Start programming FPGA using Red Pitaya board

by Zumy Topcagic

1. Introduction

In this article we will quickly show how to work with FPGA using Red Pitaya STEMlab board. If you are beginner in FPGA as I am this article will help as it is written *“from beginner to beginner”*. Red Pitaya STEMlab board development environment is quite hard to grasp. The main cause for this is the fact that Red Pitaya STEMlab board do not have an BOARD FILE supported in Vivado environment, i.e **you can’t** (Start Vivado Start New Project -> Select development board -> New block design) and begin to putting IP blocks and programing FPGA as you can for example with the Zedboard. There are some community projects enabling this approach on the Red Pitaya STEMlab board but often this projects are simplified where other features of the Red Pitaya STEMlab board are disabled. I hope that our developers will, soon as possible, take a step further and support the Red Pitaya STEMlab board enabling Vivado Block Design Environment. Notice: You still need to use Vivado SDK for the FPGA programming but you work with directly with Verilog scripts and not with an IP blocks. Notice: You can start Red Pitaya STEMlab board FPGA project using Vivado GUI interface and start putting some IP blocks but probably your IP blocks will not be connected to correct or any pins and with that your design will not work. Also for working on the RedPitaya boards you will need to use Linux, Ubuntu OS. For those who are not familiar with the Linux/Ubuntu do not worry. Basically for this FPGA project a copy-paste of the few commands will be enough.

2. Red Pitaya STEMlab FPGA

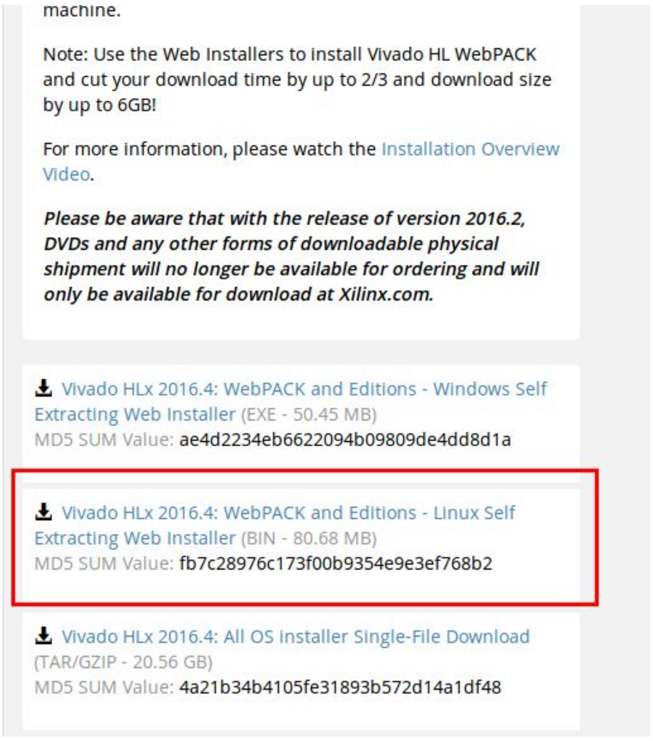
Complete documentation on Red Pitaya STEMlab board is available on this link. For us only a part of the complete documentation is important. When starting with the FPGA programming on the Red Pitaya STEMlab board first step is to understand basic design characteristic:

1. Once the FPGA bitstream is loaded on the FPGA it will start working immediately and independent of any other process on the board.
2. After bitstream is loaded it is manipulated through registers
3. On the default OS image(prepared SD card) there are more FPGA bitstream images since each application will load its own FPGA image.
4. Bitstream can be loaded manually but this can cause system crashing if there is some web application running when fpga loading is executed.
5. If you load your bitstream image and in order to see changes made you start certain application, for example Oscilloscope, no changes will be observable since when Oscilloscope application is started its default bitstream is loaded to the FPGA.
6. We can set (easily) that when Oscilloscope is started it will load our new bitstream and any changes made on the FPGA will be visible directly on the Oscilloscope application. This is useful when playing with ADC and DAC.

3. Let's start

Before we start working with the FPGA there are some prerequisites in order to have successful project. I will put them here with short description but on each separate bullet point you can find additional help on the internet.

1. Install Linux [Ubuntu 16.04 LTS]() on your machine/pc. If you can't separate from Windows then I [suggest dual boot option]() (having Ubuntu and Windows). You can also use Virtual Machine option, but this tends to be slow sometimes.
2. Install *Xilinx [Vivado 2016.4 (including SDK)]().*
   Go to the Xilinx web page, [create an account]() and download [Vivado.]()
   - Download Web Installer



machine.

Note: Use the Web Installers to install Vivado HL WebPACK and cut your download time by up to 2/3 and download size by up to 6GB!

For more information, please watch the Installation Overview Video.

Please be aware that with the release of version 2016.2, DVDs and any other forms of downloadable physical shipment will no longer be available for ordering and will only be available for download at Xilinx.com.

