Research Shell

Maciej Trawka

January 4, 2023

Contents

Chapter 1

Overview

The Research Shell is in fact a Python3 shell wrapped by `PtPython` with some useful modules, classes and methods included. This document covers those items assuming, that a reader is familiar with Python syntax.

1.1 Architecture of Research Shell

Look at Figure 1.1. It shows Research Shell wrappers from the top to the Python3 core. User calls a Bash script, which prepares and executes a command containing Python3 call, module PtPython loading, and then importing all useful libraries included in module `aio`. So to run the Research Shell you need to call:

```
research_shell [python_file_name_to_execute]
```

If no argument, then the Research Shell appears and is ready to execute Python commands. If a script file is specified as an argument, then its content is executed after importing all modules and the shell closes.

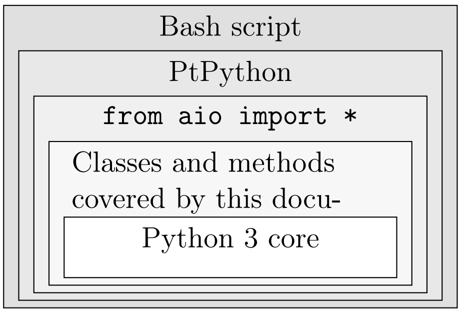

Figure 1.1: Research Shellarchitecture.

There is also a special mode of Research Shell, called **Testcase Mode**. It makes easy to execute a complete testcases. The testcase is a directory having a regular structure:

`testcase_name/`
- `data/` .... automatically added to the searching path
- `results/` .... Created automatically by Research Shell
- `driver.py` .... Main script - your Testcase code

By running the command:

```
research_shell_drun
```

the Research Shell runs in the Testcase mode. In such case it checks if the `driver.py` file exists. If so, then it removes and recreates the `results` directory and goes there (so `results` is now the Current Directory). Now, a content of `dirver.py` is executed. In `results` directory a `transcript.txt` is created. To print something to the screen and also to the transcript file, you need to call the `print(*args)` method of class `Aio`, i.e.:

```
# This text will be printed to the screen only:
print("Text on the screen only")

# This also appears in the transcript file:
Aio.print("Text on the screen and in the transcript")
```

1.2 Getting started with Research Shell in Siemens' infrastructure

There is a very simple way to start using the Research Shell having access to Siemens' infrastructure. First, login to any remote machine. Then, clone the *research* git repository:

```
git clone /wv/stsgit/research.git
```

After that, the following tree will be created in your Current Dir:

```
research/
├── research_shell/ ........................................ main Research Shell home
│   ├── bin/ ........................................ Research Shell binaries and libs
│   │   ├── research_shell ................................ Research Shell executable
│   │   └── research_shell_drun .......... quick runner for a Research Shell testcase
│   ├── libs/ .................................. core research-related Python modules
│   └── utils/ ..................................................... utilities
│       └── install_python_libs ................................ installation script
└── testcases/ ............................ a place for research-related testcases
    └── research_shell_verifier/ . a testcase to verify Research Shell installation
```

Note: It is recommended to add the `./research/research_shell/bin` directory to your `PATH` environmental variable, but is not neccessary to use the Research Shell.

To make sure you have all required Python modules installed and to install the missing ones, just run the script from `utils` directory:

```
bash ./research/research_shell/utils/install_python_libs
```

Once the script finishes, you can easy call the Research Shell:

```
./research/research_shell/bin/research_shell
```

The PTPython (Python) shell will appear. See next chapters to getting familiar with research-related modules, classes and functions, you can use together with standard Python ones in the Research Shell.

1.2.1 Verifying the Research Shell installation

There is a testcase called `research_shell_verifier` especially usable to make sure, that all dependencies are installed correctly and if there is no error in core research procedures, like those ones to primitive polynomials searching, LFSR simulators etc. Tp run that testcase (as well as any other one), simply go to the main testcase directory:

```
cd ./research/testcases/research_shell_verifier
```

and call:

```
../../research_shell/bin/research_shell_drun
```

You can see a standard output stream printed on your screen during execution. Once finished, the transcript is available in `./results/transcript.txt` file.

