

From Nothing to a Ring Oscillator Power Trace

Rahul Jayachandran, Prof. Laurent Michel

July 2023

Resources to learn Vivado and Verilog

Below are the videos, websites and articles I used to learn both Verilog the HDL and Vivado, the development environment. The GitHub repo and the rest of this paper assumes that you, the reader, have all the knowledge contained in these links, as well as basic python skills and an understanding of my research paper.

General Resources

ZYBO 7010 User Manual:

https://digilent.com/reference/_media/reference/programmable-logic/zybo/zybo_rm.pdf

General FPGA Design: <https://www.youtube.com/watch?v=Wt6aske1K10>

Version 1

Verilog Intro: <https://www.chipverify.com/verilog/verilog-tutorial>

Vivado HelloWorld: <https://www.youtube.com/watch?v=Mb-cStd4Tqs>

Vivado FPGA Programming and Hardware Manager:

<https://www.youtube.com/watch?v=f7xp3SC2iwM>

Version 2

Vivado PS and PL Interface: https://www.youtube.com/watch?v=_odNhKOZjEo,

<https://www.youtube.com/watch?v=AOy5l36DroY>

PySerial docs: <https://pyserial.readthedocs.io/en/latest/>

Version 3

Adding external modules: <https://www.youtube.com/watch?v=bsvpJYCDmCQ>

FPGA UART Interface: <https://www.youtube.com/watch?v=lzQ9hJ-wevg>,

<https://www.youtube.com/watch?v=J-nhxLiv2Uw>

FPGA PMOD Ports:

https://digilent.com/reference/_media/reference/programmable-logic/zybo/zybo_rm.pdf

USB2UART Reference:

https://digilent.com/reference/_media/reference/pmod/pmodusbuart/pmodusbuart_rm.pdf

UART Module used: <https://github.com/wd5gnr/icestickPWM>

Drivers for USB2UART: <https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers>

Version 4

ILA Demo: <https://www.youtube.com/watch?v=H4LQY6ZbBDU>

And of course, if all else fails, feel free to reach out to me at rahulvjayachandran@gmail.com anytime.

1 Version 1: The Proof Of Concept

1.1 Summary

This version features the Ring Oscillator module as well as a line from it's output to a flashing light on the ZYNQ board, which blinks on and off to demonstrate the functioning Ring Oscillator. All of this is built within the programmable logic of the FPGA.

1.2 The PL Diagram

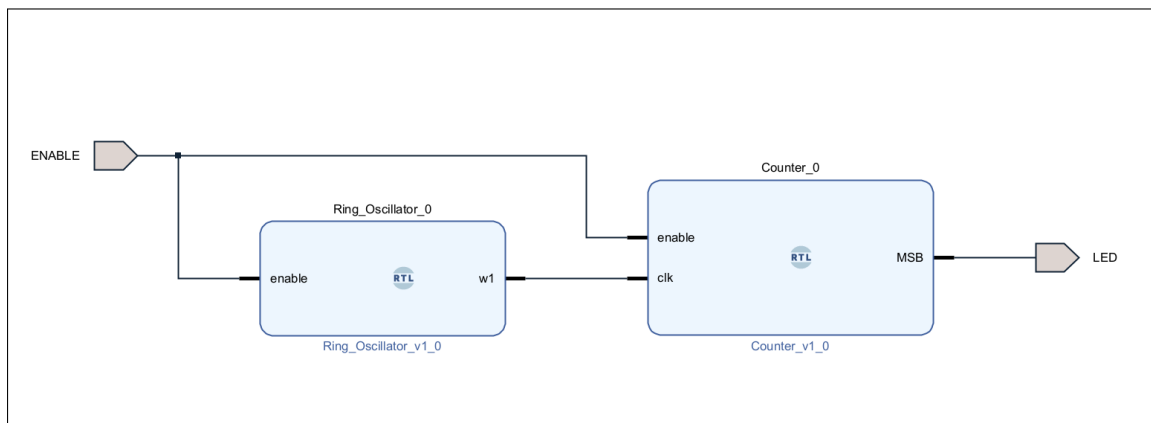


Fig 1.2.1: Version 1 PL Diagram

Here we can see the modules and connections in the programmable logic. The ring oscillator has one input which enables or disables its oscillation, and one output, which is connected to one of the wires within the module, and oscillates while the RO is live. This output wire then feeds into the clock splitter module, which simply oscillates its output every 2^{20} (around 1,000,000) oscillations of the input, thus lowering the frequency 1,000,000x. This is necessary because the RO oscillates too fast for us to see, so we must slow it down. This wire finally connects to the LED on the board.

1.3 The XDC code

We can see the input labelled ENABLE, and the output labelled LED in the PL Diagram. Within the XDC file, we see that the pins G15 and M14 have been renamed as ENABLE and LED to allow for this IO to function. We can see that here:

```

12 :
13 : ##Switches
14 : set_property -dict { PACKAGE_PIN G15   IOSTANDARD LVCMOS33 } [get_ports { ENABLE }]; #IO_L19N_T3_VREF_35 Sch=sw[0]
15 : #set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { sw[1] }]; #IO_L24P_T3_34 Sch=sw[1]
16 : #set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports { sw[2] }]; #IO_L4N_T0_34 Sch=sw[2]
17 : #set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { sw[3] }]; #IO_L9P_T1_DQS_34 Sch=sw[3]
18 :
19 :
20 : ##Buttons
21 : #set_property -dict { PACKAGE_PIN K18   IOSTANDARD LVCMOS33 } [get_ports { btn[0] }]; #IO_L12N_T1_MRCC_35 Sch=btn[0]
22 : #set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [get_ports { btn[1] }]; #IO_L24N_T3_34 Sch=btn[1]
23 : #set_property -dict { PACKAGE_PIN K19   IOSTANDARD LVCMOS33 } [get_ports { btn[2] }]; #IO_L10P_T1_AD11P_35 Sch=btn[2]
24 : #set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [get_ports { btn[3] }]; #IO_L7P_T1_34 Sch=btn[3]
25 :
26 :
27 : ##LEDs
28 : set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { LED }]; #IO_L23P_T3_35 Sch=led[0]
29 : #set_property -dict { PACKAGE_PIN M15   IOSTANDARD LVCMOS33 } [get_ports { led[1] }]; #IO_L23N_T3_35 Sch=led[1]
30 : #set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { led[2] }]; #IO_0_35 Sch=led[2]
31 : #set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports { led[3] }]; #IO_L3N_T0_DQS_AD1N_35 Sch=led[3]
32 :