Vivado HLx 2016.4: WebPACK and Editions - Windows Self Extracting Web Installer (EXE - 50.45 MB)
MD5 SUM Value: ae4d2234eb6622094b09809de4dd8d1a

Vivado HLx 2016.4: WebPACK and Editions - Linux Self Extracting Web Installer (BIN - 80.68 MB)
MD5 SUM Value: fb7c28976c173f00b9354e9e3ef768b2

Vivado HLx 2016.4: All OS installer Single-File Download (TAR/GZIP - 20.56 GB)
MD5 SUM Value: 4a21b34b4105fe31893b572d14a1df48

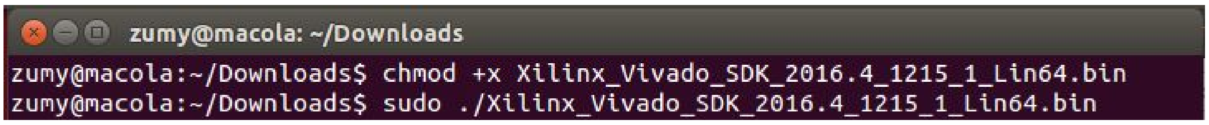
Figure 1. Downloading Web Installer for Vivado Environment

   - Open Terminal, move to Downloads directory and make a downloaded **.bin** file an executable one with following command:

```
chmod +x Xilinx_Vivado_SDK_2016.4_1215_1_Lin64.bin
```

   - Run the executable to start installation with following command:

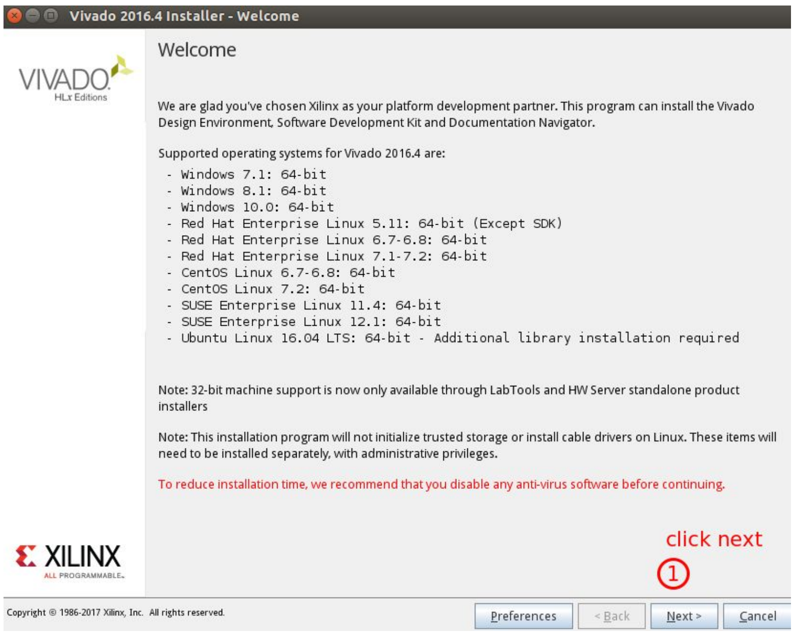
```
sudo ./Xilinx_Vivado_SDK_2016.4_1215_1_Lin64.bin
```



```
zumy@macola: ~/Downloads
zumy@macola:~/Downloads$ chmod +x Xilinx_Vivado_SDK_2016.4_1215_1_Lin64.bin
zumy@macola:~/Downloads$ sudo ./Xilinx_Vivado_SDK_2016.4_1215_1_Lin64.bin
```

Figure 2: Starting Vivado Environment Installation. Terminal commands

After executing installation commands Vivado installer should start. Follow the settings on picture bellow and installation should complete.



Vivado 2016.4 Installer - Welcome

Welcome

We are glad you've chosen Xilinx as your platform development partner. This program can install the Vivado Design Environment, Software Development Kit and Documentation Navigator.

Supported operating systems for Vivado 2016.4 are:

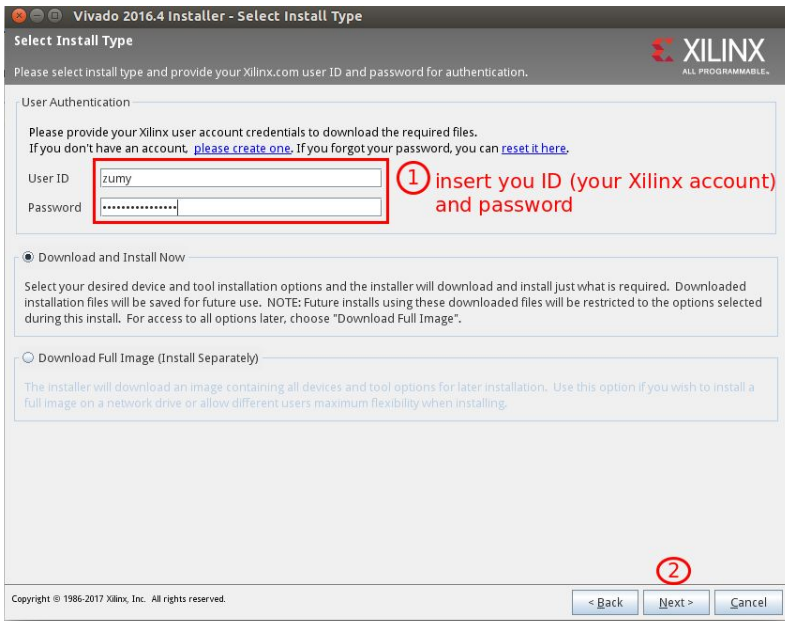
```
 - Windows 7.1: 64-bit
 - Windows 8.1: 64-bit
 - Windows 10.0: 64-bit
 - Red Hat Enterprise Linux 5.11: 64-bit (Except SDK)
 - Red Hat Enterprise Linux 6.7-6.8: 64-bit
 - Red Hat Enterprise Linux 7.1-7.2: 64-bit
 - CentOS Linux 6.7-6.8: 64-bit
 - CentOS Linux 7.2: 64-bit
 - SUSE Enterprise Linux 11.4: 64-bit
 - SUSE Enterprise Linux 12.1: 64-bit
 - Ubuntu Linux 16.04 LTS: 64-bit - Additional library installation required
```

Note: 32-bit machine support is now only available through LabTools and HW Server standalone product installers

Note: This installation program will not initialize trusted storage or install cable drivers on Linux. These items will need to be installed separately, with administrative privileges.

To reduce installation time, we recommend that you disable any anti-virus software before continuing.

