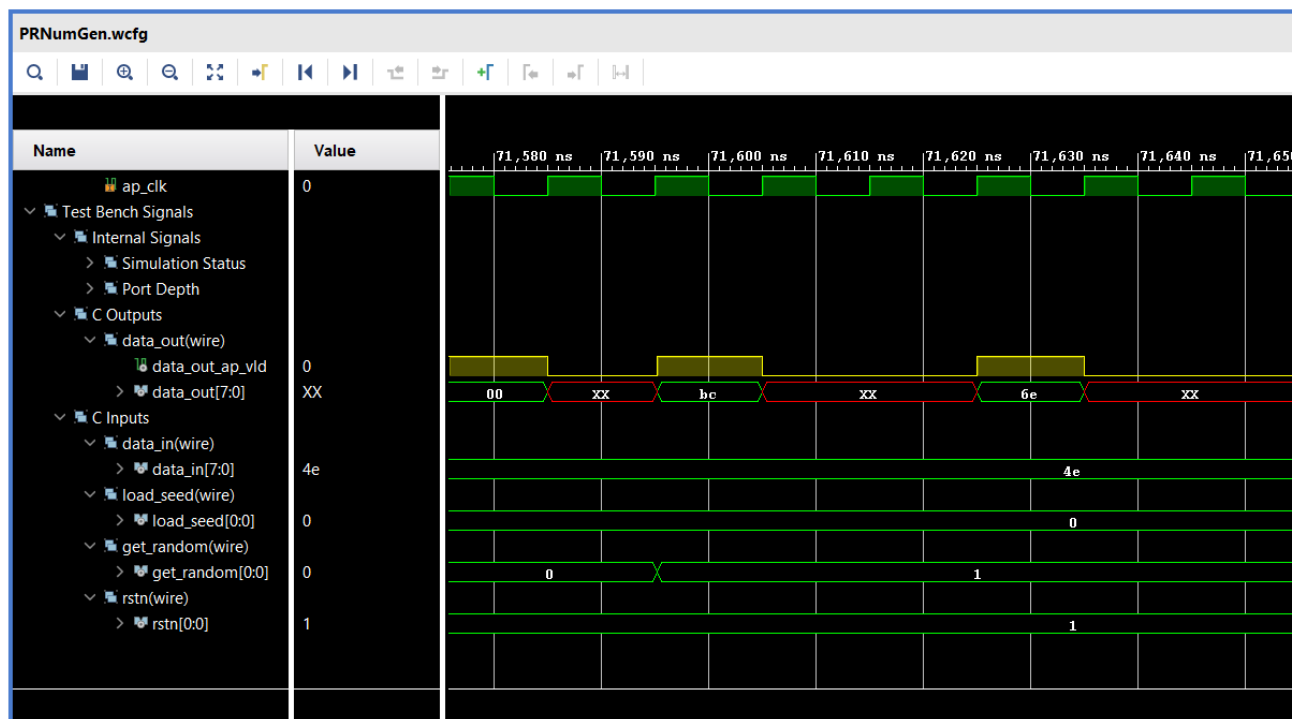


Image 4: Utilization summary of Pure Verilog Design

As per the design It takes 3 clock cycles to implement 1 output input sequence as shown in image 4, the design included Data valid signals automatically. A complete output cycle is shown below in image 5.

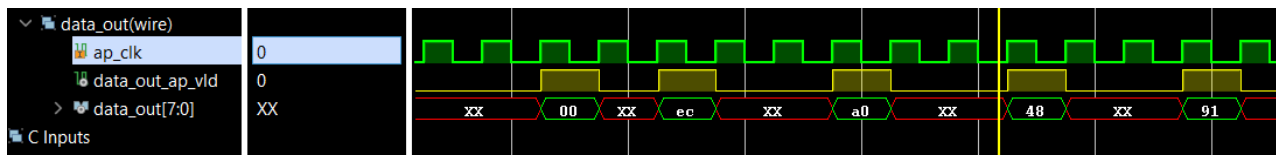


Image 5: Complete Output Cycle

B1. Directives / Pragas

1. PIPELINE Pragma:

Pipeline should increase efficiency by effectively using the resources which are dormant in stages where other parts are being used in the current cycle. After using the Pipeline pragma resource utilization is reduced and now design is implemented in 3 intervals instead of 6. I used Pipeline Pragma on Funtion level.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	63
FIFO	-	-	-	-
Instance	-	-	33	287
Memory	-	-	-	-
Multiplexer	-	-	-	111
Register	-	-	85	-
Total	0	0	118	461
Available	270	240	126800	63400
Utilization (%)	0	0	~0	~0

Image 6: Complete Output Cycle

2. ALLOCATION Pragma

Allocation Pragma should limit the resource or operator utilization for example if it set on Adder then this pragma should limit the number of adder used to the limit set by pragma. I used Allocation pragma on addition operation with a limit of 1. This resulted in decreased number of adder in the design and increase the control steps. As shown below-

Σ Expressions(7)	0	0	0	63	19	13	3
+	0	0	0	24	6	2	0
tmp_4_fu_179_p2	0	0	0	12	3	1	0
tmp_2_fu_154_p2	0	0	0	12	3	1	0

Image 7: Before Pragma Application

Σ Expressions(5)	0	0	0	42	13	8	3
+	0	0	0	12	3	1	0
grp_fu_90_p2	0	0	0	12	3	1	0

Image 8: After Pragma Application

Student Exam Code of Conduct

I certify and affirm that,

1. I am aware of the NYU Code of Conduct.
2. Work presented is 100% my own. I have not collaborated with anyone on the test.
3. I did not misrepresent someone else's work as my own.
4. I did not discuss, nor in any way divulge the content of this test or my answers.
5. I understand that my failure to abide by this code of conduct will result in consequences outlined in the NYU Code of Conduct.

Name: Anant Singh

Signature:  _____

Date: 23 December 2021