```

Fig 1.3.1: The XDC Code for IO Pins

In the GitHub repo is the master XDC for the ZYNQ-7010. Always use this XDC file to create your IO on the board, just uncomment the pins you need.

1.4 Expected outputs

When the switch is flicked into the ON position, we expect to see the LED start flashing on and off. If we flick the switch off, we expect the LED to be frozen in either the on or off position. This will demonstrate that the RO is oscillating.

1.5 Conclusion

While this is a good proof of concept, it doesn't actually measure frequency, which is the end goal. In the next versions, we build and revise different sampling frameworks to measure and transmit frequency data directly to our external computer.

2 Version 2: The naive sampling method

2.1 Summary

This version features the Ring Oscillator module, a counter, and a sampling framework contained within the onboard SoC. The counter interfaces with the SoC via an AXI GPIO (See resources), and uses a script written in C to take values and send them to our external computer. There is also a second AXI, which is used to send a signal to an external LED, which is on while the program runs.

2.2 The PL Diagram

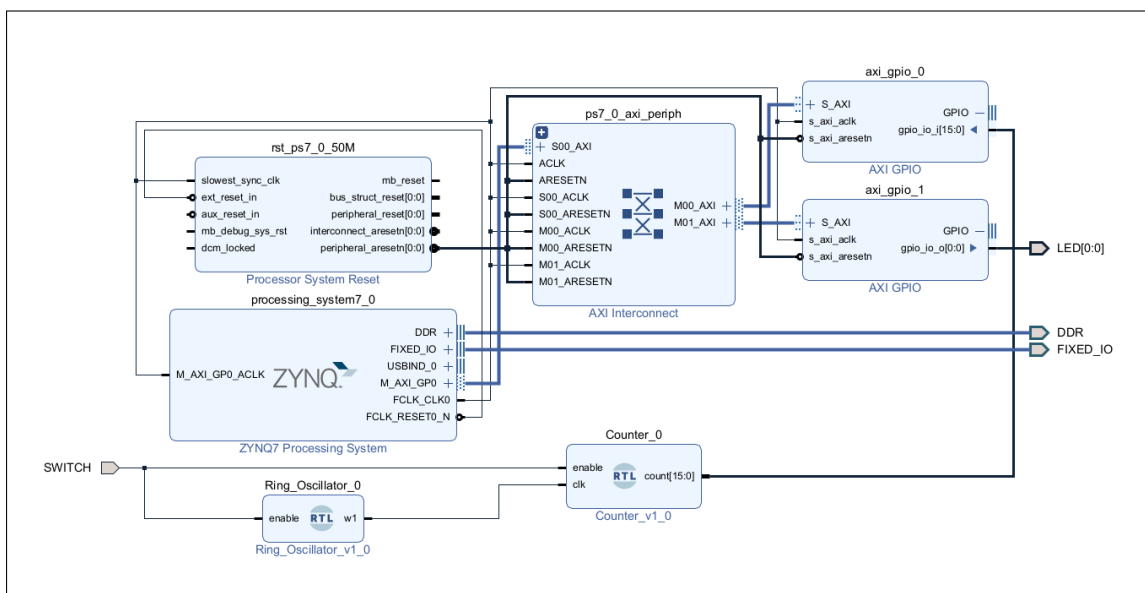


Fig 2.2.1: Version 2 PL Diagram

Here we can see the modules and connectors in the programmable logic. The ring oscillator remains the same, but now it is connected to a counting module, which increments its 32-bit output by 1 every time the RO completes an oscillation. This output is then tied to an AXI GPIO module, which allows the SoC to read the counter value at any time. The second AXI module controls the signal LED on the board (see XDC file)

2.3 the SoC Code

```

1  #include <stdio.h>
2  #include "platform.h"
3  #include "xil_printf.h"
4  #include "xgpio.h"
5  #include "xparameters.h"
6
7
8  int main()
9  {
10     init_platform();
11
12     print("Hello World\n\r");
13
14     XGpio input,output;
15     int a; //a is the input to the PS
16     int y = 1; //y is the output to the LED
17
18     //init
19     XGpio_Initialize(&input,XPAR_AXI_GPIO_0_DEVICE_ID); //make sure the device ID matches the instance (input/output)
20     XGpio_Initialize(&output,XPAR_AXI_GPIO_1_DEVICE_ID);
21
22     print("halfway there\n\r");
23
24     //code
25     //Data Direction Reg, input is 1, output is 0
26     XGpio_SetDataDirection(&input,1,0xFFFF);
27     XGpio_SetDataDirection(&output,1,0);
28
29     print("init complete\n\r");
30
31     int b = 0;
32     while(1){
33         a = XGpio_DiscreteRead(&input,1);
34         xil_printf("%d", a);
35         print("\n\r");
36     }
37
38     print("Successfully ran Hello World application");
39     cleanup_platform();
40     return 0;
41 }
42

```

Fig 2.3.1: Version 2 SoC Code

Here we can see the C code that the SoC runs. This code is very similar to that in the AXI GPIO video (See resources), except it simply runs forever, continually sampling the counter value, and sending it to our external computer as fast as possible. To run this yourself, follow the steps in the AXI GPIO video.