Chapter 2

Class Polynomial

`Polynomial` is an object intended to analyze polynomials over GF(2). An object of type `Polynomial` holds polynomial coefficients (as a list of positive integers) and a list of signs of those coefficients. Of course in case of GF(2) coefficient $x_i = -x_i$. However, negative coefficients make sense in case of some types of LFSRs, as `Polynomial` objects are used to create other objects, of type of `Lfsr`.

Below you can see an example of how to create a `Polynomial` object representing the polynomial of $x^{16} + x^5 + x^2 + x^0$:

```
p1 = Polynomial ( [16, 5, 2, 0] )
p2 = Polynomial ( 0b10000000000100101 )
p2 = Polynomial ( 0x10025 )
```

`Polynomial` class includes also a couple of static methods, especially useful to search for primitive polynomials and other ones discussed in the next part of this chapter.

2.1 Polynomial object methods

`Polynomial_object.__str__()`

`Polynomial` objects are convertible to strings.

```
p1 = Polynomial ( [16, 5, 2, 0] )
print(p1)
# >>> [16, 5, 2, 0]
```

`Polynomial_object.__hash__()`

`Polynomial` objects are hashable. Can be used as dictionary keys.:

```
p1 = Polynomial ( [16, 5, 2, 0] )
d = {}
d[p1] = "p1 value"
```

`Polynomial_object.copy()`

Returns a deep copy of the `Polynomial` object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
p2 = p1.copy()
print(p1 == p2)
```

```
# >>> True
```

`Polynomial_object.derivativeGF2()`

Returns symbolic derivative Polynomial.

```
p1 = Polynomial ( [16, 15, 2, 1, 0] )
print(p1.derivativeGF2())
# >>> [14, 0]
p2 = Polynomial ( [16, 14, 5, 2, 0] )
p3 = p2.derivativeGF2()
print(p3)
# >>> [4]
```

`Polynomial_object.getBalancing()`

Returns a difference between distances of furthest and closest coefficients.

```
p1 = Polynomial ( [16, 10, 2, 0] )
#     distances:        6   8   2
#     furthest:             8
#     closest:                  2
#   furthest-closest:  8-2 = 6
p1.getBalancing()
# >>> 6
```

`Polynomial_object.getCoefficients()`

Returns a reference to the sorted list of (unsigned) coefficients.

```
p1 = Polynomial ( [16, 2, -5, 0] )
print(p1.getCoefficients())
# >>> [16, 5, 2, 0]
```

`Polynomial_object.getCoefficientsCount()`

Returns count of the `Polynomial` object coefficients.

```
p1 = Polynomial ( [16, 5, 2, 0] )
coeffscount1 = p1.getCoefficientsCount()
print(coeffscount1)
# >>> 4
```

`Polynomial_object.getDegree()`

Returns degree of the `Polynomial` object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
deg1 = p1.getDegree()
print(deg1)
# >>> 16
```

`Polynomial_object.getDifferentTapCount(AnotherPolynomial)`

Imagine, that the `Polynomial_object` is a characteristic polynomial of a Ring Generator. Then this method compares the `Polynomial_object` with another polynomial (also being a characteristic one of a Ring Generator) and returns a number of NOT matching taps. Tap direction (given by coefficient sign) does not matter.

```
p1 = Polynomial ( [16, -5, 2, 0] )
p2 = Polynomial ( [16, 5, 2, 0] )
p3 = Polynomial ( [16, 5, 1, 0] )
p4 = Polynomial ( [16, 6, 0] )
p1.getDifferentTapCount(p2)
# >>> 0
p1.getDifferentTapCount(p3)
# >>> 1
p1.getDifferentTapCount(p4)
# >>> 2
p4.getDifferentTapCount(p1)
# >>> 1
```

`Polynomial_object.getMinDistance()`

Returns a distance between closest Polynomial's coefficients.

```
p1 = Polynomial ( [16, 5, 2, 0] )
#     distances:     11   3   2
p1.getMinDistance()
# >>> 2
```

`Polynomial_object.getReciprocal()`

Returns a new, reciprocal Polynomial object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
print(p1.getReciprocal())
# >>> [16, 14, 11, 0]
```

`Polynomial_object.getSignedCoefficients()`

Returns sorted list of signed coefficients.