Figure 3: Vivado Installer Step 1.



Vivado 2016.4 Installer - Select Install Type

Select Install Type

Please select install type and provide your Xilinx.com user ID and password for authentication.

User Authentication

Please provide your Xilinx user account credentials to download the required files.
If you don't have an account, please create one. If you forgot your password, you can reset it here.

User ID: zumy
Password: ••••••••••••••••

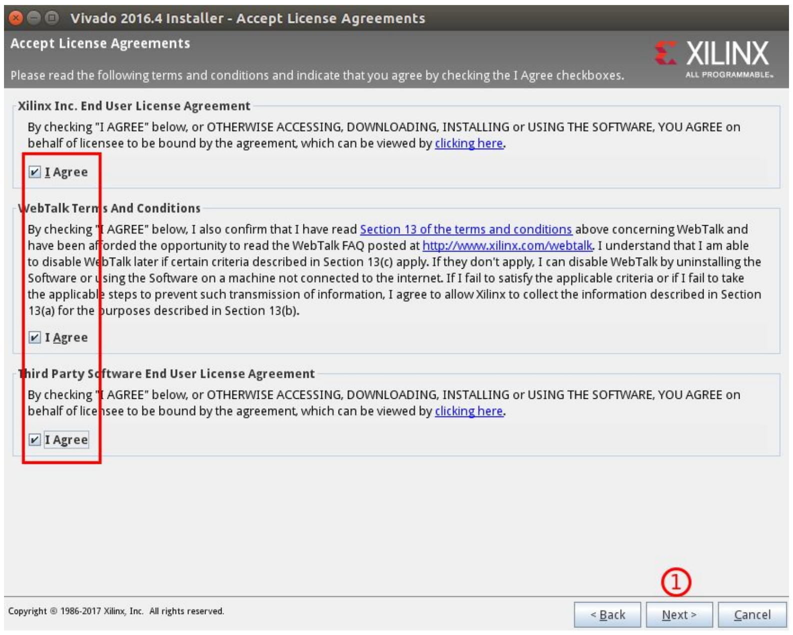
(●) Download and Install Now

Select your desired device and tool installation options and the installer will download and install just what is required. Downloaded installation files will be saved for future use. NOTE: Future installs using these downloaded files will be restricted to the options selected during this install. For access to all options later, choose "Download Full Image".

( ) Download Full Image (Install Separately)

The installer will download an image containing all devices and tool options for later installation. Use this option if you wish to install a full image on a network drive or allow different users maximum flexibility when installing.

Figure 4: Vivado Installer Step 2.


Vivado 2016.4 Installer - Accept License Agreements

Accept License Agreements

Please read the following terms and conditions and indicate that you agree by checking the I Agree checkboxes.

Xilinx Inc. End User License Agreement

By checking "I AGREE" below, or OTHERWISE ACCESSING, DOWNLOADING, INSTALLING or USING THE SOFTWARE, YOU AGREE on behalf of licensee to be bound by the agreement, which can be viewed by clicking here.

☑ I Agree

WebTalk Terms And Conditions

By checking "I AGREE" below, I also confirm that I have read Section 13 of the terms and conditions above concerning WebTalk and have been afforded the opportunity to read the WebTalk FAQ posted at http://www.xilinx.com/webtalk. I understand that I am able to disable WebTalk later if certain criteria described in Section 13(c) apply. If they don't apply, I can disable WebTalk by uninstalling the Software or using the Software on a machine not connected to the internet. If I fail to satisfy the applicable criteria or if I fail to take the applicable steps to prevent such transmission of information, I agree to allow Xilinx to collect the information described in Section 13(a) for the purposes described in Section 13(b).

☑ I Agree

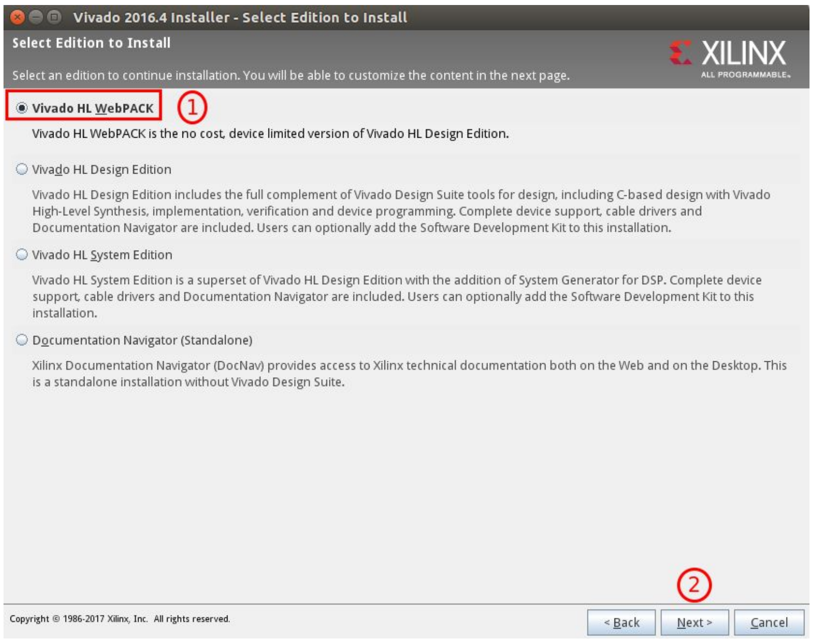
Third Party Software End User License Agreement

By checking "I AGREE" below, or OTHERWISE ACCESSING, DOWNLOADING, INSTALLING or USING THE SOFTWARE, YOU AGREE on behalf of licensee to be bound by the agreement, which can be viewed by clicking here.

☑ I Agree

Copyright © 1986-2017 Xilinx, Inc. All rights reserved.