2.4 Outputs

When run using the in built Vitis serial terminal, we see some confusing outputs:

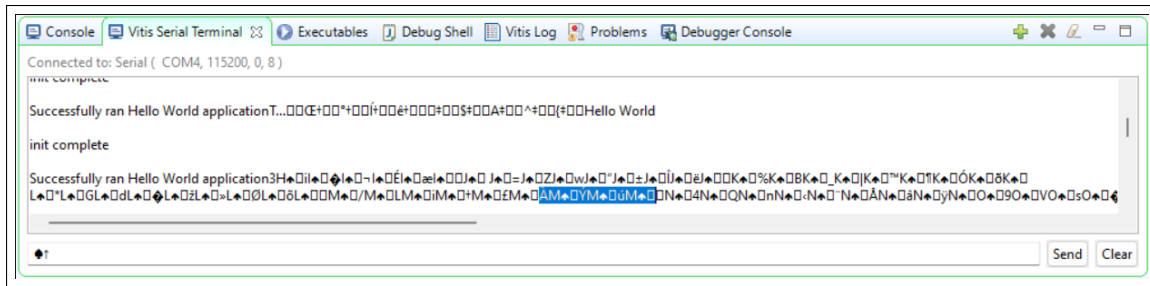
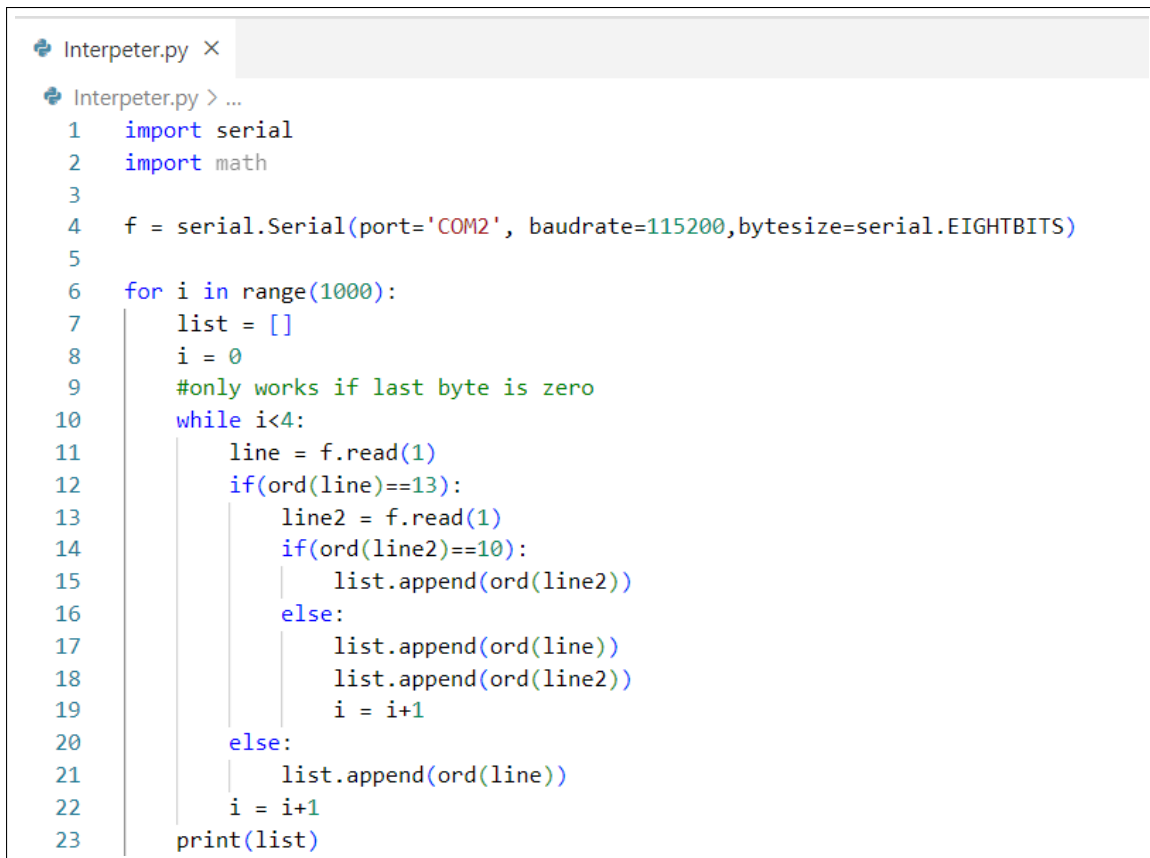


Fig 2.4.1: Version 2 Vitis terminal outputs

This is because the values returned over the serial line by the SoC are in the form of 8-bit chunks, and the serial terminal interprets them as ASCII characters. Thus, it displays the ones it is able to, and leaves boxes when unable. To circumvent this, we can write our own python script to take in this serial data and send it to our CMD window in a different format.

