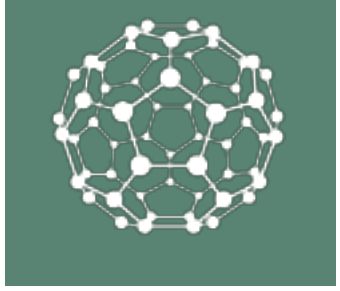


JANUARY 13, 2017 FPGA WRITTEN BY: ANTON POTOČNIK 42 COMMENTS PERMALINK

Red Pitaya FPGA Project 4 – Frequency Counter

Introduction

On the way to a powerful acquisition systems let us make a quick detour and create a useful and simple project – *the Frequency Counter*. Yes, to measure frequencies one can use Red Pitaya's native apps such as Oscilloscope or Spectrum Analyzer, however, our program will be able to determine frequencies with much higher resolution and at the same time we will learn how to use Red Pitaya's 125 Msamples/s 14-bit ADC and DAC peripherals in the FPGA program.

This project contains two separate parts: the *data acquisition* part with frequency counter and LED data display and the *signal generator* part. To communicate with these two parts we use the General Purpose IO block for setting configuration values and reading the counter output.

The frequency counter will be implemented in the reciprocal counting scheme where a period of time of a predefined number of signal oscillations is measured and then inverted and divided by the number of oscillations. Such scheme can yield a much better frequency resolution, especially for low frequency signals, compared to the conventional method where number of signal cycles are counted in a predefined gate time.

Building the Project

To start off download the project from the github. First, build the necessary custom IP cores by navigating to */redpitaya_guide* base folder in Vivado's tcl console and execute

```
source make_cores.tcl
```

This will create all custom IP cores in the */redpitaya_guide/tmp/cores* folder. In this project

we will use *axis_red_pitaya_adc* and *axis_red_pitaya_dac* IP cores from Pavel Demin which are handling fast ADCs and DACs peripherals on the Red Pitaya board. I only added a global clock buffer to the *axis_red_pitaya_adc* core for an optimal performance under Vivado 2016.3 and higher version.

Once the custom cores are created build the project by appropriately modifying the *make_project.tcl* script and execute it in Vivado's tcl console with

```
source make_project.tcl
```

Project overview

The full block design of the frequency counter project is composed of six parts: Processing System, GPIO, Signal Generator, Data Acquisition, Frequency Counter and Signal Decoder block as shown in the figure below.

Block Design Overview

These parts will be described in detail below. You can skip the lengthy description and go directly to the fun part at the end of the post.

Processing system

Let's start with the most common part – the processing system IP core. Together with the AXI Interconnect and Processor System Reset block these are the most common blocks in most of the Zynq 7000 FPGA applications. Since they take quite some space and have a lot of connections we will join them in a single hierarchy block, so they will take less space and make block design more transparent. To create a hierarchy select desired blocks, right click and select *Create Hierarchy*. From now on we will put in hierarchies most of the blocks with related functionality.

Processing System 7 Hierarchy

General Purpose Input-Output Core

In the previous project we have learned how to write and read from the FPGA logic. We will use the same approach here for setting configurations such as number of cycles and signal generator's phase increment. We will use the first GPIO port as an input to make results of the frequency counter available to a program running on the Linux side. Second GPIO port will be used as an 32-bit output port containing 27-bit *phase_inc* value for the signal generator and 5-bit *log2Ncycles* value for the frequency counter:

gpio2_io_o[31:0] = $_{31}$[{27-bit *phase_inc*} {5-bit *log2Ncycles*}]$_0$.

If you ever need more configuration output bits you can use Pavel Demin's *axi_configuration* IP core with a custom number of bits in a single output port. *axi_configuration* can be found in the *redpitaya_guide/core* folder, which is automatically created with the *make_cores.tcl* script as described above.

Signal Generator

Signal Generator hierarchy creates a sin(ωt) and cos(ωt) signals at the two DAC output ports with a user defined frequency. The analog signal is generated with three blocks: *DDS compiler* for calculating 14-bit sinusoidal values, *Clock Wizard* to create a double clock frequency which allows setting the two DAC channels on each input clock cycle and *AXI-4 Stream Red Pitaya DAC* core for setting signal values to the external DAC unit. We will use 125 MHz *adc_clock* as input clock to achieve 125 Msamples/s data rate.