Figure 5: Vivado Installer Step 3


Vivado 2016.4 Installer - Select Edition to Install

Select Edition to Install

Select an edition to continue installation. You will be able to customize the content in the next page.

● Vivado HL WebPACK

Vivado HL WebPACK is the no cost, device limited version of Vivado HL Design Edition.

○ Vivado HL Design Edition

Vivado HL Design Edition includes the full complement of Vivado Design Suite tools for design, including C-based design with Vivado High-Level Synthesis, implementation, verification and device programming. Complete device support, cable drivers and Documentation Navigator are included. Users can optionally add the Software Development Kit to this installation.

○ Vivado HL System Edition

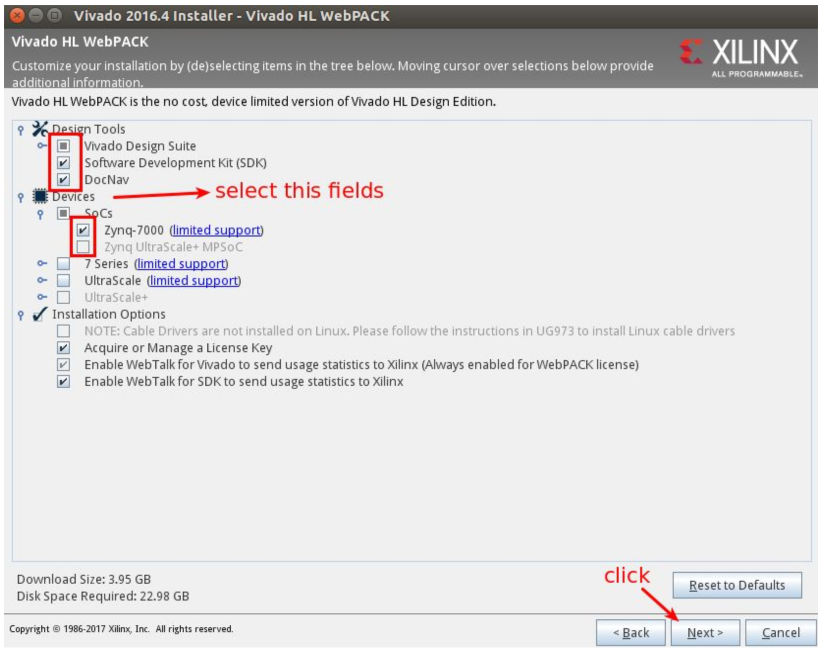
Vivado HL System Edition is a superset of Vivado HL Design Edition with the addition of System Generator for DSP. Complete device support, cable drivers and Documentation Navigator are included. Users can optionally add the Software Development Kit to this installation.

○ Documentation Navigator (Standalone)

Xilinx Documentation Navigator (DocNav) provides access to Xilinx technical documentation both on the Web and on the Desktop. This is a standalone installation without Vivado Design Suite.

Copyright © 1986-2017 Xilinx, Inc. All rights reserved.

Figure 6: Vivado Installer Step 4



Vivado 2016.4 Installer - Vivado HL WebPACK

Vivado HL WebPACK

Customize your installation by (de)selecting items in the tree below. Moving cursor over selections below provide additional information.

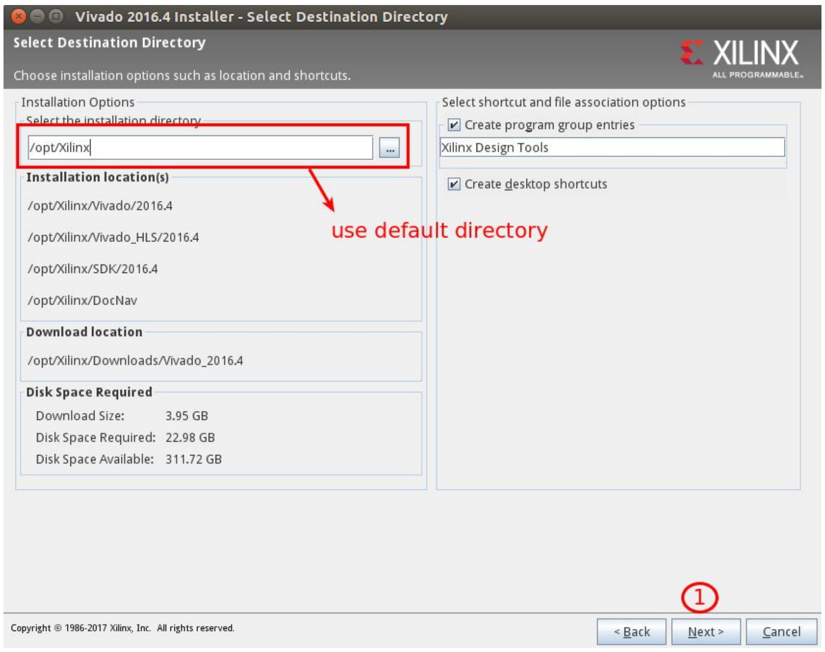
Vivado HL WebPACK is the no cost, device limited version of Vivado HL Design Edition.

- Design Tools
  - ▣ Vivado Design Suite
  - ✓ Software Development Kit (SDK)
  - ✓ DocNav
- Devices
  - ▣ SoCs
    - ✓ Zynq-7000 (limited support)
    - ☐ Zynq UltraScale+ MPSoC
  - ☐ 7 Series (limited support)
  - ☐ UltraScale (limited support)
  - ☐ UltraScale+
- ✓ Installation Options
  - ☐ NOTE: Cable Drivers are not installed on Linux. Please follow the instructions in UG973 to install Linux cable drivers
  - ✓ Acquire or Manage a License Key
  - ✓ Enable WebTalk for Vivado to send usage statistics to Xilinx (Always enabled for WebPACK license)
  - ✓ Enable WebTalk for SDK to send usage statistics to Xilinx

Download Size: 3.95 GB
Disk Space Required: 22.98 GB

Copyright © 1986-2017 Xilinx, Inc. All rights reserved.