Here is an image of the Python code, which utilizes PySerial, a package that allows interfacing with serial lines through the ports on your PC:



```
Interpeter.py ×
Interpeter.py > ...
1  import serial
2  import math
3
4  f = serial.Serial(port='COM2', baudrate=115200, bytesize=serial.EIGHTBITS)
5
6  for i in range(1000):
7      list = []
8      i = 0
9      #only works if last byte is zero
10     while i<4:
11         line = f.read(1)
12         if(ord(line)==13):
13             line2 = f.read(1)
14             if(ord(line2)==10):
15                 list.append(ord(line2))
16             else:
17                 list.append(ord(line))
18                 list.append(ord(line2))
19                 i = i+1
20         else:
21             list.append(ord(line))
22             i = i+1
23     print(list)
```

Fig 2.4.2: Version 2 PySerial Code

This code expects 4 bytes of information per sample, and displays them on a line by line basis in the CMD Prompt when run. the conditionals are used to handle an exception with the Windows OS in which it will always append a bit with value of 10 with one of value 13. This extra code removes that appended bit. The output looks like this:


```
Anaconda Prompt (anaconda3)
[24, 160, 0, 0]
[179, 162, 0, 0]
[79, 165, 0, 0]
[236, 167, 0, 0]
[136, 170, 0, 0]
[35, 173, 0, 0]
[186, 175, 0, 0]
[92, 178, 0, 0]
[249, 180, 0, 0]
[147, 183, 0, 0]
[32, 186, 0, 0]
[205, 188, 0, 0]
[105, 191, 0, 0]
```

Fig 2.4.2: Version 2 PySerial Code Output

We read this from least significant bit to most significant bit, and thus we can see that the RO oscillates about 650 times per sample

We even went 1 step further, using matplotlib to graph the values, and doing some error correction. The output graphs look like this:

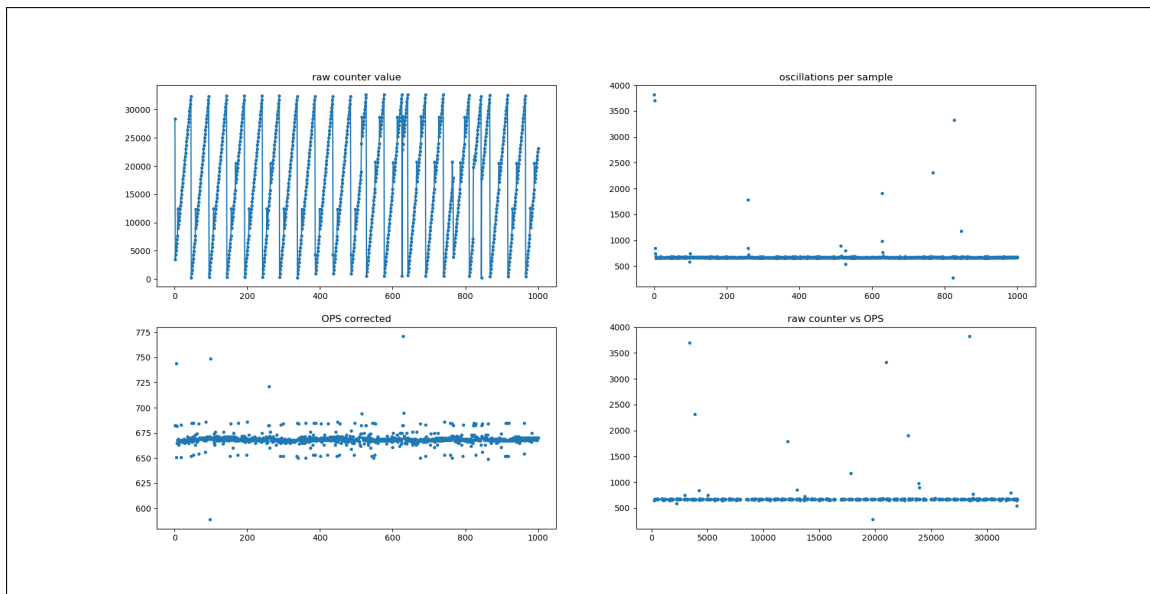


Fig 2.4.2: Version 2 PySerial Code Output (Graphed)

To replicate these graphs, simply run `Interpreter2.py` while the FPGA code is also running, then run `Processor.py` and `Grapher.py` in that order.

This looks promising, as it shows a clear steady increase in the counter value, as well as a very close grouping of the Operations per Sample (Here OPS corrected just removes outliers, read python code to see method).

2.5 Problems with this version

This version works well for graphing the RO values alone, but when we added artificial noise to the board (see research paper), the outputs became scrambled and uninterpretable, as pictured in the below `Grapher.py` output:

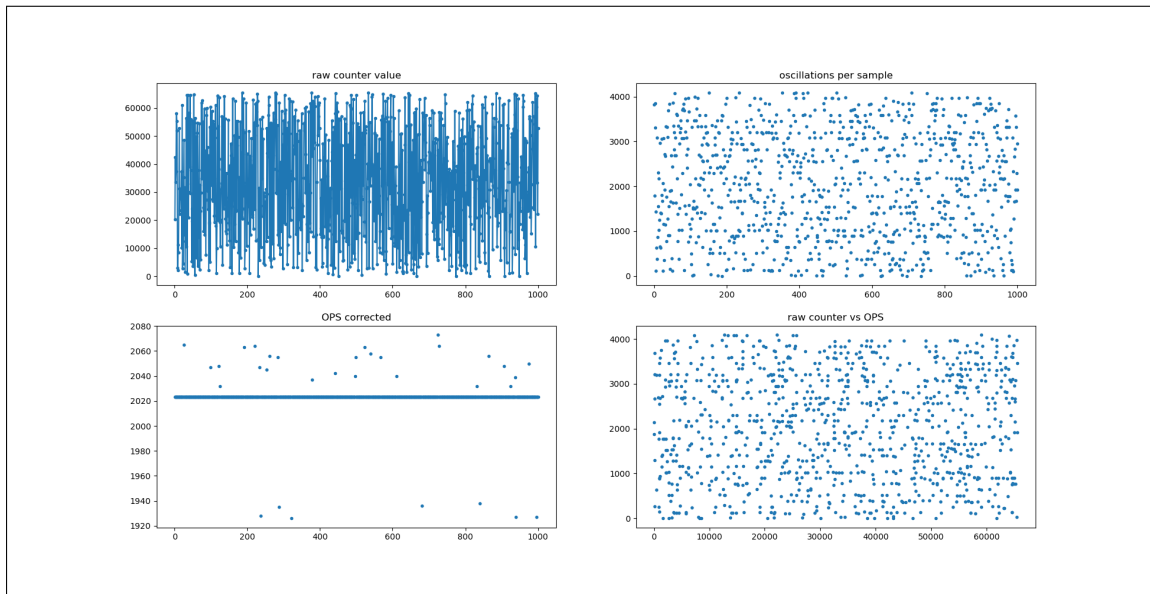


Fig 2.5.1: Version 2 Graph with single DFF in Programmable Logic

This graph was created by the RO when a single D-Flip-Flop was added to the PL. This DFF was idle, and completely connected to a switch and LED. It should have had no effect on the output, but it absolutely destroyed it.

To reconcile this bizarre outcome, I remade the counter module, rewrote the python code, and even used a system clock to make sure the sampling framework was going at a constant rate. Nothing I tried worked, and since the PL code was already validated by attempt 1, I concluded that something was wrong with the AXI framework and the C code. Furthermore,

I realized that there was a way to offload the data without using the SoC at all, which would help reduce the noise on the board, so I moved on to version 3.

3 Version 3: the UART system

3.1 Summary

This version features a new RO design which is more robust against Vivado's optimization processes. It also uses a UART module which takes in an 8 bit wide register, and outputs a serial signal in the UART format. This is then sent to the computer through an external pin and a UART to USB module.

3.2 The PL Diagram

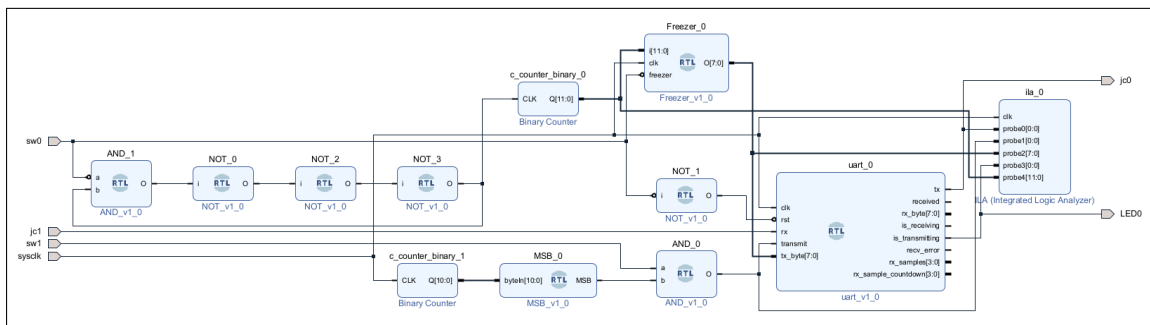


Fig 3.2.1: Version 3 PL Diagram

Here we can see the modules and connectors in the programmable logic. The ring oscillator now connects to a 12-bit counter, which then sends the 8 most significant bits to the UART module. This module in turn connects to an external PMOD pin on the FPGA board, which is used to send the serial data. When SW1 is on, the UART transmits data at 1,024,000 baud over the line, which is then intercepted by the python script on the computer.

This diagram is a bit confusing, so let us go section by section:

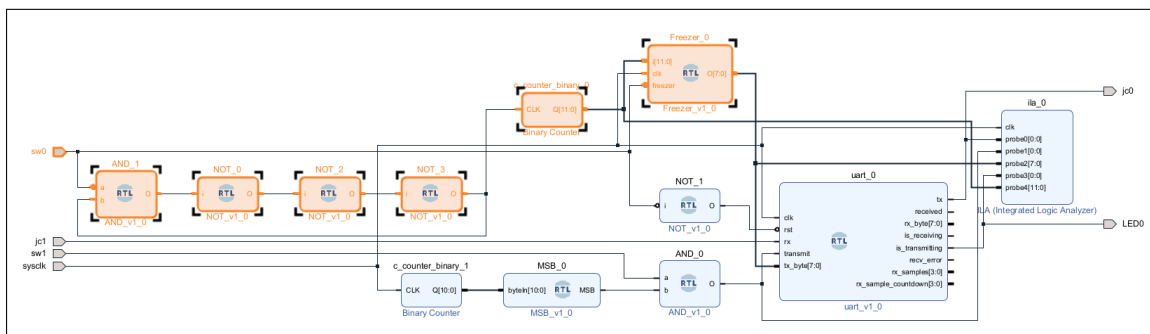


Fig 3.2.2: Version 3 PL Diagram, RO System

Here, we see the new RO design, which utilizes individual modules for the gates. We

also see the 12 bit counter, and the module called 'freezer'. The freezer module does 2 things. First, it takes the 8 most significant bits from the counter. This is because the RO oscillates too fast for the UART to just send an 8 bit count, so we must sample at a lower resolution, sending an 8 bit value which represents the number of time the RO completed 16 oscillations. The second function of the freezer is to stop the RO from oscillating while the UART is sending the value over the line. Otherwise, the value would change as the UART transmitted, and the transmitted value would be wrong.

