Research Shell

Maciej Trawka

January 20, 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.

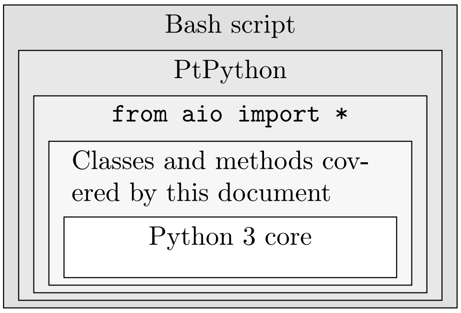

Figure 1.1: Research Shellarchitecture.

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:
```

```python
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

`Polynomial.checkPrimitives(Candidates, n=0, Silent=True)`

Takes a list of `Polynomial` objects and returns a filtered list, containing only primitives. Uses multithreading.

- `Candidates` - list of polynomials to check,
- `n` - if *n>0* then stops once n primitives found,
- `Silent` - if `False` then prints more info during searching. Otherwise (default) prints only a progress bar.

```
Polynomial.checkPrimitives([ [4,1,0], [4,2,0], [4,3,0] ])
# >>> [Polynomial([4, 3, 0]), Polynomial([4, 1, 0])]
```

`Polynomial.createPolynomial(PolynomialDegree, PolynomialCoefficientsCount, PolynomialBalancing=0, LayoutFriendly=False, MinDistance=0)`

Returns a first `Polynomial` object having a specified properties. Such object may be used then as iterator, to go through all polynomials having the same properties.

- `PolynomialDegree` - degree of polynomial,
- `PolynomialCoefficientsCount` - coefficients count, including degree and 0,
- `PolynomialBalancing` - maximum allowed balancing. '0' means 'no limit'. See `Polynomial.getBalancing` on page 5 for more details,
- `LayoutFriendly` - wheather the polynomial must be layout-friendly or not. See `Polynomial.isLayoutFriendly` on page 6,
- `MinDistance` - minimum required distance between successive coefficients.

```
p = Polynomial.createPolynomial(16, 4, MinDistance=4)
p
# >>> Polynomial([16, 8, 4, 0])
for pi in p: print(pi)
# >>> [16, 8, 4, 0]
# >>> [16, 9, 4, 0]
# >>> [16, 10, 4, 0]
# >>> [16, 11, 4, 0]
# ...
# >>> [16, 12, 8, 0]
```

`Polynomial.decodeUsingBerlekampMassey(Sequence)`

Simply realizes a Berlekamp-Massey algorithm to decode a characteristic polynomial basing on a sequence generated by an LFSR.

- `Sequence` - any iterable object, like bitarray, string and so on, or pointer to `Lfsr` object.

```
# Decode a polynomial from a sequence:
Polynomial.decodeUsingBerlekampMassey("110001")
# >>> Polynomial([4, 3, 0])
# Decode a characteristic polynomial of an LFSR:
lfsr = Lfr([4,1,0], GALOIS)
Polynomial.decodeUsingBerlekampMassey(lfsr)
# >>> Polynomial([4, 1, 0])
```

2.3 Static Polynomial methods to search for primitives

Below are listed static `Polynomial` method intended to search for primitive ones. Each method has three versions:

1. `list<*>Primitives` - returning a list of primitive polynomials,
2. `print<*>Primitives` - printing found polynomials in a human-readable form,

3. `first<*>Primitive` - returning a first found polynomial (or `None` if nothing found).

All those methods have the same arguments, so that below are listed only those having a name starting from `list<*>`. Consider also using *interactive menu*, when working in interactive shell. It makes easier to configure below described searching methods (see page 34).

`Polynomial.listAllPrimitivesHavingSpecifiedCoeffs(CoefficientList, CoeffsCount=0, DontTouchBounds=True, OddOnly=True)`

Takes a list of all available coefficients. Tries to search for primitive polynomials having coefficients included in the given list.

- `CoefficientList` - list of all available coefficients,
- `CoeffsCount` - specify here how many coefficients must have the found primitive polynomials (0 = 'any'),
- `DontTouchBounds` - if `True` then the highest and lowest coefficients (degree and 0) must always be included,
- `OddOnly` - limit searching only to such candidates having odd coefficient count.

```
# Let's say we have a ring generator, having gated taps on such
# positions, so that the most rich polynomial implemented by thus
# ring generator is:
# [16, 14, 13, 12, 10, 9, 8, 5, 4, 2, 1, 0]
# Let's find how many primitive polynomials may we implement
# using such register:
len(Polynomial.listAllPrimitivesHavingSpecifiedCoeffs(\
  [16, 14, 13, 12, 10, 9, 8, 5, 4, 2, 1, 0]))
# >>> 49
# Let's determine also how many pentanomials may we implement
# using our register:
len(Polynomial.listAllPrimitivesHavingSpecifiedCoeffs(\
  [16, 14, 13, 12, 10, 9, 8, 5, 4, 2, 1, 0], 5))
# >>> 7
```

`Polynomial.listDensePrimitives(PolynomialDegree, m=0, Silent=True, StartingPolynomial=None)`

Search for dense primitives [2].

- `PolynomialDegree` - degree of polynomial,
- `n` - if *n>0* then stops once n primitives found,
- `Silent` - if `False` then prints more info during searching. Otherwise (default) prints only a progress bar,
- `StartingPolynomial` - you may specify a starting polynomial having the same properties as the ones passed to this method.