Figure 7: Vivado Installer Step 4



Vivado 2016.4 Installer - Select Destination Directory

Select Destination Directory

Choose installation options such as location and shortcuts.

Installation Options

Select the installation directory

/opt/Xilinx

Installation location(s)

/opt/Xilinx/Vivado/2016.4

/opt/Xilinx/Vivado_HLS/2016.4

/opt/Xilinx/SDK/2016.4

/opt/Xilinx/DocNav

Download location

/opt/Xilinx/Downloads/Vivado_2016.4

Disk Space Required

Download Size:	3.95 GB
Disk Space Required:	22.98 GB
Disk Space Available:	311.72 GB

Select shortcut and file association options

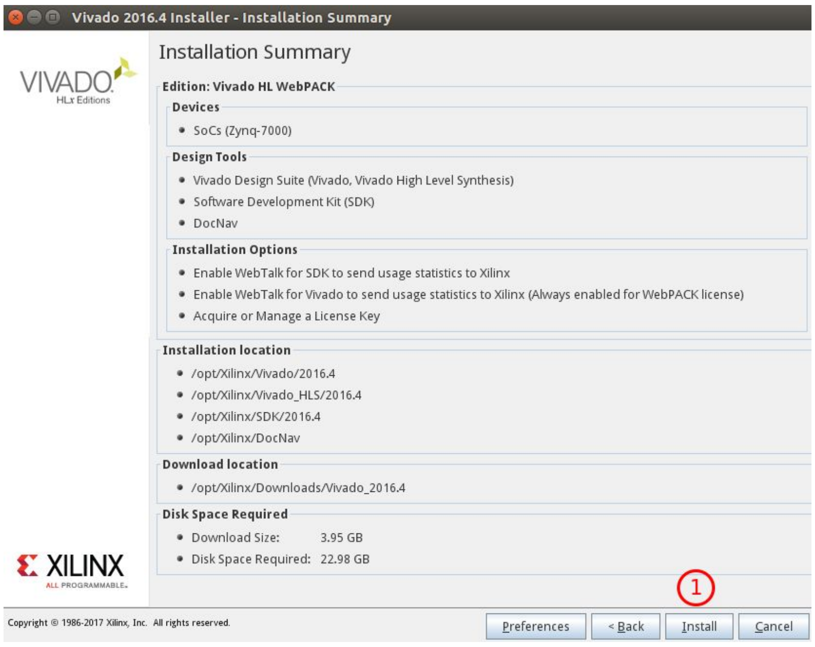
✓ Create program group entries

Xilinx Design Tools

✓ Create desktop shortcuts

Copyright © 1986-2017 Xilinx, Inc. All rights reserved.

Figure 8: Vivado Installer Step 5



Vivado 2016.4 Installer - Installation Summary

Installation Summary

Edition: Vivado HL WebPACK

Devices

- SoCs (Zynq-7000)

Design Tools

- Vivado Design Suite (Vivado, Vivado High Level Synthesis)
- Software Development Kit (SDK)
- DocNav

Installation Options

- Enable WebTalk for SDK to send usage statistics to Xilinx
- Enable WebTalk for Vivado to send usage statistics to Xilinx (Always enabled for WebPACK license)
- Acquire or Manage a License Key

Installation location

- /opt/Xilinx/Vivado/2016.4
- /opt/Xilinx/Vivado_HLS/2016.4
- /opt/Xilinx/SDK/2016.4
- /opt/Xilinx/DocNav

Download location

- /opt/Xilinx/Downloads/Vivado_2016.4

Disk Space Required

- Download Size: 3.95 GB
- Disk Space Required: 22.98 GB

Copyright © 1986-2017 Xilinx, Inc. All rights reserved.

Figure 9: Vivado Installer Step 6

3. After installing Vivado install additional libraries by executing following command in Terminal

```
apt-get install libxft2 libxft2:i386 lib32ncurses5
```

4. On your PC download latest source code for the Red Pitaya STEMlab boards.

For downloading Red Pitaya source code needed for your FPGA project development first you should prepare working directory on your PC.

- Create working directory in your home directory, here we use “GitHub_RedPitaya”

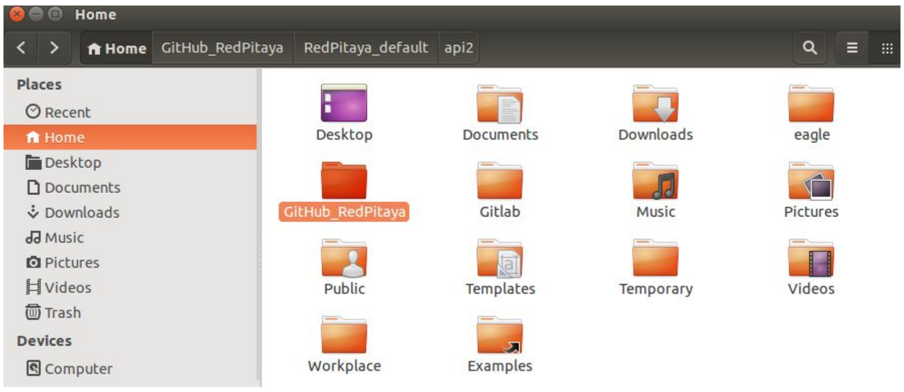
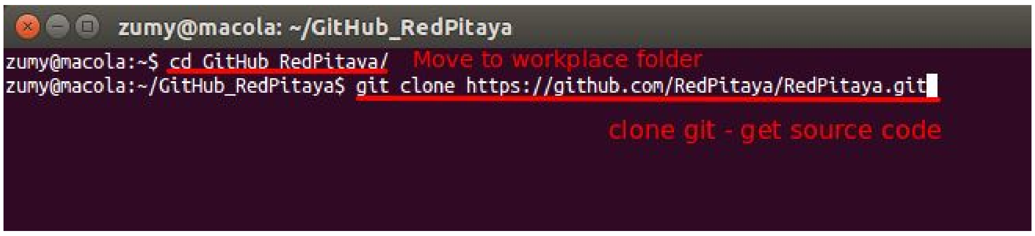