Signal Generator Hierarchy

Frequency, amplitude and other parameters can be set in the Direct Digital Synthesizer (DDS) re-customization dialog. Current DDS core settings will create sin(ωt) on one and cos(ωt) on the other DAC channel with maximal amplitude of +/- 1V (maximal range) on both channels.

The synthesized signal frequency is in the DDS compiler determined by a phase increment value at each clock cycle. A nice description of the signal synthesizer operation can be found in the DDS compiler product guide. The signal frequency can be set fixed at the

design stage by choosing *Fixed* Phase Increment in the DDS re-customization dialog. In this case the dialog automatically calculates the required constant phase increment for a desired frequency and frequency resolution. Note that the output frequency will be a divisor of the clock frequency and might therefore deviate from the requested frequency.

Since we want to change the frequency during an operation we choose *Streaming* Phase Increment in the re-customization dialog, which requires a phase increment value to be continuously supplied to the S_AXIS_PHASE input interface. AXIS interface implements the AXI4-Stream protocol developed for fast directed data flow. It implements the basic handshake using at least *tvalid* and *tready* signals, however, we will neglect even those for our nearly constant phase increment value. To create a continuous stream of the user defined values we use Pavel Demin's AXI4-Stream Constant IP core, which converts 32-bit input bus to the AXIS master interface. For the input we take 27-bit *phase_inc* value from the *gpio2_io_o* port using Slice IP core. Calculation of the *phase_inc* for a desired output frequency will be discussed in the last part of the post.

Data Acquisition

AXI4-Stream Red Pitaya ADC Core

The first block in the Data Acquisition hierarchy is the axis_red_pitaya_adc_v1_0 IP core with two main features. First, it converts the external 125 MHz clock from *adc_clk_a* and *adc_clk_b* differential external ports into our programmable logic as a *adc_clk* clock. Second, it reads the ADC data from two input channels which becomes available on each *adc_clk* clock cycle and makes it available over the AXI Stream (AXIS) interface *M_AXIS*. *axis_red_pitaya_adc_v1_0* IP core uses two ports of the AXIS interface, the *axis_tvalid* port which is always asserted and the *axis_tdata* a 32-bit data port with new measurements available on every clock cycle. 32-bit *axis_tdata* contains 16-bit channel 2 value and 16-bit channel 1 value:

M_AXIS_tdata[31:0] = $_{31}$[{16-bit ADC2 value} {16-bit ADC1 value}]$_{0}$.

Since Red Pitaya has 14-bit ADC the 16-bit value has two most significant bits set to either 00 or 11 depending on the sign of the measured value. It is instructive to have a look at

the Verilog code of AXI4-Stream Red Pitaya ADC core. Note that Red Pitaya's ADC core has an additional output port (*adc_csn)* connected to the external port *adc_csn_o* for a clock duty cycle stabilization.

Data Acquisition Hierarchy

Signal Split Module

The second block in the hierarchy is the *signal_split* RTL module. It transforms ADC output interface M_AXIS with two channel values into two M_AXIS output interfaces each containing a single channel value. The module has a very simple Verilog code, which can be found on github.

It is interesting to note that if you want to create an input or an output interface on a RTL module, simply name the input or output ports with a standard interface notation (see Vivado IP user guide). For example, in the signal_split RTL block port names: *S_AXIS_PORT1_tdata* and *S_AXIS_PORT1_tvalid* are automatically combined into an *S_AXIS_PORT1* interface.

Frequency Counter Module

The frequency counter hierarchy is build around its main RTL module *frequency_counter* with two main inputs: *S_AXIS_IN* interface containing measured single channel ADC signal and *Ncycles,* a value that specifies a number of signal oscillation for time measurement. Since exact number for *Ncycles* is not important user specifies a 5-bit logarithmic value *log2Ncycles* via the gpio core. *Ncycles* is then calculated as

Ncycles = 2^*log2Ncycles*

using a *pow2* RTL module. See the figure below.