`Polynomial.listEveryNTapsPrimitives(PolynomialDegree, EveryN, m=0, Silent=True, StartingPolynomial=Nor`

Search for well balanced (see page 5) polynomials having specified average distance between successive coefficients.

- `PolynomialDegree` - degree of polynomial,
- `EveryN` - average distance between successive coefficients.
- `n` - if *n>0* then stops once n primitives found,
- `Silent` - if `False` then prints more info during searching. Otherwise (default) prints only a progress bar,
- `StartingPolynomial` - you may specify a starting polynomial having the same properties as the ones passed to this method.

`Polynomial.listHybridPrimitives(PolynomialDegree, PolynomialCoefficientsCount, PolynomialBalancing=0, LayoutFriendly=False, MinDistance=0, m=0, NoResultsSkippingIteration=0, StartingPolynomial=None, MinNotMatchingTapsCount=0)`

Search for polynomials which can create maximum length Hybrid LFSRa (see page 14).

- `PolynomialDegree` - degree of polynomial,
- `PolynomialCoefficientsCount` - coefficients count, including degree and 0,
- `PolynomialBalancing` - maximum allowed balancing. '0' means 'no limit'.
  See `Polynomial.getBalancing` on page 5 for more details,
- `LayoutFriendly` - wheather the polynomial must be layout-friendly or not.
  See `Polynomial.isLayoutFriendly` on page 6,
- `MinDistance` - minimum required distance between successive coefficients,
- `n` - if *n>0* then stops once n primitives found,
- `Silent` - if `False` then prints more info during searching. Otherwise (default) prints only a progress bar,
- `NoResultsSkippingIteration` - if specified to be > 0 then it breaks the searching process if no result within the given number, successive iterations,
- `StartingPolynomial` - you may specify a starting polynomial having the same properties as the ones passed to this method,
- `MinNotMatchingTapsCount` - using this parameter you can specify at least how many taps must differ all found polynomials. In other words, if you specify minimum k different taps and you get a result, so take any one of the found polynomials and compare with each one of the rest results - you can consider, than each time at least k coefficients are different.

`Polynomial.listPrimitives(PolynomialDegree, PolynomialCoefficientsCount, PolynomialBalancing=0, LayoutFriendly=False, MinDistance=0, m=0, Silent=True, MaxSetSize=10000, ExcludeList=[], ReturnAlsoAllCandidaes=False, FilteringCallback=None, NoResultsSkippingIteration=0, StartingPolynomial=None)`

The most general method to search for primitive polynomials.

- `PolynomialDegree` - degree of polynomial,
- `PolynomialCoefficientsCount` - coefficients count, including degree and 0,
- `PolynomialBalancing` - maximum allowed balancing. '0' means 'no limit'.
  See `Polynomial.getBalancing` on page 5 for more details,
- `LayoutFriendly` - wheather the polynomial must be layout-friendly or not.
  See `Polynomial.isLayoutFriendly` on page 6,
- `MinDistance` - minimum required distance between successive coefficients,
- `n` - if *n>0* then stops once n primitives found,
- `Silent` - if `False` then prints more info during searching. Otherwise (default) prints only a progress bar,
- `MaxSetSize` - how many polynomials should be tested in parallel in one iteration,
- `ExcludeList` - polynomials included in this list will e skipped from checking,
- `ReturnAlsoAllCandidaes` - if `True` then as a result this method will return list of 2 lists:
  `[ [primitive_polys], [all_checked_polys] ]`,
- `FilteringCallback` - if specified, then will be called for each candidate polynomial:
  `callback(candidate_polynomial)`
  and should return `True` if the given polynomial should be checked, otherwise the given polynomial will be skipped from checking,
- `NoResultsSkippingIteration` - if specified to be > 0 then it breaks the searching process if no result within the given number, successive iterations,

- `StartingPolynomial` - you may specify a starting polynomial having the same properties as the ones passed to this method.

```
Polynomial.listTapsFromTheLeftPrimitives(PolynomialDegree, PolynomialCoefficientsCount, MaxDIstance=3,
m=0, Silent=True)
```

Search for such primitive polynomials having coefficients grouped near the left corner, like `[10,9,8,7,0]`.

- `PolynomialDegree` - degree of polynomial,
- `PolynomialCoefficientsCount` - coefficients count, including degree and 0,
- `MaxDistance` - maximum required distance between successive coefficients grouped near the left corner,
- `n` - if *n>0* then stops once n primitives found,
- `Silent` - if `False` then prints more info during searching. Otherwise (default) prints only a progress bar.

```
Polynomial.listTigerPrimitives(PolynomialDegree, PolynomialCoefficientsCount,
PolynomialBalancing=0, LayoutFriendly=False, MinDistance=0, m=0,
NoResultsSkippingIteration=0, StartingPolynomial=None, MinNotMatchingTapsCount=0)
```

Search for polynomials which can create maximum length Tiger LFSRa (see page 15).