Figure 10: Creating working directory for FPGA source code

- [Download source code](https://github.com/RedPitaya/RedPitaya.git) from Red Pitaya Github repository using **git clone command**

```
git clone https://github.com/RedPitaya/RedPitaya.git
```



```
zumy@macola:~$ cd GitHub_RedPitaya/
zumy@macola:~/GitHub_RedPitaya$ git clone https://github.com/RedPitaya/RedPitaya.git
```

Figure 11: Getting source code using **git clone** command

- When you open working directory "GitHub_RedPitaya", you should see something like this

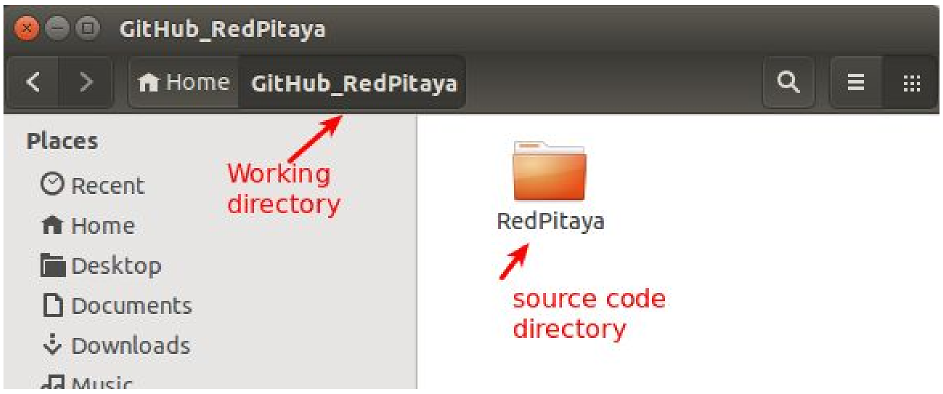


Figure 12: Red Pitaya source code

- In the **RedPitaya** directory is all source code for the Red Pitaya STEMlab boards, from FPGA to the applications. For us only a FPGA source code part is needed.

- FPGA source code is put into **fpga** folder

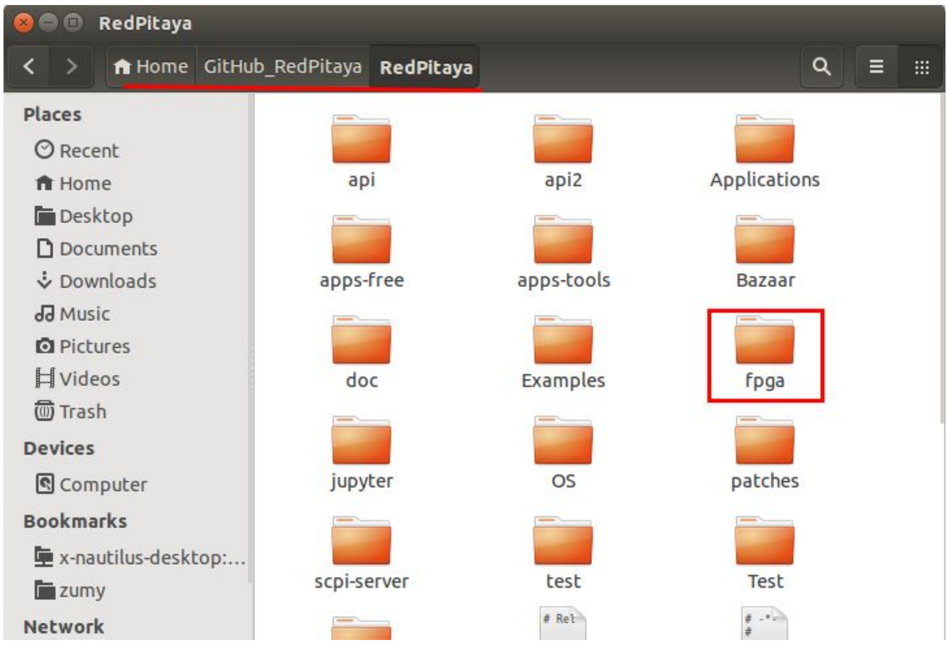
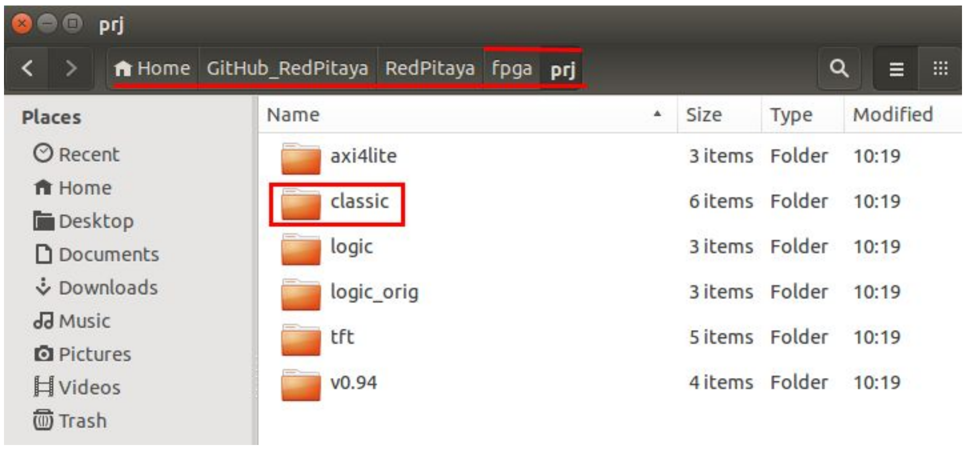