```
p1 = Polynomial ( [16, 2, -5, 0] )
print(p1.getSignedCoefficients())
# >>> [16, -5, 2, 0]
```

`Polynomial_object.getSigns()`

Returns signs of all sorted coefficients (as a list of 1s and -1s).

```
p1 = Polynomial ( [16, -5, 2, 0] )
print(p1.getSigns())
# >>> [1, -1, 1, 1]
```

`Polynomial_object.isLayoutFriendly()`

Returns True if a Ring Generator, based on the Polynomial_object, is layout friendly. It checks if the minimum distance between successive coefficients is at least 2.

```
p1 = Polynomial ( [16, 15, 2, 1, 0] )
p1.isLayoutFriendly()
# >>> False
p2 = Polynomial ( [16, 14, 5, 2, 0] )
p2.isLayoutFriendly()
# >>> True
```

`Polynomial_object.isPrimitive()`

Returns True if the given polynomial is primitive over GF(2). All coefficients are considered to be positive. Note, that the first call of this method may take more time than usual, because of prime dividers database loading. This methods bases on fast simulation of LFSRs described in [1].

```
p1 = Polynomial ( [16, 5, 2, 0] )
p1.isPrimitive()
# >>> False
p2 = Polynomial ( [4, 1, 0] )
p2.isPrimitive()
# >>> True
```

`Polynomial_object.iterateThroughSigns()`

This is generator method. Each time yields new `Polynomial` object with other combinations of coefficient signs. Note, that the highest and lowest coefficients are untouched. All-positive and all-negative combinations are also not yielded.

```
p1 = Polynomial ( [16, -5, 2, 1, 0] )
for pi in p1.iterateThroughSigns(): print(pi)
# >>> [16, -5, 2, 1, 0]
# >>> [16, 5, -2, 1, 0]
# >>> [16, -5, -2, 1, 0]
# >>> [16, 5, 2, -1, 0]
# >>> [16, -5, 2, -1, 0]
# >>> [16, 5, -2, -1, 0]
```

`Polynomial_object.nextPrimitive(Silent=True)`

Tries to find next polynomial which is primitive over GF(2). Returns True if found, otherwise returns False. if `Silent` argument is False, then searching process is shown in the terminal.

```
p1 = Polynomial ( [16, 15, 2, 1, 0] )
p1.nextPrimitive()
# >>> True
print(p1)
# >>> [16, 12, 3, 1, 0]
p1.nextPrimitive()
# >>> True
print(p1)
# >>> [16, 6, 4, 1, 0]
```

`Polynomial_object.printFullInfo()`

Prints (also to the transcript in testcase mode) full info about the Polynomial_object. See the example below:

```
p1 = Polynomial ( [16, 5, 2, 0] )
p1.printFullInfo()
#
# ----------------------------
# Polynomial  deg=16, bal=9
# ----------------------------
#
# Degree             :  16
# Coefficients count:  4
# Hex with degree    :  10(2)25
# Hex without degree:  25
# Balancing          :  9
# Is layout-friendly:  True
# Coefficients       :  [16, 5, 2, 0]
```

`Polynomial_object.setStartingPointForIterator(StartingPolynomial)`

`Polynomial` object may be used as generators, to iterate through all possible polynomials with respect to some requirements (see createPolynomial() method). This one is used to set starting point for iterator. See the example below. Note, that `StartingPolynomial` may be another `Polynomial` object, or a list of coefficients. The starting polynomial is checked to have the same degree and coefficients count as the Polynomial_object.

```
p1 = Polynomial ( [6,1,0] )
for pi in p1: print(pi)
# >>> [6, 1, 0]
# >>> [6, 2, 0]
# >>> [6, 3, 0]
# >>> [6, 4, 0]
# >>> [6, 5, 0]
p1.setStartingPointForIterator( [6,3,0] )
for pi in p1: print(pi)
# >>> [6, 3, 0]
# >>> [6, 4, 0]
# >>> [6, 5, 0]
p1.setStartingPointForIterator( Polynomial([6,4,0]) )
for pi in p1: print(pi)
# >>> [6, 4, 0]
# >>> [6, 5, 0]
```

`Polynomial_object.toBitarray()`

Returns a bitarray object representing the Polynomial.