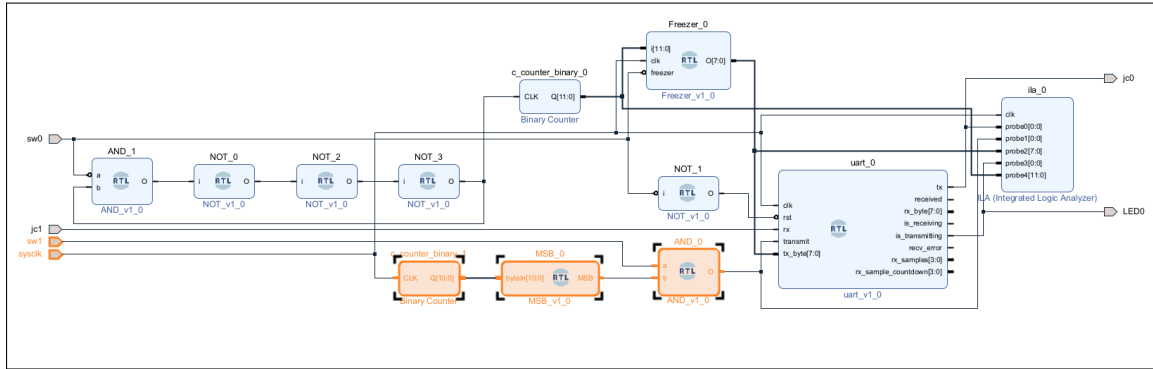


Fig 3.2.3: Version 3 PL Diagram, UART Timing System

Here, we see the timing system. The 125 mHz system clock (sysclk, see XDC) is fed into an 11 bit counter, and then the last bit is used to send the transmit signal to the UART. Thus, the UART transmits values at a rate of 61 kHz per transmit.

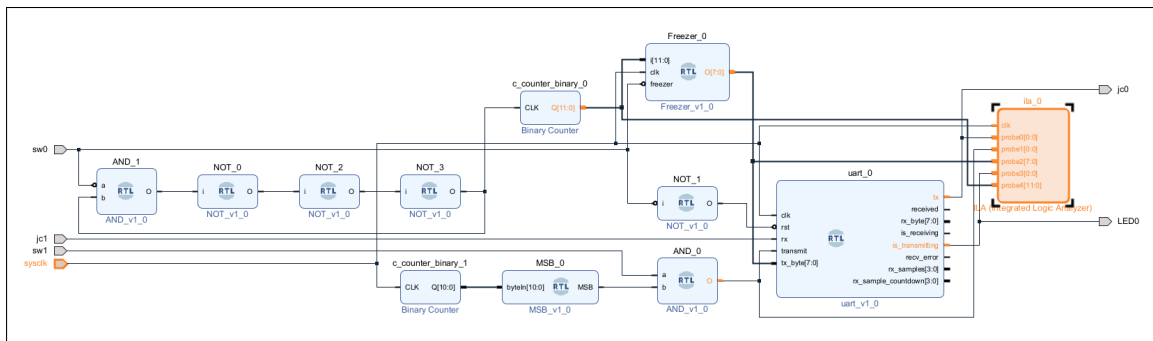


Fig 3.2.4: Version 3 PL Diagram, ILA System

Finally, the ILA framework. We use the ILA here to monitor the UART signal and ensure it works. connect the trigger to the transmit signal, so it shows us the serial transmit value.

3.3 The new Ring Oscillator

As we saw, the RO is now made of 4 modules, instead of combining them into 1. I did this because I do not want Vivado to try and optimize my code, thus changing the wiring design.

Vivado takes your verilog and creates LUTs with truth tables using boolean expressions for the outputs. Thus, it does not directly preserve your design unless forced. That is why I keep each gate as its own module, and add the DONT TOUCH line to them. In this way, Vivado is forced to make an individual LUT for each of them. Below are images of Vivado's implemented schematics for both the old and new ROs (pictures taken from the schematic of the implemented design):