Figure 13: FPGA source code

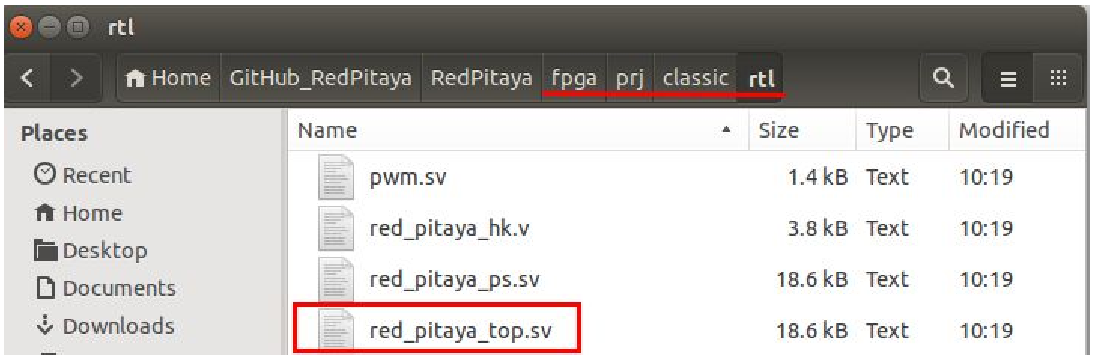
In the **fpga** folder there are many other directories and files but for us the most important are **prj** (as project) and **classic** directories. Classic refers to the the FPGA bitstream source code used by majority of applications and functionalities (Oscilloscope & Signal generator, SCPI server, API-s...). Logic analyzer application for example has different FPGA bitstream and for that there is a **logic** directory containing FPGA source code for the bitstream needed for LA application. (**How to know and how to set which fpga bitstream is loaded when certain application is started will be shown later**)



Name	Size	Type	Modified
axi4lite	3 items	Folder	10:19
classic	6 items	Folder	10:19
logic	3 items	Folder	10:19
logic_orig	3 items	Folder	10:19
tft	5 items	Folder	10:19
v0.94	4 items	Folder	10:19

Figure 14: FPGA source code (classic) used for main fpga image

- **Classic** directory will also contain some sub directories and files but now we have located a **most important directory** for us and that is **rtl. In this directory the MAIN source code file for the classic fpga bitstream is located.**



Name	Size	Type	Modified
pwm.sv	1.4 kB	Text	10:19
red_pitaya_hk.v	3.8 kB	Text	10:19
red_pitaya_ps.sv	18.6 kB	Text	10:19
red_pitaya_top.sv	18.6 kB	Text	10:19

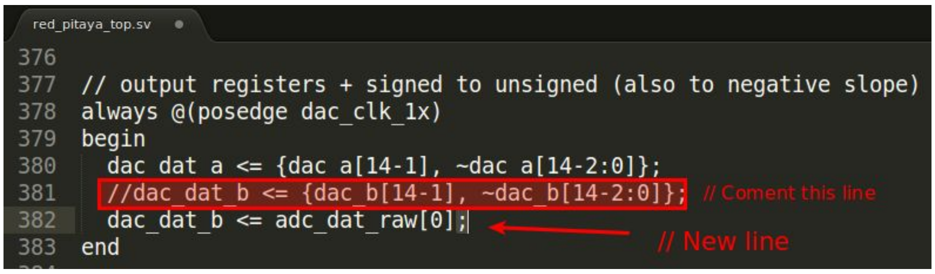
Figure 15: Main source file for classic fpga image

In the “red_pitaya_top.sv” file is the Verilog source code for the classic fpga bitstream. For us as an beginners this is ONLY file which we will EDIT in order to customize our fpga design. Any other files and folders are not in interest for us.

4. Our project

In the sections above we have shown how to prepare source code environment. Now let's pretend that we know something on FPGA and Verilog and make our first project. Normally, first project are something like Blink LED but here we will go and play directly with ADCs and DACs. **Our project is to customize classic fpga bitstream so when Oscilloscope & Signal generator application is started the OUT2 will generate any signal which is acquired on the IN1.** For implementing this we need to custimeze ony a **red_pitaya_top.sv** file and make/build new bitstream.
Let's do this:

1. Open **red_pitaya_top.sv** file usin some text editor
2. Customize the code as is shown in picture bellow.



```
// output registers + signed to unsigned (also to negative slope)
always @(posedge dac_clk_1x)
begin
  dac_dat_a <= {dac_a[14-1], ~dac_a[14-2:0]};
  //dac_dat_b <= {dac_b[14-1], ~dac_b[14-2:0]};
  dac_dat_b <= adc_dat_raw[0];
end
```

Figure 16: Modifying red_pitaya_top.sv file to implement our project

THAT is THAT for our project. Since the script is already written we only customized it a little bit to accommodate our project "**SEND IN1 signal to OUT2**". As you can see from the picture above/code instead of sending **dac_b values to the DAC we send adc_dat_raw values which are samples directly from ADC.**

5. Build/Compile bitstream

How to build FPGA image is described in the documentation. We can skip all step up to building process instructions. Now we will just run commands according to the instructions.