```
p1 = Polynomial ( [16, 5, 2, 0] )
p1.toBitarray()
# >>> bitarray('10100100000000001')
```

`Polynomial_object.toHexString(IncludeDegree=True, shorten=True)`

Returns a string of hexadecimal characters describing the Polynomial_object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
p1.toHexString()
# >>> '10(2)25'
p1.toHexString(IncludeDegree=False)
# >>> '25'
p1.toHexString(shorten=False)
# >>> '10025'
```

`Polynomial_object.toInt()`

Returns an integer representing the Polynomial object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
bin(p1.toInt())
# >>> '0b10000000000100101'
```

`Polynomial_object.toMarkKassabStr()`

Returns a string used by Mark Kassab's C++ code to add a polynomial to the internal database.

```
p1 = Polynomial ( [16, -5, 2, 0] )
p1.toMarkKassabStr()
# >>> 'add_polynomial(16, 5, 2, 0);'
```

2.2 Static Polynomial methods

Chapter 3

class Lfsr

`Lfsr` is an object type allowing to simulate and analyze any type of Linear Feedback Shift Register. Simulations are performed using `bitarray` objects, where `bitarray_object[N]` holds value of Flip-Flop having index N. `Lfsr` objects are always simulated assuming, that data is shifted from higher to lower indexed flip-flop (from FF[1] to FF[0], from FF[2] to FF[1] and so on).

3.1 Lfsr types

3.1.1 Fibonacci

Look at the Figure 3.1. It shows an example of Fibonacci LFSR implementing the polynomial of $x^8 + x^6 + x^5 + x^2 + 1$. There are couple of ways to create such `Lfsr` object:

```
# using existing Polynomial object:
p1 = Polynomial([8,6,5,2,0])
lfsr1 = Lfsr(p1, FIBONACCI)
# using Polynomial created in place:
lfsr1 = Lfsr(Polynomial([8,6,5,2,0]), FIBONACCI)
# using coefficients list:
lfsr1 = Lfsr([8,6,5,2,0], FIBONACCI)
```

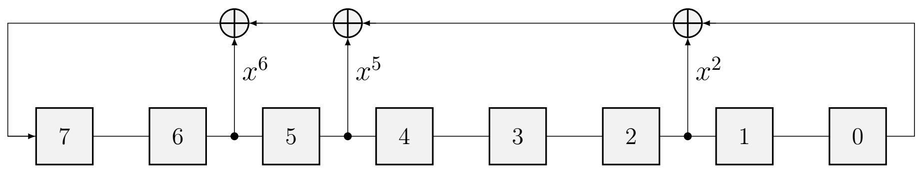

Figure 3.1: Fibonacci LFSR implementing polynomial $x^8 + x^6 + x^5 + x^2 + 1$.

3.1.2 Galois

Figure 3.1 shows an example of Galois LFSR implementing the polynomial of $x^8+x^6+x^5+x^2+1$. There are couple of ways to create such `Lfsr` object:

```
# using existing Polynomial object:
p1 = Polynomial([8,6,5,2,0])
lfsr1 = Lfsr(p1, GALOIS)
# using Polynomial created in place:
lfsr1 = Lfsr(Polynomial([8,6,5,2,0]), GALOIS)
# using coefficients list:
lfsr1 = Lfsr([8,6,5,2,0], GALOIS)
```

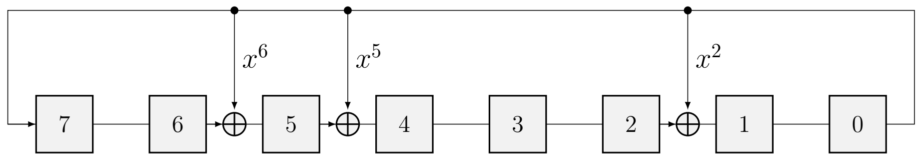

Figure 3.2: Galois LFSR implementing polynomial $x^8 + x^6 + x^5 + x^2 + 1$.

3.1.3 Ring Generator

Ring generator is a structure discussed in [1]. Example of a Ring Generator is shown in the Figure 3.3 implementing the polynomial $x^8 + x^6 + x^5 + x^2 + 1$. Ways to create such object are:

```
# using existing Polynomial object:
p1 = Polynomial([8,6,5,2,0])
lfsr1 = Lfsr(p1, RING_GENERATOR)
# using Polynomial created in place:
lfsr1 = Lfsr(Polynomial([8,6,5,2,0]), RING_GENERATOR)
# using coefficients list:
lfsr1 = Lfsr([8,6,5,2,0], RING_GENERATOR)
```

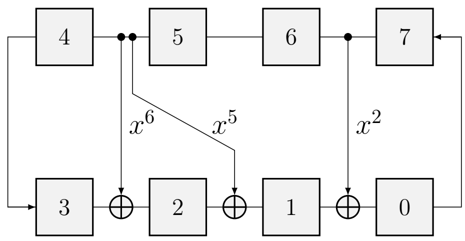

Figure 3.3: Ring Generator implementing polynomial $x^8 + x^6 + x^5 + x^2 + 1$.

3.1.4 Ring with manually specified taps

If you wish to create a LFSR specifying a taps by hand, then choose *Ring with manually specified taps*. To create such object you need to specify a size of Lfsr (flip-flop count) and a list of tap definitions. Each tap is defined as another list: `[source_ff_index, destination_ff_index]`. For better understanding consider the example from Figure 3.4. You can see there the list of 3 taps: `[[4,7], [8,2], [9,0]]`. The first tap is `[4,7]` which is considered as *from output of 4th flip-flop to a XOR gate at 7th flip-flop input*. So be careful and remember: *from <source> OUTPUT to the XOR at <destination> INPUT*. You can create an `Lfsr` object implementing the structure shown in the Figure 3.4 that way:

```
lfsr1 = Lfsr(16, RING_WITH_SPECIFIED_TAPS, [[4,7], [8,2], [9,0]])
#             ->||<- FFs count              |<------ taps ------>|
```

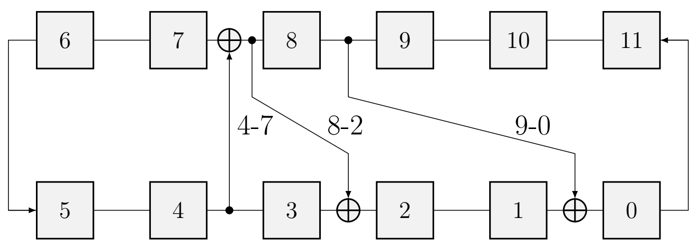

Figure 3.4: Ring with manually specified taps `[[4,7], [8,2], [9-0]]`

3.1.5 Hybrid Ring Generator

Hybrid Ring Generator is very similar to Ring Generator. The only difference is, that taps direction is configurable. If the corresponding coefficient of polynomial *(note: this polynomial is NOT the characteristic one!)* is positive, then tap direction is down, as in case of Ring Generator. When the corresponding coefficient is negative, then tap direction is up. Look at the example shown in the Figure 3.5. To create an `Lfsr` object representing the Hybrid Ring Generator mentioned in the example, use such code:

```
# using existing Polynomial object:
p1 = Polynomial([8,6,-5,-2,0])
lfsr1 = Lfsr(p1, HYBRID_RING)
# using Polynomial created in place:
lfsr1 = Lfsr(Polynomial([8,6,-5,-2,0]), HYBRID_RING)
# using coefficients list:
lfsr1 = Lfsr([8,6,-5,-2,0], HYBRID_RING)
```

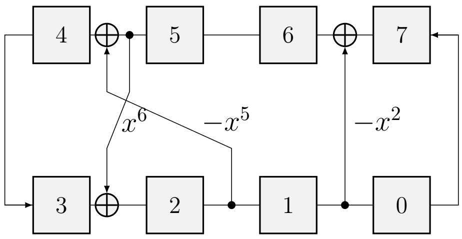

Figure 3.5: Hybrid Ring Generator implementing polynomial $x^8 + x^6 - x^5 - x^2 + 1$.

3.1.6 Tiger Ring Generator

Tiger Ring Generator is a special case of Hybrid Ring Generator. In case of Tiger Ring taps are directed up-down-up-down etc. The most right tap is directed up, as shown in the example at Figure 3.6. The advantage of Tiger Ring Generator over Hybrid Ring Generator is, that signs of polynomial coefficients do not matter. Look at the code used to implement the Tiger Ring from the example:

```
# using existing Polynomial object:
p1 = Polynomial([8,6,5,2,0])
lfsr1 = Lfsr(p1, TIGER_RING)
# using Polynomial created in place:
lfsr1 = Lfsr(Polynomial([8,6,5,2,0]), TIGER_RING)
# using coefficients list:
lfsr1 = Lfsr([8,6,5,2,0], TIGER_RING)
```

Consider, that the polynomial coefficients in the code above are positive. As mentioned, their signs do not matter and the same result can also be obtained using such code:

```
lfsr1 = Lfsr([8,-6,5,-2,0], TIGER_RING)
lfsr2 = Lfsr([8,-6,-5,-2,0], TIGER_RING)
lfsr3 = Lfsr([8,6,-5,2,0], TIGER_RING)
# ...etc.
```

As in case of Hybrid Ring Generators, the polynomial used to implement Tiger Rings is NOT their *characteristic* one.

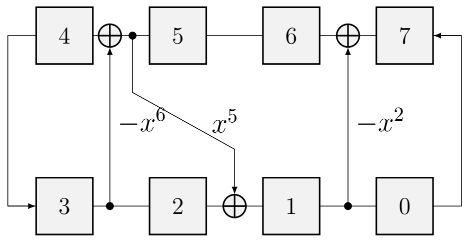

Figure 3.6: Tiger Ring Generator implementing polynomial $x^8 - x^6 + x^5 - x^2 + 1$.

3.2 Lfsr object methods

`Lfsr_object.__str__()`

It returns a string containing binary value of the `Lfsr` object (left MSb).

```
lfsr1 = Lfsr([4,1,0], GALOIS)
str(lfsr1)
# >>> '0001'
lfsr1.next()
str(lfsr1)
# >>> '1001'
```

`Lfsr_object.clear()`

Removes the *fast simulation array* of the `Lfsr` object, if exists. Use this method to clear memory if fast simulation of the `Lfsr` object is no longer necessary.

```
lfsr1 = Lfsr([4,1,0], GALOIS)
# Check if the lfsr1 generates M-Sequence. That check requires
# the Fast Simulation Array to be created, so the array
# is built in the background.
lfsr1.isMaximum()
# >>> True
lfsr.clear()
```

`Lfsr_object.createPhaseShifter(OutputCount, MinimumSeparation=100, MaxXorInputs=3, MinXorInputs=1, FirstXor=None)`

Returns a `PhaseShifter` object. Needs some parameters to calculate phase shifting XORs:

- `OutputCount` - how many outputs the Phase Shifter to have,
- `MinimumSeparation` - minimum separation between Phase Shifter outputs,
- `MaxXorInputs` - how many inputs the largest XOR gate may have,
- `MinXorInputs` - how many inputs the smallest XOR gate may have,
- `FirstXor` - you can specify a list of Lfsr output bits indexes making the first Phase Shifter XOR gate. If `None` (not specified), then the first output of the Phase SHifter is considered the Lfsr FF[0].

```
lfsr1 = Lfsr([32, 28, 23, 18, 12, 6, 0], GALOIS)
lfsr1.createPhaseShifter(OutputCount=48, MinimumSeparation=40, \
  MaxXorInputs=10, MinXorInputs=1)
# >>> PhaseShifter(Lfsr([32, 28, 23, 18, 12, 6, 0], LfsrType.Galois), 48)
```

`Lfsr_object.getDual()`

Returns a reference to taps list of the LFSR.

```
lfsr1 = Lfsr([4,1,0], RING_GENERATOR)
lfsr1.getTaps()
# >>> [[3, 3]]
```

`Lfsr_object.getPhaseShiftIndexes(ListOfXoredOutputs : list, DelayedBy : int)`

Given a sequence obtained by XORing outputs of specified flip-flops. This method returns a list of other flip-flop indexes, at XOR of which the sequence is delayed by specified clock cycles than the one mentioned in the above assumption.

```
lfsr1 = Lfsr([4,1,0], GALOIS)
# How to obtain a sequence observed at FF[0] delayed by 2 cycles?
lfsr1.getPhaseShiftIndexes([0], 2)
# >>> [2]
# ...so at FF[2] we an observe the same sequence as at FF[0] delayed
# by 2 cycles.
# How to obtain a sequence observed at XOR(FF[3], FF[0]) delayed
# by 5 cycles?
lfsr1.getPhaseShiftIndexes([3,0], 5)
# >>> [1, 3]
# ... so at XOR(FF[1], FF[3]) we can observe the same sequence as at
# XOR(FF[3], FF[0]) delayed by 5 cycles.
```

`Lfsr_object.getPeriod()`

Does the standard simulation and finds a period of the `Lfsr`May take much time! Consider using `isMaximum()` method if possible.