- `PolynomialDegree` - degree of polynomial,
- `PolynomialCoefficientsCount` - coefficients count, including degree and 0,
- `PolynomialBalancing` - maximum allowed balancing. '0' means 'no limit'.
  See `Polynomial.getBalancing` on page 5 for more details,
- `LayoutFriendly` - wheather the polynomial must be layout-friendly or not.
  See `Polynomial.isLayoutFriendly` on page 6,
- `MinDistance` - minimum required distance between successive coefficients,
- `n` - if *n>0* then stops once n primitives found,
- `Silent` - if `False` then prints more info during searching. Otherwise (default) prints only a progress bar,
- `NoResultsSkippingIteration` - if specified to be > 0 then it breaks the searching process if no result within the given number, successive iterations,
- `StartingPolynomial` - you may specify a starting polynomial having the same properties as the ones passed to this method,
- `MinNotMatchingTapsCount` - using this parameter you can specify at least how many taps must differ all found polynomials. In other words, if you specify minimum k different taps and you get a result, so take any one of the found polynomials and compare with each one of the rest results - you can consider, than each time at least k coefficients are different.

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)
```

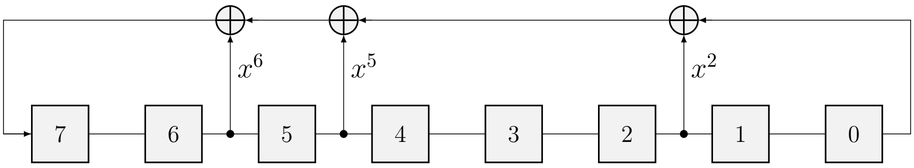

Figure 3.1: Fibonacci LFSR implementing polynomial $x^8 + x^6 + x^5 + x^2 + 1$.

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)
```

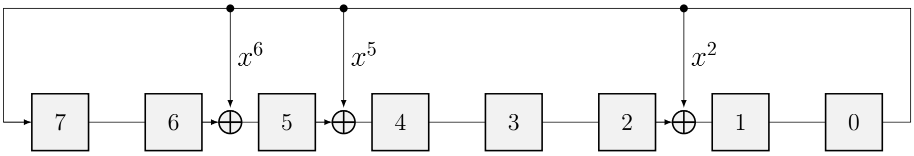

Figure 3.2: Galois LFSR implementing polynomial $x^8 + x^6 + x^5 + x^2 + 1$.

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)
```

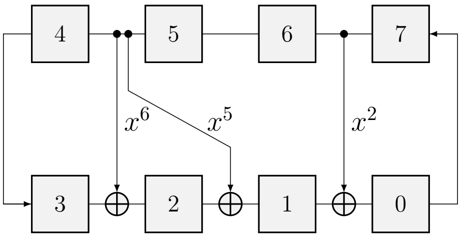

Figure 3.3: Ring Generator implementing polynomial $x^8 + x^6 + x^5 + x^2 + 1$.

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 ------>|
```

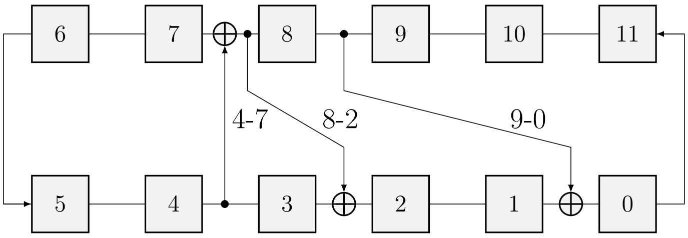

Figure 3.4: Ring with manually specified taps `[[4,7], [8,2], [9-0]]`

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)
```

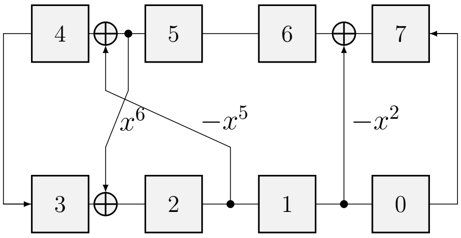

Figure 3.5: Hybrid Ring Generator implementing polynomial $x^8 + x^6 - x^5 - x^2 + 1$.

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.

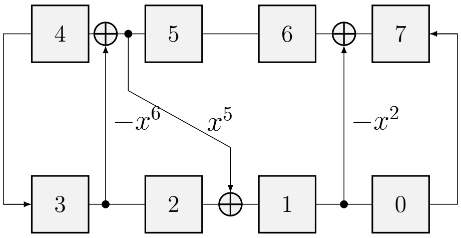

Figure 3.6: Tiger Ring Generator implementing polynomial $x^8 - x^6 + x^5 - x^2 + 1$.

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.

```
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.setValue(Value)`

Sets a new value of the Lfsr. It checks the length and generates error if not match the Lfsr size. Value must be given as a bitarray object or a string containing bit values counted from 0 to (N-1).

`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.

```
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.