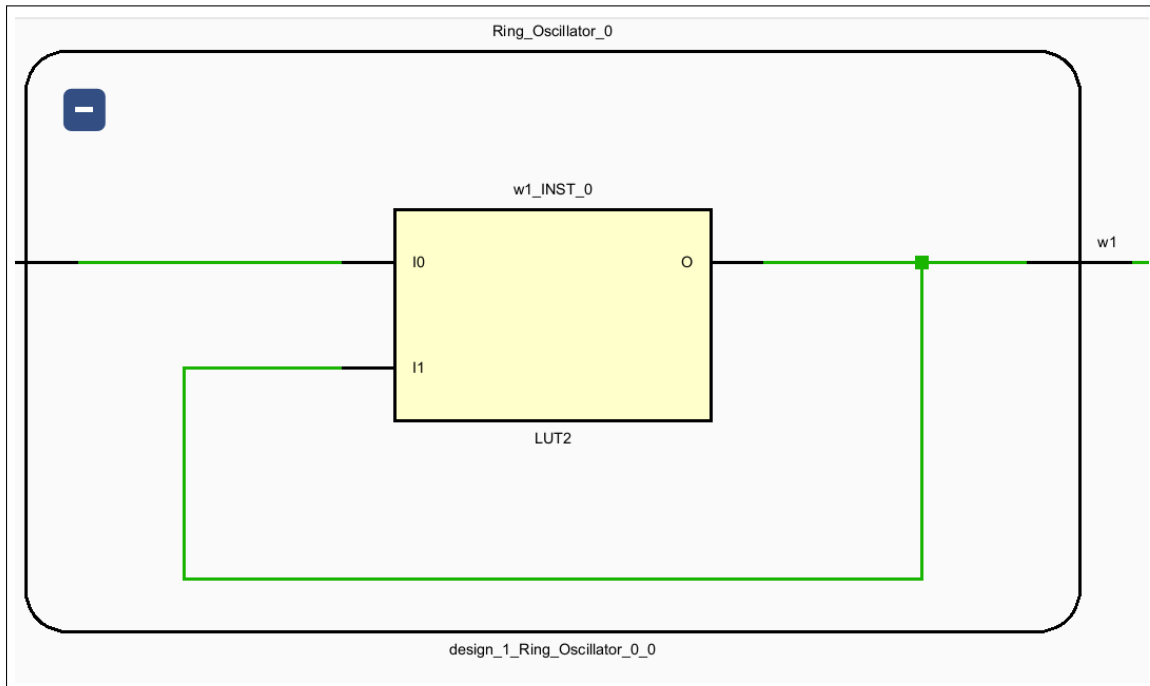


Fig 3.3.1: Old RO Implemented Schematic

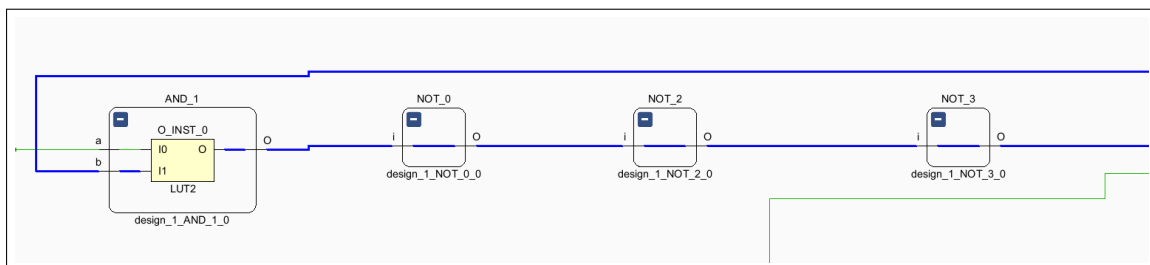


Fig 3.3.2: New RO Implemented Schematic

As we can see, the new code assures the correct implementation of the RO.

3.4 the UART module

this module was taken from wd5gnr at <https://github.com/wd5gnr/icestickPWM>. It is very well documented, and everything is explained in the README, and the comments in the actual code (I used version 2). One important note for this specific use: you MUST match the baud rate in the UART module code to that on the python script, they are currently both set to 1,024,000 in the GitHub repo.

3.5 The UART to USB board

This board creates a virtual serial line on a USB cable and sends the UART values over it. You have to install a driver onto your computer for it to function, however, (See resources) and once you do, it should appear in device manager as a serial line, much like the one used in version 2.

3.6 The Setup

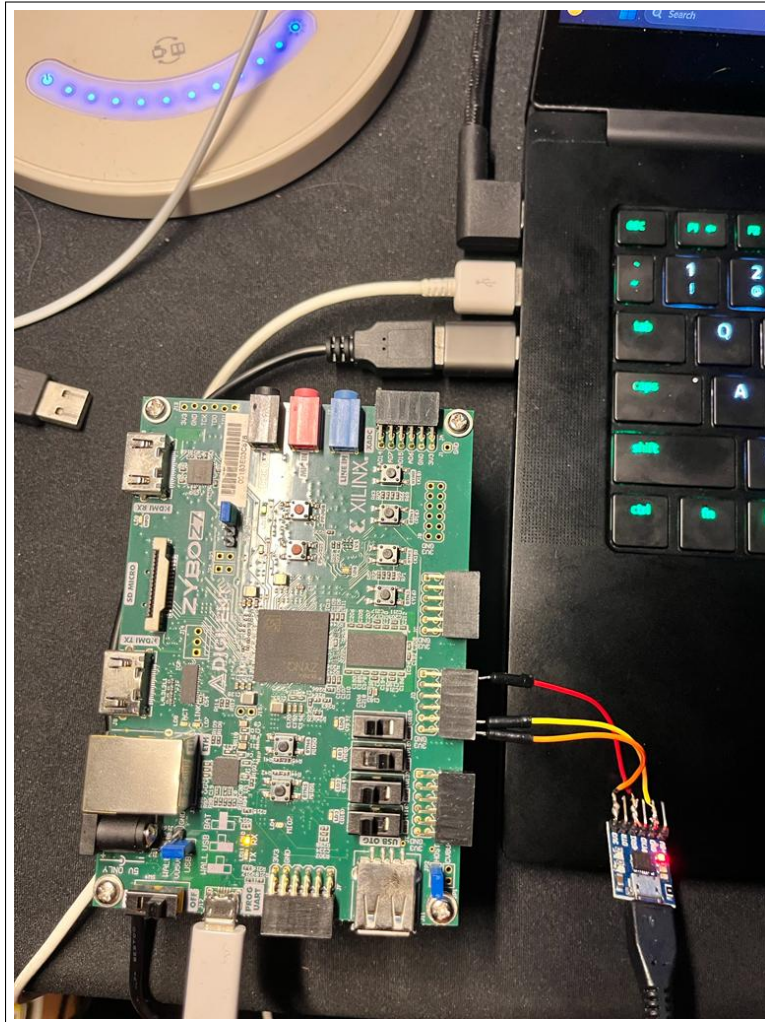


Fig 3.6.1: The RO With UART Setup

In this image, we see the programming cable from the computer to the mini USB port on the Zybo, then the 3 pin wires (3v3, signal, and ground) from the USB2UART module to the FPGA's PMOD ports, then the USB cable from the USB2UART module to another USB port on the computer. To run this version, first start the python code, then start the UART module.

3.7 Conclusion

While this setup does work, it is limited by the speed at which the UART can transmit data, which limits resolution. The RO oscillates more than 256 times in the duration of the UART transmit, which means that we have to take the 8 most significant bits of a 12 bit counter, which means that we are counting every 16 oscillations, instead of every single one.

Furthermore, the UART module requires physical access to the PMOD pins, which does not allow for cloud based attacks, which was an integral part to the paper.

4 Version 4: The Final Setup

4.1 Summary

This version utilizes Xilinx's ILA cores to read the values on the RO counter, and transmit them back over the same line used to program the device. The ILA removes the need for PMOD ports or interfacing with the SoC, making this design much simpler.