```
lfsr1 = Lfsr([4,1,0], GALOIS)
lfsr1.getPeriod()
# >>> 15
```

`Lfsr_object.getSize()`

Returns a size (flip-flops count) of the `Lfsr` object.

```
lfsr1 = Lfsr([4,1,0], GALOIS)
lfsr1.getSize()
# >>> 4
```

`Lfsr_object.getValue()`

Returns a reference to actual value of `Lfsr` (bitarray).

```
lfsr1 = Lfsr([4,1,0], GALOIS)
lfsr1.getValue()
# >>> bitarray('0001')
lfsr1.next()
lfsr1.getValue()
# >>> bitarray('1001')
```

`Lfsr_object.getMSequence(BitIndex=0, Reset=True)`

Returns a bitarray object conteining the M-Sequence observed at selected bit.

- `BitIndex` - at which flip-flop the sequence to observe,
- *Reset* - if True then the `.reset()` method is called before simulation.

```
lfsr1 = Lfsr([4,1,0], GALOIS)
lfsr1.getMSequence()
for value in values: print(value)
# >>> bitarray('111101011001000')
```

`Lfsr_object.getSequence(BitIndex=0, Reset=True, Length=0)`

Returns a bitarray object conteining a bit sequence observed at selected bit.

- `BitIndex` - at which flip-flop the sequence to observe,
- *Reset* - if True then the `.reset()` method is called before simulation,
- `Length` - length of the requested sequence.0 means M-Sequence.

```
lfsr1 = Lfsr([4,1,0], GALOIS)
lfsr1.getSequence(Length=6)
for value in values: print(value)
# >>> bitarray('111101')
```

`Lfsr_object.getValues(n=0, step=1, reset=True)`

Does simulation of the `Lfsr` and returns a list of values.

- `n` - how many values to return. Default os 0 meaning we want all values to get,
- `step` - how many clock cycles per step. If *step*=N then consecutive values are obtained every N clock cycles,
- *reset* - if True then the `.reset()` method is called before simulation.

```
lfsr1 = Lfsr([4,1,0], GALOIS)
values = lfsr1.getValues()
for value in values: print(value)
# >>> bitarray('1000')
# >>> bitarray('1001')
# >>> bitarray('1011')
# ...
# >>> bitarray('0100')
```

`Lfsr_object.isMaximum()`

Returns True if the `Lfsr` generates a M-Sequence. Uses fast simulation method [1] and checks also subcycles.

```
lfsr1 = Lfsr([4,1,0], RING_GENERATOR)
lfsr1.isMaximum()
# >>> True
```

`Lfsr_object.next(steps=1)`

Calculates (and also returns a reference to) the next value of the `Lfsr`. This is the core method of LFSR simulation flow. If the specified `steps` > 1, then it engages Fast Simulation method [1].

```
lfsr1 = Lfsr([4,1,0], GALOIS)
lfsr1.getValue()
# >>> bitarray('0001')
lfsr1.next()
# >>> bitarray('1001')
lfsr1.next(2)
# >>> bitarray('1111')
```

`Lfsr_object.printFastSimArray()`

Prints the array used for fast simulation [1].

```
lfsr1 = Lfsr([4,1,0], GALOIS)
lfsr1.printFastSimArray()
# >>> 1001    0001    0010    0100
# >>> 1101    1001    0001    0010
# >>> 1110    1111    1101    1001
# >>> 1011    0101    1010    0111
```

`Lfsr_object.printValues(n=0, step=1, reset=True)`

Does the same as `Lfsr_object.getValues()`, but prints (to the screen and to the transcript as well) the result in human-readable form.

```
lfsr1 = Lfsr([4,1,0], GALOIS)
lfsr1.printValues()
# >>> 0001
# >>> 1001
# >>> 1101
# >>> ...
# >>> 0010
```

`Lfsr_object.reset()`

Sets 0s to all `Lfsr` flip-flops, besides FF[0] which is set to 1. Also returns a reference to actual value of the `Lfsr`.