Frequency Counter Hierarchy

The verilog code of the *frequency_counter* RTL module has three main parts. The first part directly wires the *S_AXIS_IN* to the *M_AXIS_OUT* interface so that data is transferred to the next block for processing. Instead, we could split the AXIS interface before the module, however, this would require an additional IP core – the AXI3-Stream Broadcaster.

The second part of the code sets the *state* buffer depending on the measured signal value relative to the high or low threshold values. If the signal is above the high threshold value *state* buffer is set to one and if the signal is below the low threshold value *state* buffer is set to 0. Using two threshold values helps to prevent false *state* transitions in case of noisy data.

The third part increments *counts* register on each clock cycle, increments *cycles* register on each positive *state* transition and clears *cycles* and *counter* registers when *cycles* exceeds *Ncycles*. Before clearing the *counter* its value is copied to the *counter_output* register which is wired to the output port. The result of the frequency counter module is therefore a number of clock cycles in a time of *Ncycles* signal oscillations, updated on each *Ncycles* signal oscillations. The frequency is then calculated as

$frequency = Ncycles/counts * 125\ \text{MHz}.$

Signal Decode Module

The final block in the ADC signal chain and in the block design is the *signal_decode* RTL module. Its purpose is to display the ADC value on the Red Pitaya LED bar mostly for visual effects. The implementation is a simple 8-bit decoder from Vivado's Language Templates. In signal_decoder.v the three MSBs of the ADC value are decoded and displayed on LEDs. However, if your ADC range jumpers are set to +/- 20 V instead of +/-1 V you will see no activity when connecting the output of the Red Pitaya's DAC to the input of its ADC port. In this case *BIT_OFFSET* parameter can be set to 4 to decode 4th, 5th and 6th signal's MSBs. Shifting the bit position is related to signal amplification by a factor of 2. You can play with this value if the range is not optimal.

Fun Part

We are ready to test the frequency counter. Connect the Red Pitaya's OUT1 port to the IN1 port. Save the project, create bitstream and write it to the FPGA as described in previous projects.

Next, copy the *counter.c* program found in *redpitaya_guide/4_frequency_counter/server* folder to Red Pitaya's Linux, compile it and execute it as shown in the figure below.

Demonstration of counter.c program

The program can be used with the following parameters:

```
./counter {Log2Ncycles} {Frequency_Hz}
```

Keep in mind that the frequency resolution depends on the number of clock counts within *Ncycles* signal oscillations. Low frequency signals require small *Ncycles* and high frequencies signals require large *Ncycles*. The maximal number of *counts* can be 2^32, the highest DAC frequency can be 125 MHz/4 = 31.25 MHz and the lowest frequency can be approx. 1 Hz. The conversion from the desired frequency into the *phase_inc* is done in the *counter.c*.

When setting the frequency to 2 Hz the LED bar on the Red Pitaya board looks very much like Knight Rider's lights 🙂

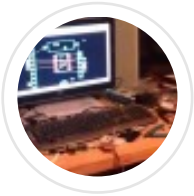
Scroll horizontally to view the whole table

<< Red Pitaya Project 3 – Stopwatch Red Pitaya Project 5 – Averager >>

References

- Reference on RTL module interface Xilinx User Guide – Designing IP Subsystems Using IP Integrator.
- Xilinx Product Guide – DDS Compiler
- Vivado's AXI4 Reference Guide

42 Comments



Mick Phillips

February 11, 2017 at 6:02 pm

Reply

This works great with the ADC gain jumpers set to HV, but not when they're set to LV. From the code comments, I saw that I needed to change the BIT_OFFSET on the signal decoder – this fixes the LEDs. However, the counts and reported frequencies are wrong. I think I also need to change the thresholds in the frequency counter. I can not figure out values that work (I tried scaling the existing values according to the full-scale ranges I see on the ADC at the two different gains, but it didn't work). Can you tell me more about how those threshold values are chosen?

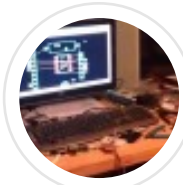
Thanks for this great blog series, by the way! I have learned a lot.