```
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)]
```

Chapter 4

Class ProgrammableLfsr

`ProgrammableLfsr` is a class intended to make easy to design and analyse any kind of LFSR with programmable feedback taps. To create a `ProgrammableLfsr` object one need to specify a list of configurable taps. Each configurable tap is described by a dictionary. Each key of such dictionary is a custom name, like "tapOn", or just a number - does not matter - and is used to make analysis reports readable. The corresponding value is a standard tap definition: `[<source_FF_index>, <destination_FF_index>]`. Consider some examples of configurable tap dictionaries:

- `{'Tap1':[3 6]}` - mandatory (constant) tap from output of `FF[3]` to a XOR gate at input of `FF[6]`
- `{'Tap1_OFF':None, 'Tap1_ON':[3,6]}` - gated tap from output of `FF[3]` to a XOR gate at input of `FF[6]`. Consider the value of the first key: `None`, which means *no connection*
- `{'Sw1':[3,6], 'Sw2':[3,8]}` - switched tap: from output of `FF[3]` to a XOR gate at input of `FF[6]`, or from output of `FF[3]` to a XOR gate at input of `FF[8]`
- `{'Sw0':None, 'Sw1':[3,6], 'Sw2':[3,8]}` - like above, but this time there is a way to disable (no connection)
- `{'Sw1':[3,6], 'Sw2':[4,8]}` - switched tap: from output of `FF[3]` to a XOR gate at input of `FF[6]`, or from output of `FF[8]` to a XOR gate at input of `FF[8]`

Consider, that there is no required to make common source or common destination anywhere. See also `ProgrammableLfsrConfig` utility class on page 25

4.1 ProgrammableLfsr object methods

`ProgrammableLfsr_object.__init__(SizeOrProgrammableLfsrConfiguration, TapsList=[], OperateOnTapsOnly=False)`

`ProgrammableLfsr` object initializer.

- `SizeOrProgrammableLfsrConfiguration` - size of the LFSR or a reference to `ProgrammableLfsrConfig` utility object
- `TapsList` - list of Configurable Taps Dictionaries (see description on page 21)
- `OperateOnTapsOnly` - as a result of analysis you may get a list of `Lfsr` objects. However, to save the memory, by setting this parameter to `True`, you may force to operate on tap lists only, so that tap lists will be returned instead of lists of `Lfsr` objects.

```
# we need a 8-bit LFSR having:
# - a mandatory tap from output of FF[7] to a XOR at input of FF[1],
# - a gated tap from output of FF[5] to a XOR at input of FF[3],
ProgLfsr1 = ProgrammableLfsr(8, [
  {'tap1':[7,1]},
  {'tap2_off':None, 'tap2_on':[5,3]}
])
```

`ProgrammableLfsr_object.clear()`

It simply deletes results of Programmable Lfsr analysis to save the memory

`ProgrammableLfsr_object.getAllPosssibleTaps()`

Returns a list containing all connections (taps) possible to implement by the `ProgrammableLfsr` object. See the example on page 22

`ProgrammableLfsr_object.getSize()`

Returns a size (flip-flop count)of the Programmable Lfsr object.

`ProgrammableLfsr_object.getPolynomialsAndLfsrsDictionary(Optimization=False)`

Returns a dictionary, containing all primitive polynomials the `ProgrammableLfsr` object can implement and corresponding `Lfsr`' objects (or tap lists). If `Optimization` is set to `True`, then it tries to optimize all configurations of the programmable LFSR so that as less taps are used, as possible. See the example on page 22

`ProgrammableLfsr_object.getLfsrs(Optimization=True)`

Returns a list containing all maximum Lfsrs implemented by the `ProgrammableLfsr` object. If `Optimization` is set to `True`, then it tries to optimize all configurations of the programmable LFSR so that as less taps are used, as possible. See the example on page 22

`ProgrammableLfsr_object.getPolynomials()`

Returns a list containing all primitive polynomials implemented by the `ProgrammableLfsr` object. See the example on page 22

`ProgrammableLfsr_object.getUsedTaps()`

Returns a list containing used connections (taps) after optimization. See the example on page 22

`ProgrammableLfsr_object.getUnusedTaps()`

Returns a list containing unused connections (taps) after optimization. Removing those connections will not affect the primitive polynomials list implemented by this `ProgrammableLfsr` object. See the example on page 22

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

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

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

See the example on page 22

Example of Programmable Lfsr analysis:

```
# let's examine a 8-bit LFSR having:
# - a gated tap from output of FF[6] to a XOR at input of FF[0],
# - a gated tap from output of FF[5] to a XOR at input of FF[0],
# - a switchable tap tap from output of FF[4] to a XOR at input of FF[3]
#     of from output of FF[3] to a XOR at input of FF[4]
#     with no-connection (disable) option
# - a switchable tap tap from output of FF[2] to a XOR at input of FF[3]
#     of from output of FF[4] to a XOR at input of FF[2]
#     with no-connection (disable) option
```