```
lfsr1 = Lfsr([4,1,0], GALOIS)
lfsr1.getValue()
# >>> bitarray('0001')
lfsr1.next()
# >>> bitarray('1001')
lfsr1.reset()
# >>> bitarray('0001')
```

`Lfsr_object.reverseTap(TapIndex)`

Reverses tap in case of `Lfsr` object having taps, like Ring generator etc. Such fsr object have list of taps, so this method takes a tap index telling them which one to revert. Tap reversing is used to obtain a dual Lfsr, for example.

```python
lfsr1 = Lfsr([64, 15, 7, 0], RING_GENERATOR)
lfsr1.getTaps()
# >>> [[60, 2], [56, 6]]
# let's revert the second tap: [56, 6]:
lfsr1.reverseTap(1)
# >>> True
# ...True means the tap index and the Lfsr type are correct.
lfsr1.getTaps()
# >>> [[60, 2], [7, 55]]
```

`Lfsr_object.simulateForDataString(Sequence, InjectionAtBit=0, StartValue=None)`

Performs a simulation for the Lfsr object having one injector. Step count (number of clock cycles) is equal to the length of a given `Sequence`. Returns the last `Lfsr` value.

- `Sequence` - any iterable object, whose consecutive values are convertible to `bool`,
- `InjectionAtBit` - index of flip-flop at which input the injector is placed,
- *StartValue* - seed of the LFSR.

```python
lfsr1 = Lfsr([4,1,0], GALOIS)
lfsr1.simulateForDataString('110010')
# >>> bitarray('1001')
```

`Lfsr_object.toVerilog(ModuleName, InjectorIndexesList=[])`

Returns a string containing Verilog description of the `Lfsr` object.

- `ModuleName` - name of the Verilog module,
- `InjectorIndexesList` - a list containing indexes of flip-flops at which input injectors have to be placed.

```python
lfsr1 = Lfsr([4,1,0], GALOIS)
print(lfsr1.toVerilog("MyModule", [0,2]))
# >>> module MyModule (
# >>>   input wire clk,
# >>>   input wire enable,
# >>>   input wire reset,
# >>>   input wire [1:0] injectors,
# >>>   output reg [3:0] O
# >>> );
# >>>
# >>> always @ (posedge clk or posedge reset) begin
# >>>   if (reset) begin
# >>>     O <= 4'd0;
# >>>   end else begin
# >>>     if (enable) begin
# >>>       O[0] <= O[1] ^ O[0] ^ injectors[0];
# >>>       O[1] <= O[2];
# >>>       O[2] <= O[3] ^ injectors[1];
# >>>       O[3] <= O[0];
# >>>     end
# >>>   end
# >>> end
```

```
# >>>
# >>> endmodule
```

3.3 Lfsr static methods

`Lfsr.checkMaximum(LfsrsList, n=0, SerialChunkSize=20, ReturnAlsoNotTested=False)`

Takes a list of Lfsr objects and checks each one if can produce M-Sequence. Returns a list containing only maximum ones.

- `LfsrsList` - list of `Lfsr` objects,
- `n` - how many maximum Lfsrs you need. If the n is achieved, breaks the simulation. 0 means *no limit*,
- *SerialChunkSize* - simulation are performed using multithreading. This value means how many Lfsrs can be simulated in series per one thread,
- `ReturnAlsoNotTested` - using this argument makes sense with n > 0. if True, then it returns a list containing two other lists: *[MaximumLfsrs, NotTestedLfsrs]*

```
lfsr1 = Lfsr([5,1,0], GALOIS)
lfsr2 = Lfsr([5,2,0], GALOIS)
lfsr3 = Lfsr([5,3,0], GALOIS)
Lfsr.checkMaximum([lfsr1, lfsr2, lfsr3])
# >>> [Lfsr([5, 2, 0], LfsrType.Galois),
# >>> Lfsr([5, 3, 0], LfsrType.Galois)]
```

Bibliography

[1] Nilanjan Mukherjee, Janusz Rajski, Grzegorz Mrugalski, Artur Pogiel, and Jerzy Tyszer. Ring generator: An ultimate linear feedback shift register. *Computer*, 44(6):64–71, 2011.

Index