Assignment 1

Welcome to PyRTL[1]

PyRTL Installation:

Installation of PyRTL is very easy. The following is from the manual written by one of the authors and pioneers of this fantastic project, Dr. Tim Sherwood, UC Santa Barbara.

To install this package you can issue this command in terminal in all three major operating systems Windows, Mac and Linux:

```
pip install pyrtl
```

PyRTL is listed in PyPI and can be installed with **pip** or **pip3**. If the above command fails due to insufficient permissions, you may need to do **sudo pip install pyrtl** (to install as superuser) or **pip install --user pyrtl** (to install as a normal user). PyRTL is tested to work with Python 3.8+.

Using the Template (skeleton file)

You will develop your project on the skeleton file which has the necessary parts for your project to work and you build up on that. The files exist in the repository named **lfsr_combinatorial.py** and **lfsr_behavioral.py**.

A Few Words About LSFR

LFSR or Linear Feedback Shift Register is a digital circuit that is used for generation of pseude-random numbers. The generated numbers are not completely random but psuedo-random as this circuit repeats the generated numbers after a very long time. These two links give good information on LFSR:

From Wikipedia.org

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

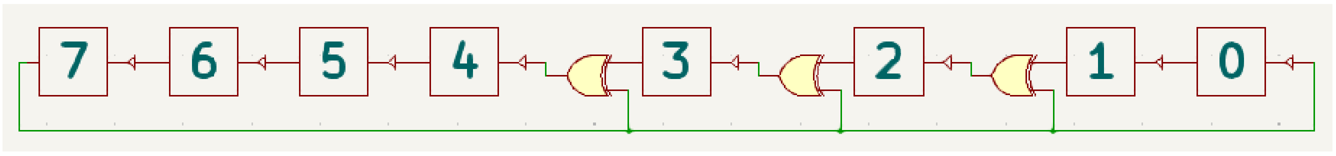
From eetimes.com

https://www.eetimes.com/tutorial-linear-feedback-shift-registers-lfsrs-part-1/

1 This assignment is inspired by CSE x25 - Assignment 1 used in UC Santa Cruz

Assignment 1

Below is a diagram of an LFSR. Note that the boxes with numbers inside them are sub-modules that you have to implement the circuit so that the whole LFSR can be initialized to a predetermined number upon Initialization and keeps on generating pseudo-random numbers in the output of each box. The arrows indicate the data flow. The numbers inside the boxes are the bit position for this 8 bit LFSR.



PART 1

LSFR implementation in Combinatorial Logic

In this part, you will implement the LFSR in a combinatorial way. Although it's not a practical way to do it in PyRTL, because PyRTL is made to use abstraction. You will come out of this part of the lab appreciating the abstraction and see how abstraction makes designing digital circuits much easier.

Create the necessary modules separately and then wire them up together. Your circuit should have means by which the LSFR can be initialized to a predetermined number so that it starts generating pseudo-random numbers starting with that number. The bits of the generated 8-bit numbers are tapped from the output of the boxes. Use the skeleton file **combinatorial.py** and build up your LFSR using that.

PART 2

LSFR implementation in Behavioral Logic

In this part, you will develop the LSFR in a behavioral way. Use the skeleton file **behavioral.py** and build up your LSFR using that.

Helpful hint to test your circuit even without a test-bench

Once you are done with the implementation of both combinatorial and behavioral LFSRs, submit your files along with the lab write up and please include a snapshot of PyRTL terminal waveform viewer which is initialized by the number "A5", in which case the second generated number should be "57". In order to check if your circuit works, it's very common to develop a test bench for the design and this part is a Must, because you want to make sure your circuit is working properly. But here is an easy way to double check the proper working on your circuit.

Once you initialize your circuit with the number A5, your circuit should generate the following number sequence until re-initializing the LFSR by asserting the "init" input while initializing with a number in the sequence and the circuit should generate the numbers to the right of the initializing number in the sequence. Here is the set of outputs results that should be generated by your LFSR:

{A5, 57, AE, 41, 82, 01, 32, 64, C8, 8D, 07, …}

So for example if you initialize the circuit with the number 41 in this sequence, you should have the following number as the first number generated by the circuit:

{82, 01, 32, 64, C8, 8D, 07, …}

This is the reason why LFSR is called a Pseudo-random number generator, because the generated numbers are not truly random and they follow a pattern but since this circuit repeats itself after a long time, it seems that the generated numbers are random. You can generate this sequence yourself for further testing and learning about LFSR. First pick any 8-bit number you like as an initializing number, then follow the logic of the circuit as the bits are getting shifted and Xor'ed and you get the numbers that are expected to be generated in next cycle. You will see the circuit should generate the numbers you came up with, after initializing the circuit with your particular number.

Some notes on the test-benches

A very important part of implementing designs in every project no matter how small it is, is creating a test-bench for that project, because verification is a very very important part of every design. Without verification it is not possible to tell if the design will work. In case of large industrial projects, skipping this step will cost millions of dollars as its risky. So testing your design is a very critical stage of the project. Test-benches are like a test circuit that you design to test the behavior of the circuit to make sure it does what the designer has intended. Once you gain mastery in the hardware definition language you are learning, you will be able to develop your own test-benches to test and verify your designed circuits.

In this assignment, you could do the verification by looking at the waveform generated either by the built-in waveform viewer or the Verilog test-bench produced by PyRTL. The Verilog test-bench created by PyRTL may need some tweaking after optimization and you may need to add some conditions to check the circuit against, and print a success/fail message for the designer to verify the proper functioning of the circuit or catch the bugs in the original code or add stepping and/or timing as a flag to study the behavior of the circuit under test (or DUT Device Under Test). This requires knowledge of Verilog HDL but you can develop your own test-bench using PyRTL without a need for PyRTL generated Verilog code and its corresponding test bench.

There is a Verilog test-bench provided (not generated by PyRTL) that you can run it against your designs and it will give you pass/fail results for your verification. Looking at its code also gives you so clue as far as what is expected from the circuit to do.

How to use the provided Verilog test-bench

For this part you should have Icarus Verilog (iverilog) installed on your computer. You can find the installation instructions here: https://steveicarus.github.io/iverilog/usage/installation.html
or use the easiest way either your Linux distribution’s package manager or if you are on an Ubuntu based Linux distro (i.e. Linux Mint) you can issue this command to get it installed:

```
sudo apt install iverilog
```

Copy the PyRTL generated Verilog files of your implementations (lfsr_combinational.v and lfsr_behavioral.v) in the same folder as lfsr_tb.v (in the testbench folder) and run this command in your terminal:

```
iverilog lfsr_combinational.v lfsr_behavioral.v lfsr_tb.v -o lfsr_tb.out
```

Now iverilog will generate an executable that you can run in your terminal and it will display pass/fail messages once it tests both of your designs at the same time. If your implementation is correct, you should see the following message on your terminal:

```
Structural LFSR passed
Behavioral LFSR passed
```

If you don’t understand anything in the Verilog code for the test-bench and other PyRTL generated Verilog files, don’t worry, you will learn Verilog in the future and once you compare Verilog hardware description language with PyRTL as a modern HLS (High Level Synthesis) tool, you will appreciate many features of PyRTL which makes the task of hardware programming much easier and productive.