```
plfsr = ProgrammableLfsr(8, [
{'6-0_off': None, '6-0_on': [6, 0]},
{'5-0_off': None, '5-0_on': [5, 0]},
{'None': None, '4-3': [4, 3], '3-4': [3, 4]},
{'None': None, '2-3': [2, 3], '4-2': [4, 2]}
], OperateOnTapsOnly = True)
plfsr.getPolynomialsAndLfsrsDictionary()
# >>> {Polynomial([8, 5, 3, 2, 0]): [[[6, 0], [3, 4]], [[6, 0], [2, 3]]]}
plfsr.getPolynomialsAndLfsrsDictionary(True)
# >>> {Polynomial([8, 5, 3, 2, 0]): [[6, 0], [2, 3]]}
# As we can see, such Programmable Lfsr can implement 2 maximum Lfsrs,
# but only 1 primitive polynomial.
# We can even get a simple list of primitive polynomials implemented
# by the 'plfsr':
plfsr.getPolynomials()
# >>> [Polynomial([8, 5, 3, 2, 0])]
# ...and a simple list of all possible Lfsrs (tap lists in our case).
# So without optimization (all possible, maximum ones):
plfsr.getLfsrs(False)
# >>> [[[6, 0], [3, 4]], [[6, 0], [2, 3]]]
# ...and with optimization:
plfsr.getLfsrs(True)
# >>> [[[6, 0], [3, 4]]]
# DO you know all possible connections? Let's check it:
plfsr.getAllPosssibleTaps()
# >>> [[6, 0], [5, 0], [4, 3], [3, 4], [2, 3], [4, 2]]
# Now determine, which connections are used:
plfsr..getUsedTaps()
# >>> [[6, 0], [3, 4]]
# ...so it seems only 2 taps are required. So which ones are unused?
plfsr.getUnusedTaps()
# >>> [[5, 0], [4, 3], [2, 3], [4, 2]]
# Finally, let's generate a verilog describingout Programmable Lfsr.
# In addition assume we have injectors at FF[2,1,0]:
print(plfsr.toVerilog("MyProgrammableLfsr", [2,1,0]))
# module MyProgrammableLfsr (
#   input wire clk,
#   input wire enable,
#   input wire reset,
#   input wire [2:0] injectors,
#   output reg [7:0] O
# );
#
# reg T0;
# reg T0_0;
# reg T1;
# reg T1_0;
# reg T2;
# reg T2_3;
# reg T2_4;
# reg T3;
# reg T3_3;
# reg T3_2;
#
#
# always @ (*) begin
#   T0 <= 1'b0;
#   if (config[0] == 1'd1) begin
#     T0 <= O[6];
#   end
# end
#
# always @ (*) begin
#   T1 <= 1'b0;
#   if (config[1] == 1'd1)) begin
#     T1 <= O[5];
#   end
# end

# always @ (*) begin
#   T2 <= 1'b0;
#   if (config[3:2] == 2'd1)) begin
#     T2 <= O[4];
#   end
#   if (config[3:2] == 2'd2)) begin
#     T2 <= O[3];
#   end
# end
#
# always @ (*) begin
#   T3 <= 1'b0;
#   if (config[5:4] == 2'd1)) begin
#     T3 <= O[2];
#   end
#   if (config[5:4] == 2'd2)) begin
#     T3 <= O[4];
#   end
# end
#
# always @ (*) begin
#   T0_0 <= 1'b0;
#   if ((config[0:0] == 1'd1)) begin
#     T0_0 <= T0;
#   end
# end
#
# always @ (*) begin
#   T1_0 <= 1'b0;
#   if ((config[1:1] == 1'd1)) begin
#     T1_0 <= T1;
#   end
# end
#
# always @ (*) begin
#   T2_3 <= 1'b0;
#   if ((config[3:2] == 2'd1)) begin
#     T2_3 <= T2;
#   end
# end
#
# always @ (*) begin
#   T2_4 <= 1'b0;
#   if ((config[3:2] == 2'd2)) begin
#     T2_4 <= T2;
#   end
# end
#
# always @ (*) begin
#   T3_3 <= 1'b0;
#   if ((config[5:4] == 2'd1)) begin
#     T3_3 <= T3;
#   end
# end
#
# always @ (*) begin
#   T3_2 <= 1'b0;
#   if ((config[5:4] == 2'd2)) begin
#     T3_2 <= T3;
#   end
# end
#
# always @ (posedge clk or posedge reset) begin
#   if (reset) begin
#     O <= 8'd0;
#   end else begin
#     if (enable) begin
#       O[0] <= O[1] ^ T0_0 ^ T1_0 ^ injectors[2];
#       O[1] <= O[2] ^ injectors[1];
#       O[2] <= O[3] ^ T3_2 ^ injectors[0];
#       O[3] <= O[4] ^ T2_3 ^ T3_3;
#       O[4] <= O[5] ^ T2_4;
#       O[5] <= O[6];
#       O[6] <= O[7];
#       O[7] <= O[0];
#     end
#   end
# end
#
# endmodule
```

4.2 ProgrammableLfsrConfig utility class

`ProgrammableLfsrConfig` is a utility class making easier to create a configuration list for `ProgrammableLfsr`. Consider the example from page 22. Let's try to make the same `ProgrammableLfsr` object but this time using the `ProgrammableLfsrConfig` utility class:

```
# let's examine a 8-bit LFSR having:
# - a gated tap from output of FF[6] to a XOR at input of FF[0],
# - a gated tap from output of FF[5] to a XOR at input of FF[0],
# - a switchable tap tap from output of FF[4] to a XOR at input of FF[3]
#     of from output of FF[3] to a XOR at input of FF[4]
#     with no-connection (disable) option
# - a switchable tap tap from output of FF[2] to a XOR at input of FF[3]
#     of from output of FF[4] to a XOR at input of FF[2]
#     with no-connection (disable) option
config = ProgrammableLfsrConfiguration(8)
config.addGated(6,0)
# >>> {'6-0_off': None, '6-0_on': [6, 0]}
config.addGated(5,0)
# >>> {'5-0_off': None, '5-0_on': [5, 0]}
config.addSwitched(None,[4,3],[3,4])
# >>> {'None': None, '4-3': [4, 3], '3-4': [3, 4]}
config.addSwitched(None,[2,3],[4,2])
# >>> {'None': None, '2-3': [2, 3], '4-2': [4, 2]}
plfsr = ProgrammableLfsr(config, OperateOnTapsOnly=True)
```

Consider also a way to create all-combinations switch:

`ProgrammableLfsr_object.addAllCombinationsSwitch(SourceList, DestinationList, IncludeNone=False)`

- `SourceList` - list of source FFs indexes
- `DestinationList` - list of destination FFs indexes
- `IncludeNone` - if `True`, then `None` will also be included in the resultant taps switch