4.2 The PL Diagram

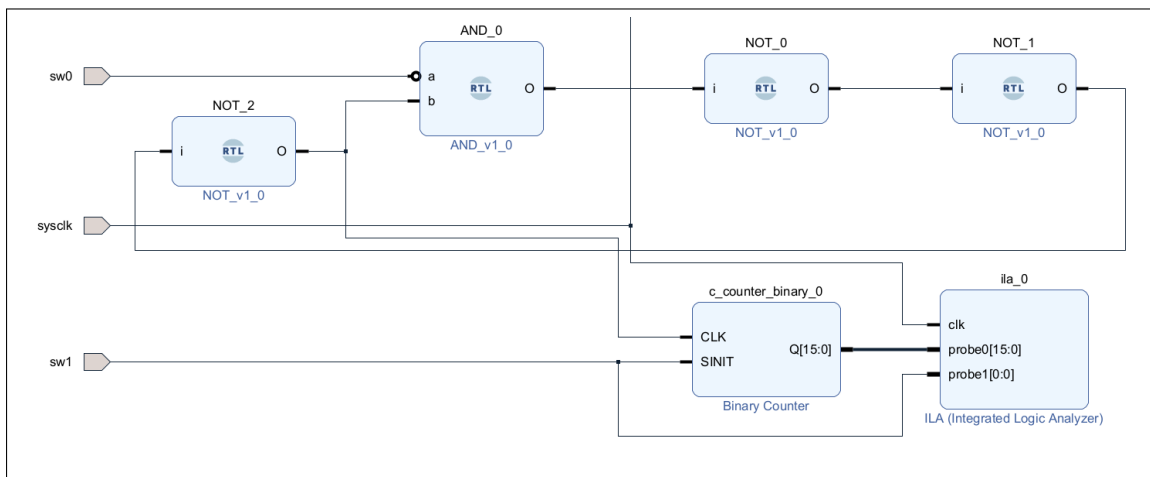


Fig 4.2.1: Version 4 PL Diagram

This design features the new RO (seen as the AND and NOT gates, a binary counter which is 15 bits wide, and an ILA, which samples the counter, and the counter's enable switch on every oscillation of the 125mHz sysclk. Here, the enable switch also acts as a trigger for the ILA, allowing us to control when it samples data (see resources). Since all results are contained in the paper, I will not discuss them here. I will, however, explain the power virus framework.

4.3 The Power Virus Framework

An individual virus is simply a DFF which takes the negative of it's output as it's input. Thus, it switches value on every oscillation of its clock (see Virus1.v for code). I then take this virus (called Vi), and generate 1000, 2000, or 4000 of them, depending on the module (Virus1, Virus2, or Virus3). Then, I use a combination of 2 switches to enable any combination of these 3 modules. This can be seen in the below image:

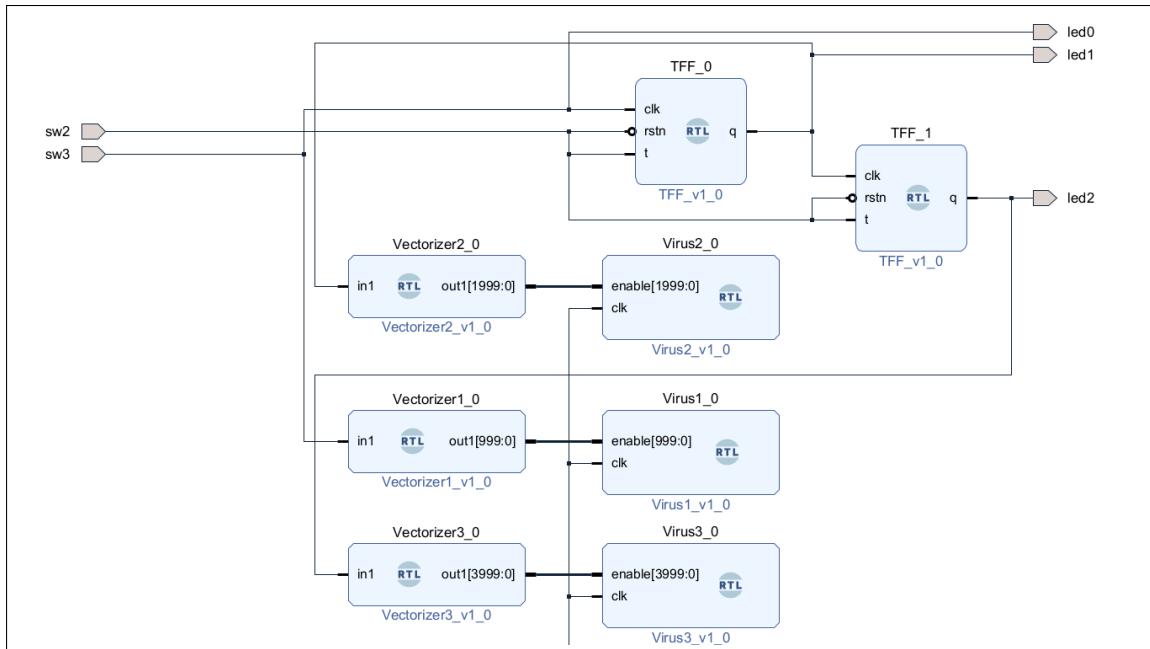


Fig 4.3.1: Version 4 Power Virus Framework

(The vectorizer modules simply split a single wire input into many wires with the same value)

This design allows for any combination of the Virus modules to be activated, thus allowing for increments of 1000 for my tests.

4.4 Conclusion

The ILA module allows for very high resolution sampling of the data. For a continuous sampling framework, the UART module may be better, since ILA requires triggers, and I was not able to connect it to a python script for automatic data recording. The next step would be to automate the trigger (using the auto trigger function), and find a way to feed the serial data to a csv file.