Anton Potočnik

February 14, 2017 at 12:53 am

Reply

Hi, the mean value of the thresholds I got from another application where I actually measure and display raw ADC data (project 5 at github) and then the difference between the high and the low value I chose to be larger than measurement uncertainty, but still of comparable order. I would probably do the same for the LV setting. Good starting values for the thresholds would be values close to zero with similar difference as in HV case.



Mick Phillips

February 22, 2017 at 2:36 pm

Reply

Ah – the problem could be that I scaled the range, rather than using a similar difference! Thanks. I'll take a look at project 5, too.

John Quin

February 22, 2017 at 1:19 am

Reply

Project 5 is very cool. Thanks for doing all this.

Pingback: SDR-WSPR with GPSDO 10MHz frequency stabilization – WSPRlive

---

Kerrie

May 12, 2017 at 5:37 pm

Reply

whoah this blog is great i love reading your posts. Keep up the good work! You understand, many people are searching around for this info, you could aid them greatly.

Don Pablo

August 25, 2017 at 8:03 pm

Reply

Hi!, I've been having problems, no matter what log2Ncycles or what frequency I input all I got is random numbers, I didn't touch anything in the project, can you please tell me what may be wrong? This is some of the data I get:

root@rp-f0514b:/tmp# ./counter 0 10

Counts: 41, cycles: 1, frequency: 3048780.48780 Hz

root@rp-f0514b:/tmp# ./counter 0 10

Counts: 14, cycles: 1, frequency: 8928571.42857 Hz

root@rp-f0514b:/tmp# ./counter 0 10

Counts: 43351, cycles: 1, frequency: 2883.43983 Hz

root@rp-f0514b:/tmp# ./counter 0 10

Counts: 21, cycles: 1, frequency: 5952380.95238 Hz

root@rp-f0514b:/tmp# ./counter 0 10

Counts: 37, cycles: 1, frequency: 3378378.37838 Hz

Youssef

September 15, 2017 at 8:13 pm

Reply

I'm having the same issue. Did you ever figure it out?

Anton Potočnik

September 15, 2017 at 11:47 pm

Reply

Hi Youssef and Don Pablo,

sorry for my inactivity lately. Mick's suggestion is very good and it might indeed work. See our earlier comments.

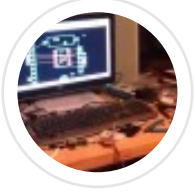
Let me know if this solves your problem so I can put a note in the text.

Youssef

September 19, 2017 at 3:19 pm

Reply

Changing the gain jumper setting to HV seemed to fix the problem. Thank you so much.

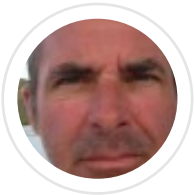


Mick Phillips

September 15, 2017 at 11:34 pm

Reply

I remember having the same issue. I think I resolved it by changing the gain jumpers on the input.



Lathuile Jean Pierre

September 24, 2017 at 11:04 am

Reply

Hi,

For those who use vivado 2017.2 change clock wizard version fron 5.3 to 5.4.

That’s mean line 80 of basic_red_pitaya_bd.tcl come :

create_bd_cell -type ip -vlnv xilinx.com:ip:clk_wiz:5.4 clk_wiz_0

Great job Anton.

parveen nisha

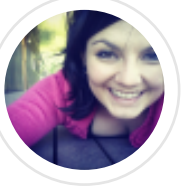
November 17, 2017 at 5:53 pm

Reply

Hi,

I am trying to implement this project with some modifications. I took an external clock for counting the frequency through the GPIO pin. I don’t know the C coding. Is the counter.c code is generated by launching the SDK in vivado? Can you suggest me what are the changes i need to do in C code for getting the output.

Thanks



mkfail

November 27, 2017 at 4:05 pm

Reply

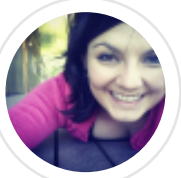
Hi Anton!

Your tutorials are the best, thank you so much for them!

I wonder if you where going to write a tutorial for the averager you have out on GIT?

I cant figure out what to do about the client / server so I am a bit stuck.