```
config = ProgrammableLfsrConfiguration(8)
config.addAllCombinationsSwitch([1,2], [4,5])
# >>> {'1-4': [1, 4], '1-5': [1, 5], '2-4': [2, 4], '2-5': [2, 5]}
config.addAllCombinationsSwitch([1,2], [4,5], True)
# >>> {'None': None, '1-4': [1, 4], '1-5': [1, 5], '2-4': [2, 4], '2-5': [2, 5]}
```

Chapter 5

Class Nlfsr

NLFSR (*Non-Linear Feedback Shift Register*) is very similar to LFSR (*Linear Feedback Shift Register*) with the only difference: feedback taps may be results of a non-linear function of register's flip-flops. Consider the example from Fig. 5.1. We can see there a 10-bit register having two taps: `from FF[4] to a XOR at input of FF[5]` and *from (FF[7] AND FF[3]) to a XOR at input of FF[0])*. This may be strictly implemented using `Nlfsr` object by:

```
MyNLFSR = Nlfsr(10, [ [5, [4]], [0, [3,7]] ])
#         size >|--|<
#                      |<-------- taps -------->|
# tap 1: [5, [4]]    : FF[4] to input of FF[5]
# tap 2: [0. [3,7]]  : (FF[3] AND FF[7]) to input of FF[0]
```

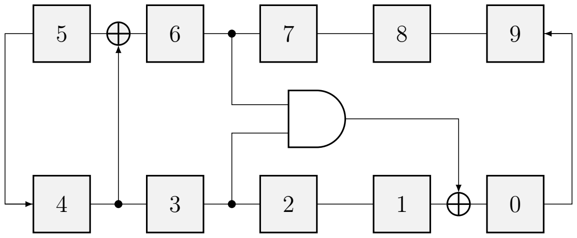

Figure 5.1: Example of Nonlinear Feedback Shift Register.

If is worth nothing, that tap description of `Nlfsr` is different than tap description of `Lfsr`. The difference is, that while each tap is still defined by a list, but the first item is not a source, but destination tap and the second item is a list of ANDed sources. So, each tap of `Nlfsr` looks like:
`[ <destination_FF>, [<source_FF_1>, <source_FF_2>, ...]  ]`.

As you can see, source values may only be ANDed. However, there is still a way to make a tap implementing other nonlinear function. All indexes given in tap list may also be negative numbers. If a source index is negative, then it considers NOT(source). If a destination index is negative, then it considers NAND instead of AND. See example taps description:

- `[2, [-3,4]]` - AND(NOT(FF[3]), FF[4]) to a XOR at input of FF[2]
- `[-2, [3,4]]` - NAND(FF[3], FF[4]) to a XOR at input of FF[2]
- `[-2, [-3,-4]]` - NAND(NOT(FF[3]), NOT(FF[4])) = OR(FF[3], FF[4]) to a XOR at input of FF[2]

Keep in mind, that there is not any *fast simulation* method for NLFSRs. The only way to determine the period of NLFSR is simply simulate it step-by-step. To make it as fast as possible, he Research Shell implements a C++ based subroutine.

5.1 Nlfsr object methods

Not all `Nlfsr` object methods will be described. `Nlfsr` class bases on `Lfsr` so almost all methods are inherited from the LFSR (see page 15).

`Nlfsr_object.__init__(Size, Config=[])`

Object initializer.

- `Size` - size (flip-flop count) of the NLFSR,
- `Config` - list of taps definition (see page 27)

`Nlfsr_object.__str__()`

It returns a string containing binary value of the `Nlfsr` object (left MSb). See also `Lfsr.__str__` on page 16

`Nlfsr_object.getArchitecture()`

Returns a string containing human-readable description of taps.

```
MyNLFSR = Nlfsr(10, [ [5, [4]], [0, [3,7]] ])
print(MyNLFSR.getArchitecture())
# >>>  0 <=  3 AND 7
# >>>  5 <=  4
```

`Nlfsr_object.getFullInfo(Header=False)`

Returns a string containing some information about NLFSR. Dependently on `Header` value, the first line is a header (see the example below). Next lines are human-readable description of taps. Last four lines are simplified equations - update functions - of the NLFSR. First update function - NLFSR as is; the second - complementary; third - reversed, and finally - reversed-complementary.

```
MyNLFSR = Nlfsr(10, [ [5, [4]], [0, [3,7]] ])
print(MyNLFSR.getFullInfo(True))
# >>>  10-bit NLFSRs and taps list and equations:
# >>>  0 <=  3 AND 7
# >>>  5 <=  4
# >>> 2, (4, 8)
# >>> C  2, 4, 8, (4, 8)
# >>> R 8, (6, 2)
# >>> CR 2, 6, 8, (6, 2)
```

`Nlfsr_object.next(steps=1)`

See `Lfsr.next` on page 18.

`Nlfsr_object.printFullInfo()`

Prints the full information (without header) about the NLFSR. Does the same as *Aio.print(NlfsrObject.getFullInfo(Fa*

```
MyNLFSR = Nlfsr(10, [ [5, [4]], [0, [3,7]] ])
MyNLFSR.printFullInfo()
# >>>  0 <=  3 AND 7
# >>>  5 <=  4
# >>> 2, (4, 8)
# >>> C  2, 4, 8, (4, 8)
# >>> R 8, (6, 2)
# >>> CR 2, 6, 8, (6, 2)
```

Chapter 6

Network and distributed computing

6.1 UDP

Research Shell includes a few classes making UDP networking easier: `UdpSender` to send messages and `UdpMonitor` to receive them.

It is well known that UDP packets, dependent on MTU size, does not allow to transmit large amounts of data. However, `UdpSender` and `UdpMonitor` can automatically divide huge data into small fragments and that is realized i the background.

6.1.1 UDP Monitor

This class is used to create monitor objects, which are working using a separate thread and may call a specified function if data arrives or may print the data to the screen. Below the object-related methods are listed:

`UdpMonitor_object.__init__(PortList, Callback=None, BufferSize=4096, ReturnString=False, BindToIp="")`

`UdpMonitor` object initializer.

- `PortList` - port or list of ports to listen,
- `Callback` - function to call when new UDP message is received. If `None` then incoming messages will be printed to the screen,
- `BufferSize` - size of socket buffer. If too small, then some messages may be lost,
- `ReturnString` - messages are returned as `bytes`, but you may force to convert it into `str` by specifying this parameter `True`,
- `BindToIp` - in case of many network interfaces you may bind the listener to a specified one by giving its IP.

```
# Simply look what is received on 1234 port:
UdpMonitor(1234).start()
# Receive and process incoming messages on 1235 and 1236 ports:
def Cbk(args):
  DATA = args[0]
  FROM_IP = args[1]
  FROM_PORT = args[2]
Monitor2 = UdpMonitor([1235, 1236], Callback=Cbk)
Monitor1.start()
```

`UdpMonitor_object.start()`

Starts UDP monitor.

```
# Simply look what is received on 1234 port:
```

```
UdpMonitor(1234).start()
```

`UdpMonitor_object.stop()`

Stops UDP monitor. It is worth nothing, that there is a risk than one message will be received even after `UdpMonitor` atopping!

```
# Simply look what is received on 1234 port during 10s:
Monitor = UdpMonitor(1234)
Monitor.start()
sleep(10)
Monitor.stop()
```

6.1.2 UDP Sender

The `UdpSender` object is intended to make UDP messages sending easier. Its related methods:

`UdpSender_object.__init__(Port=None, DestinationIp=None)`

`UdpMonitor` object initializer.

- `Port` - you may specify a default port if the `UdpSender` object is used to send data on known port,
- `DestinationIp` - you may specify a default destination IP when this `UdpSender` object is used to send data to one destination

```
# Simply send dta to 192.168.0.1:80:
UdpSender(80, "293.268.0.2").send("Hi!")
# Periodically send broadcast messages on 81 port:
Sender = UdpSender(81)
while(1):
  Sender.send("Hi all!")
  sleep(1)
```

`UdpSender_object.send(Data, Ip=None, Port=None)`

Send given data to a specified host.

- `Data` - `bytes` or *str* data to send,
- `Ip` - destination IP, If not specified, tries to use the default one, if still `None` then sends broadcast message,
- `Port` - destination port. If not specified, uses the default one.

```
# Simply send dta to 192.168.0.1:80:
UdpSender(80).send("Hi!", "293.268.0.2")
```

6.2 Distributed computing

The Research Shell is equipped with a utility to make distributed computing simple. The main goal is: *I have some tasks to do. Is any other Research Shell ready to take one of them and execute?*.

So, two classes are available: `RemoteAioNode` - a ready-to-work instance, and `RemoteAIoScheduler` - a task scheduler.

Using the `RemoteAIoScheduler` one can request any number of tasks (by specifying a string containing a Research Shell code) and then the scheduler will try to find ready-to-work nodes, queuing, sending tasks and receiving responses - all in the background.

Let's say you requested a task. Here is what happens then:

1. The `RemoteAIoScheduler` considered, its task queue is not empty. It sends a UDP broadcast message telling others, that there "Hi, I am <IP>:<Port> and I have some task to do".

2. Ready to work nodes respond, telling "Hi, I am <IP>:<Port> and I am ready to work".
3. The scheduler sends queuing tasks to those ready-to-work nodes and moves those task to the end of its task queue.

On the other hand, when a node finishes the requested task:

1. The node receives the result to the scheduler.
2. Once the scheduler receives the response, it removes the task from its task queue.

That simple mechanism makes the task scheduling independent of nodes failures, appearing and disappearing, and so on.

It is also possibility to specify the scheduler address when `RemoteAioNode` is initialized (or later) in case if the node cannot, for any reason, receive scheduler's broadcast messages.

Finally, it is worth nothing that th scheduler may (and it is enabled by default) use the local machine to perform scheduled tasks without any networking.

6.3 RemoteAioNode object

`RemoteAioNode` object creates a node, which is used to process requested tasks.

`RemoteAioNode_object.__init__(Port=3099, CustomServers=[], Enable=True)`

Object initializer.

- `Port` - the port at which the node will listen for broadcast messages from schedulers in the LAN. If - for any reason - there is impossible to configure the node to work on the same port as schedulers, see the `CustomServers` parameter,
- `CustomServers` - list of addresses of schedulers which the node should ping. The reason why to configure this parameters is i.e. if node works on another port than a scheduler, or if the node cannot receive broadcast messages from a scheduler because both are part of different subnets. The `CustomServers` may be a list of addresses, or may be a string containing single IP. In case of list, each address may be a string, containing IP, or a list, containing IP and port. In any case, if no port specified, then it assumes the same port as the one at which the node is listenning,
- `Enable` - if *True* then the node automatically starts its background processes.