It would be incredibly helpful to get some pointers if you were not thinking about making a proper tutorial for that project.



mkfail

November 28, 2017 at 4:22 pm

Reply

Nevermind, I figured out that the thing with the client and server. I guess I will have to learn python now to understand what is going on but this does seem to be precisely the thing I need to do.

I need to measure impedance using a four terminal method so I will need to

measure on both input channels. It is my impression that you are doing all the calculations in the python program so I guess it is possible to take reproduce the averager hierarchy and attach it to the M_AXIS_PORT2, supposing there is enough memory available, this I dont know.

Then I need to figure out the code?

Anton Potočnik

December 14, 2017 at 3:58 am

Reply

Hi mkfail,

The next post is now published.

I hope you will enjoy it.

Abinaya

January 2, 2018 at 8:59 am

Reply

Hi Anton,

We are using your frequency counter project. In this Counter C code,

we couldn’t understand the following line.

phase_inc = (uint32_t)(2.147482*freq_in);

Kindly explain how this works and the number 2.147482.

Thanks in advance

Anton Potočnik

January 2, 2018 at 11:14 am

Reply

Dear Abinaya,

Thank you for your question. Direct Digital Synthesizer creates a time dependent sinusoidal signal from ever increasing phase value (phi = omega*t). Inside DDS block, there are lookup tables which convert input phase value phi to output sin(phi) or cos(phi) values. This approach avoids the use of resource demanding multiplication and division operations. To obtain time dependent sinusoidal signal with a specific frequency the phase value phi has to be increased at each clock cycle. Since the clock cycle duration is fixed (t0 = 1/125 MHz) the phase increase value (phase_inc) determines the frequency of oscillations. The frequency is proportional to the phase_inc and the proportional factor is 2.147482. I got this factor by playing with the DDS IP block re-customize dialog where you can read the phase increase factor for a desired frequency in the fixed phase increase programmability mode. If you want to calculate this value yourself, have a look at page 10 of the DDS Compiler manual (pg141-dds-compiler.pdf in References section).

Good luck!

Abinaya

January 2, 2018 at 3:32 pm

Reply

Thanks for the detailed reply. Few more doubt here,

Regarding the threshold value selection (-100, -150), We would like to know how you are choosing this numbers and whether they are variable.

Also,

In our application, we have a non periodic external analog signal instead of DDS compiler IP. In this case we need to consider only the positive half cycles. So , I am little unclear about the setting threshold values. Could you please help in this?

Regards

Abinaya

Anton Potočnik

January 7, 2018 at 10:20 am

Reply

Dear Abinaya

In the current project the threshold values are hard-coded in the .bit file. One can of course modify the FPGA program in Vivado and make it programmable. I would do it in the same way as the Ncycles parameter. However, for that you would need another GPIO IP or use Pavel's configurations ip core to set all the variables as mentioned in the fifth project.

The current threshold values are set close to 0 V when converted form the 14-bit ADC value. Depending on your jumper settings on the board the maximal values are:

+/- 2^(14-1) = +/-8192 corresponds to +/-1 V for LV or +/-23 V for HV jumper setting.

I aimed for threshold to be exactly at zero, however, when working with the averager FPGA program I realized that 0V corresponds to roughly -100 ADC value due to a small dc drift of the ADC.

I hope this helps you continue your project.

Best

Antoine

February 1, 2018 at 12:54 am

Reply

Hi, this is great work, thanks.

How hard would it be to include the frequency counter in the existing red pitaya fpga project ?

Antoine

jose

February 27, 2018 at 12:39 pm

Reply

where do you get the pow 2 RTL

Anton Potočnik

February 27, 2018 at 11:05 pm

Reply

The verilog code for pow2 RTL module can be found in the

```
redpitaya_guide/projects/4_frequency_counter/
```

folder.

jose

February 26, 2018 at 4:19 pm

Reply

Hi Anton,

I generated the custume IP core but I don’t understand what you mean by appropriately modifying the make_project.tcl script and execute it in Vivado’s tcl

console with

1

source make_project.tcl

Anton Potočnik

February 27, 2018 at 11:02 pm

Reply

Dear Jose,

Thanks for pointing out this part of the text. What I meant by appropriately modifying the script is to uncomment line 14 in the make_project.tcl file so that when you run

```
source make_project.tcl
```

the fourth project will be created.

I will try to improve the text.

Best

Abinaya

March 9, 2018 at 1:59 pm

Reply

Hi Anton,

I have created the AXI Interface IP for frequency counter logic.In this I am trying to read the count value in XSDK. But I am always reading “0” from this particular address.

Thanks,

Abinaya

Anton Potočnik

March 13, 2018 at 10:16 am

Reply

Hi, I am not sure what you mean with XSDK. But when you generate your bit-file correctly and upload it on the fpga, running the c program on the github should help you start and stop the measurement.

Abinaya P

March 13, 2018 at 10:43 am

Reply

Hi Anton,

XSDk-Xilinx Software Development Kit. In this we can export our hardware and run the C code here instead of linux terminal.

https://www.xilinx.com/products/design-tools/embedded-software/sdk.html

---

Felix

May 1, 2018 at 11:05 am

Reply

Hi Anton, thanks for your awesome tutorial.

We trying to build projects 4 and 5, I get the following error:

ERROR: [BD 5-390] IP definition not found for VLNV: pavel-demin:user:axis_red_pitaya_adc:*

This is in line 69 of the basic_red_pitaya_bd.tcl:

create_bd_cell -type ip -vlnv pavel-demin:user:axis_red_pitaya_adc axis_red_pitaya_adc_0

I'm using Vivado 2017.2

Any ideas what this might be?

Thanks, Felix

Anton Potočnik

July 1, 2018 at 1:58 pm

Reply

Hi, you may need to recompile ip cores. See project 4 and check if there are any errors during compilation of ip cores.

Lasse Tim Christensen

June 29, 2018 at 4:14 pm

Reply

Hi Anton!

Great series of post, the best that i have been able to find so far.

I have one question, which i believe to be rather simple. You write the following: "Frequency, amplitude and other parameters can be set in the Direct Digital Synthesizer (DDS) re-customization dialog" And i can easily spot how to change the frequency, but was not able to find out how to change the amplitude? Would you be so kind to elaborate on that, so i can set a desired amplitude?

Thanks in advance.

Lasse

Anton Potočnik

July 1, 2018 at 1:20 pm

Reply

Hi, you would need to multiply the DDS output with your factor.

See https://forums.xilinx.com/t5/Welcome-Join/dds-amplitude-problem/td-p/731387.

Lasse Tim Christensen

July 2, 2018 at 4:34 pm

Reply

Thanks for the fast reply.

I have previously looked at the discussion which you send a link to, but could not figure out of to implement it to this case. I have looked into several multiply modules, but cant figure out which one to use. I desired to lower the amplitude, and will therefore like to multiply with a value > 1 and < 2. Since the output is given as a twos complement i dont see how a can multiply it with a fraction?

Thanks!

Lasse

Lasse Til Christensen

July 3, 2018 at 3:21 pm

Reply

I mean a value > 0 and < 1 off course 🙂

Lasse

July 12, 2018 at 12:53 pm

Reply

I have solved this problem, thanks for your help Anton.

Andre Himanshu

July 1, 2018 at 7:10 pm

Reply

Hi Anton,

Great work with the tutorials. I have worked in the past with FPGA and writing VHDL codes.

I see that you have extensively used C codes in your project.

My question is from where can such advacned c programming be learnt so that people who less experience can learn and implement advance interfaces using C codes.

Thanks

Anton Potočnik

July 1, 2018 at 10:33 pm

Reply

Hi Andre, I learned C/C++ many years ago mostly by searching on-line how to solve particular coding problems. For complete beginners I am sure google will find a number of good on-line tutorials. Some of the Red Pitaya specific C lines I took from Red Pitaya documentation/forum and Pavel Demin's webpage.

Andre Himanshu

July 2, 2018 at 7:08 am

Reply

Hi Anton,

Thanks for the reply. I will target socket programming in C language and memory operations.