```
# Start a simple, standard node
RemoteAioNode()
# Create a simple node, but not start it automatically
Node = RemoteAioNode(Enable=False)
Node.start()
# Start a node and include a specified scheduler address:
RemoteAioNode(CustomServers="10.10.2.100")
# Start a node working at different port than the scheduler:
RemoteAioNode(Port=4000, CustomServers=[["10.10.2.100", 3099]])
```

`RemoteAioNode_object.addCustomServer(Ip, Port)`

Add a custom server address to the node (see the node initializer).

`RemoteAioNode_object.start()`

Start the background process of the node.

`RemoteAioNode_object.stop()`

Stop the background process of the node.

6.4 RemoteAioScheduler object

`RemoteAIoScheduler` is the one object which takes tasks and tries to process them as fast as possible, using local machine (if allowed) and any available `RemoteAioNode`. It has a few methods making task scheduling easier - all are listed below.

`RemoteAioScheduler_object.__init__(Port=3099, Enable=True, LocalExecution=False)`

`RemoteAIoScheduler` object initializer.

- `Port` - the port at which the scheduler sends the broadcasts messages and listens for incoming packets,
- `Enable` - if `True` then it automatically starts its background processes (otherwise it will not process requested tasks)
- `LocalExecution` - whether to use the local machine to process tasks together with other available nodes or not. IT IS NOT RECOMMENDED enabling the local execution when you plan to schedule extensive jobs, especially using multi-threading. It is a high risk that the scheduler may get not enough resources to coordinate all jobs and network communication.

```
# Create a standard scheduler
S = RemoteAioScheduler()
```

`RemoteAioScheduler_object.addTask(Code)`

`RemoteAIoScheduler` object initializer.

- `Code` - a string containing a code to be executed.

This method returns a `RemoteAioTask` object. Look at the example below to see how to use it in your code.

```
S = RemoteAioScheduler()
task = S.addTask("2 * 2")
while not task:
sleep(0.01)
result = task.Response
print(result)
# >>> 4
```

`RemoteAioScheduler_object.map(CodeList, ShowStatus=False)`

This method is used to request for execution of many tasks. It returns an ordered list or results. See the example below:

- `CodeList` - each item of this list is a string containing Python (Research Shell) code to executed,
- `ShowStatus` - if `True`, then it will periodically (1s) visualize mapped tasks, where each task is a colorised: yellow = in progress, green = done.

```
S = RemoteAioScheduler()
codelist = [
  "2*2",
  "3*3",
  "4*4"
]
results = S.map(codelist)
print(results)
# >>> [4, 9, 16]
```

`RemoteAioScheduler_object.mapGenerator(CodeList, ShowStatus=False)`

This is not a method, but a generator. It yields ordered results as soon as possible. See the example below.

- `CodeList` - each item of this list is a string containing Python (Research Shell) code to executed,
- `ShowStatus` - if `True`, then it will periodically (1s) visualize mapped tasks, where each task is a colorised: yellow = in progress, green = done.

```
S = RemoteAioScheduler()
codelist = [
  "[2, 2*2]",
  "[3, 3*3]",
  "[4. 4*4]"
]
for result in S.mapGenerator(codelist):
  print(result)
# >>> [2, 4]
# >>> [3, 9]
# >>> [4, 16]
```

`RemoteAioScheduler_object.mapUnorderedGenerator(CodeList, ShowStatus=False)`

This is not a method, but a generator. It yields UNORDERED results as soon as possible. See the example below.

- `CodeList` - each item of this list is a string containing Python (Research Shell) code to executed,
- `ShowStatus` - if `True`, then it will periodically (1s) visualize mapped tasks, where each task is a colorised: yellow = in progress, green = done.

```
S = RemoteAioScheduler()
codelist = [
  "[2, 2*2]",
  "[3, 3*3]",
  "[4. 4*4]"
]
for result in S.mapUnorderedGenerator(codelist):
  print(result)
# this is the example output. It may differ
# as the mapping is unordered:
# >>> [3, 9]
# >>> [2, 4]
# >>> [4, 16]
```

`RemoteAioScheduler_object.start()`

Start the background process of the scheduler.

`RemoteAioScheduler_object.stop()`

Stop the background process of the scheduler.

Chapter 7

Interactive menu

Bibliography

[1] N. Mukherjee, J. Rajski, G. Mrugalski, A. Pogiel, and J. Tyszer. Ring generator: An ultimate linear feedback shift register. *Computer*, 44(6):64–71, 2011.

[2] G. Mrugalski, M. Mukherjee, J. Rajski, and J. Tyszer. High performance dense ring generators. *IEEE Transactions on Computers*, 55(1):83–87, 2006.

Index