Also as per my understanding if I need to develop some custom program for RP, I need to import the constraint files (port.xdc and clock.xdc) from your projects and can start the soc development.

Pls let me know if I am correct. I am planning to implement project from https://www.youtube.com/watch?v=kxvEcCchRrI (TU kaiserlaten Sadri's program) on RP.

Thanks for your information

Ignaz

August 14, 2018 at 1:39 pm

Reply

Hi Anton,

first of all: thank You very much for these great tutorials. These help me a lot!

I have a short verilog question: In the state-machine in your frequency_counter.v line 100 to 115, You have this combinational logic "always" block:

always @* // logic for counter, counter_output, and cycle buffer

begin

counter_next = counter + 1; // increment on each clock cycle

counter_output_next = counter_output;

cycle_next = cycle;

if (state = Ncycles-1)

begin

counter_next = 0;

counter_output_next = counter;

cycle_next = 0;

end

end

Why does i.e. the

“counter_next = counter + 1” (which is always “assigned”)

not collide with the

” counter_next = 0;” (which is only “assigned” if (state < state_next)

Is there some priority of "assignment" in verilog.

Thanks in advance.

Ignaz

Anton Potočnik

August 14, 2018 at 5:43 pm

Reply

Hey, yes, you are right, blocking assignments happen sequentially.

Have a look at https://class.ee.washington.edu/371/peckol/doc/Always@.pdf

Ignaz

August 15, 2018 at 9:39 am

Reply

Thank You, Anton.

Shawn

December 26, 2018 at 4:42 pm

Reply

Another interesting technique is to assign everything a default state at the top of an always block, then only pick cases that are different and assign there....

ex.

always @(posedge clk) begin

x<= 0;

y<= 0;

z<= 1;

if(ctrl == 2b'00) begin

x <= 1;

end else if(ctrl == 2b'01) begin

y <= 1;

end else if(ctrl == 2'b10) begin

z <= 0;

end

end

helps make sure you don't forget about a signal in a big case statement or whatever...

Leave a Reply

Your email address will not be published. Required fields are marked *

COMMENT

NAME *

EMAIL *

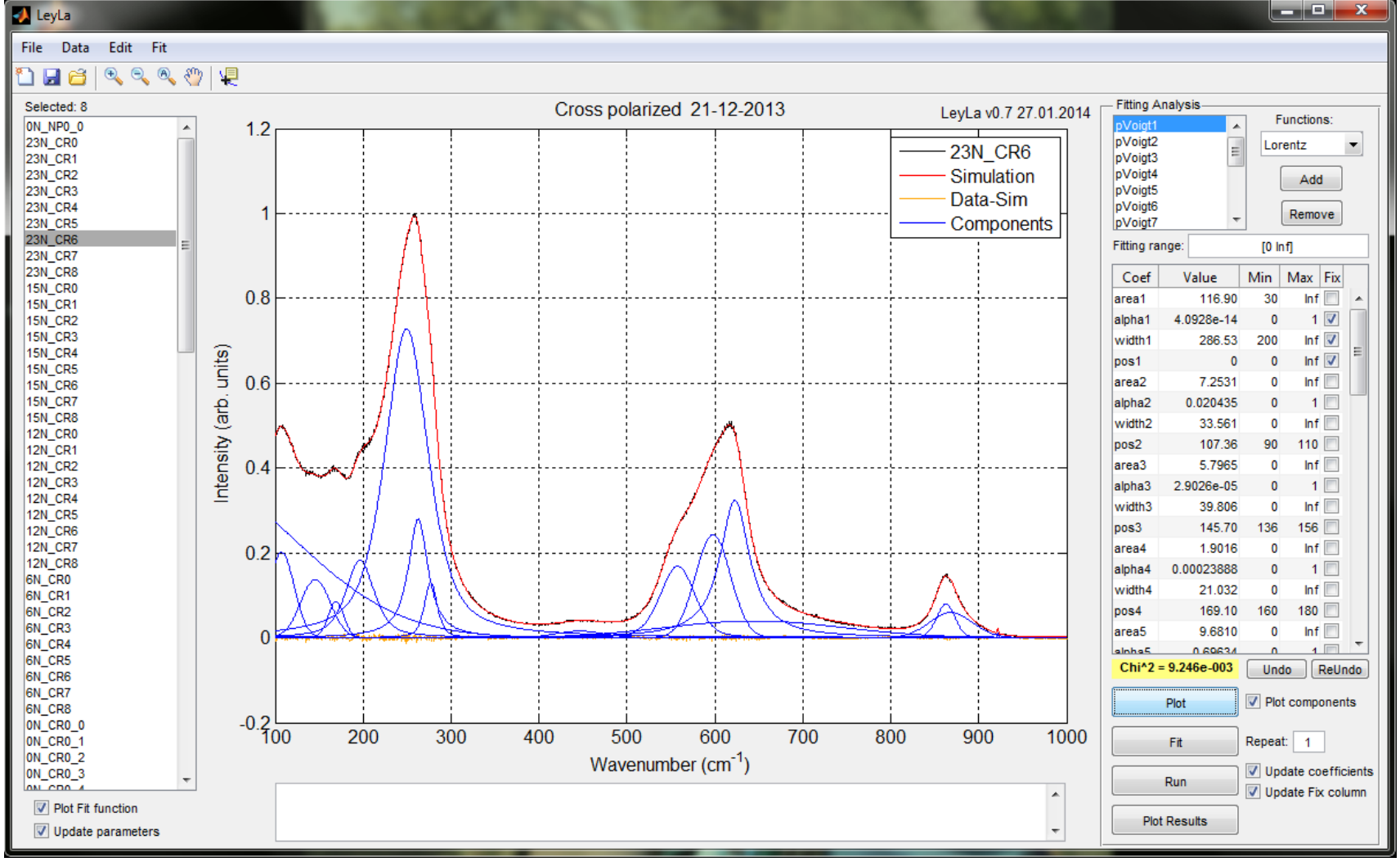
WEBSITE

NOTIFY ME OF FOLLOW-UP COMMENTS BY EMAIL.

NOTIFY ME OF NEW POSTS BY EMAIL.

Post Comment

Recent Posts


LeyLa
File Data Edit Fit
Selected: 8
0N_NP0_0
23N_CR0
23N_CR1
23N_CR2
23N_CR3
23N_CR4
23N_CR5
23N_CR6
23N_CR7
23N_CR8
15N_CR0
15N_CR1
15N_CR2
15N_CR3
15N_CR4
15N_CR5
15N_CR6
15N_CR7
15N_CR8
12N_CR0
12N_CR1
12N_CR2
12N_CR3
12N_CR4
12N_CR5
12N_CR6
12N_CR7
12N_CR8
6N_CR0
6N_CR1
6N_CR2
6N_CR3
6N_CR4
6N_CR5
6N_CR6
6N_CR7
6N_CR8
0N_CR0_0
0N_CR0_1
0N_CR0_2
0N_CR0_3
Plot Fit function
Update parameters
Cross polarized 21-12-2013
LeyLa v0.7 27.01.2014
23N_CR6
Simulation
Data-Sim
Components
Intensity (arb. units)
Wavenumber (cm-1)
Fitting Analysis
pVoigt1
pVoigt2
pVoigt3
pVoigt4
pVoigt5
pVoigt6
pVoigt7
Functions:
Lorentz
Add
Remove
Fitting range:
[0 Inf]
Coef Value Min Max Fix
area1 116.90 30 Inf
alpha1 4.0928e-14 0 1
width1 286.53 200 Inf
pos1 0 0 Inf
area2 7.2531 0 Inf
alpha2 0.020435 0 1
width2 33.561 0 Inf
pos2 107.36 90 110
area3 5.7965 0 Inf
alpha3 2.9026e-05 0 1
width3 39.806 0 Inf
pos3 145.70 136 156
area4 1.9016 0 Inf
alpha4 0.00023888 0 1
width4 21.032 0 Inf
pos4 169.10 160 180
area5 9.6810 0 Inf
Chi^2 = 9.246e-003
Undo
ReUndo
Plot
Plot components
Fit
Repeat: 1
Run
Update coefficients
Update Fix column
Plot Results