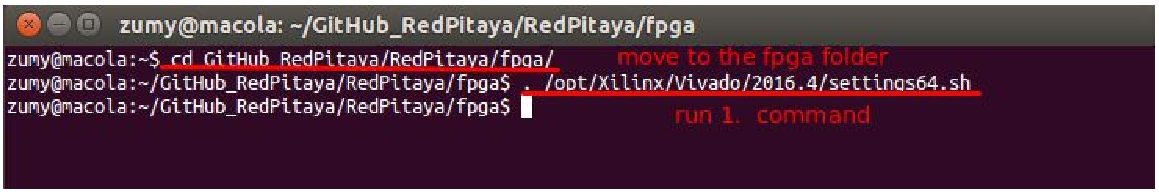
1. Open Terminal
2. Move to the folder **fpga** in the source code and run the commands
   Command1:

   ```
   ./opt/Xilinx/Vivado/2016.4/settings64.sh
   ```

   Command2:

   ```
   make PRJ=classic
   ```

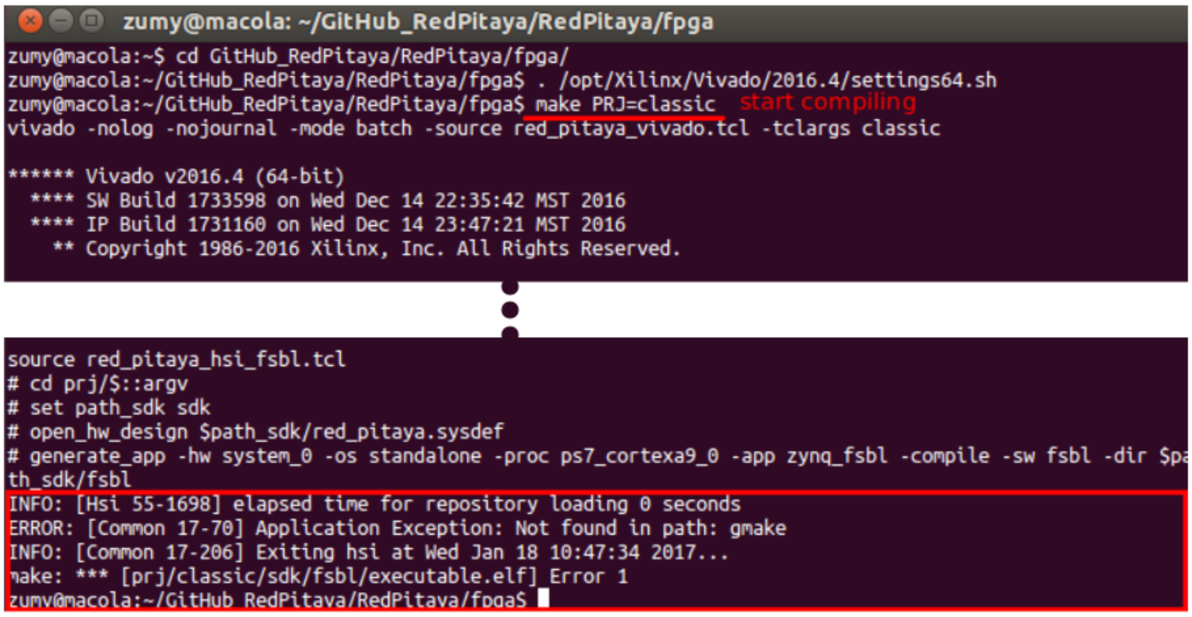
In command 2 we set for **name = classic**, which means we want to build classic bitstream.



```
zumy@macola: ~/GitHub_RedPitaya/RedPitaya/fpga
zumy@macola:~$ cd GitHub_RedPitaya/RedPitaya/fpga/
zumy@macola:~/GitHub_RedPitaya/RedPitaya/fpga$ . /opt/Xilinx/Vivado/2016.4/settings64.sh
zumy@macola:~/GitHub_RedPitaya/RedPitaya/fpga$
```

Figure 17: Move to the **fpga** directory and execute command 1.

Now we can execute **make** command and build our bitstream.



```
zumy@macola: ~/GitHub_RedPitaya/RedPitaya/fpga
zumy@macola:~$ cd GitHub_RedPitaya/RedPitaya/fpga/
zumy@macola:~/GitHub_RedPitaya/RedPitaya/fpga$ . /opt/Xilinx/Vivado/2016.4/settings64.sh
zumy@macola:~/GitHub_RedPitaya/RedPitaya/fpga$ make PRJ=classic
vivado -nolog -nojournal -mode batch -source red_pitaya_vivado.tcl -tclargs classic

****** Vivado v2016.4 (64-bit)
  **** SW Build 1733598 on Wed Dec 14 22:35:42 MST 2016
  **** IP Build 1731160 on Wed Dec 14 23:47:21 MST 2016
    ** Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.

source red_pitaya_hsi_fsbl.tcl
# cd prj/$::argv
# set path_sdk sdk
# open_hw_design $path_sdk/red_pitaya.sysdef
# generate_app -hw system_0 -os standalone -proc ps7_cortexa9_0 -app zynq_fsbl -compile -sw fsbl -dir $pa
th_sdk/fsbl
INFO: [Hsi 55-1698] elapsed time for repository loading 0 seconds
ERROR: [Common 17-70] Application Exception: Not found in path: gmake
INFO: [Common 17-206] Exiting hsi at Wed Jan 18 10:47:34 2017...
make: *** [prj/classic/sdk/fsbl/executable.elf] Error 1
zumy@macola:~/GitHub_RedPitaya/RedPitaya/fpga$
```

Figure 18: Bitstream build process

On picture above is shown what we should get after running make command. When make command is executed alongside our bitstream some other stuff is generated also. **For our project only a new bitstream is important and it can be found in >>out<< directory which is in the >>classic<< directory. The >>out<< directory is generated when make command is executed**.

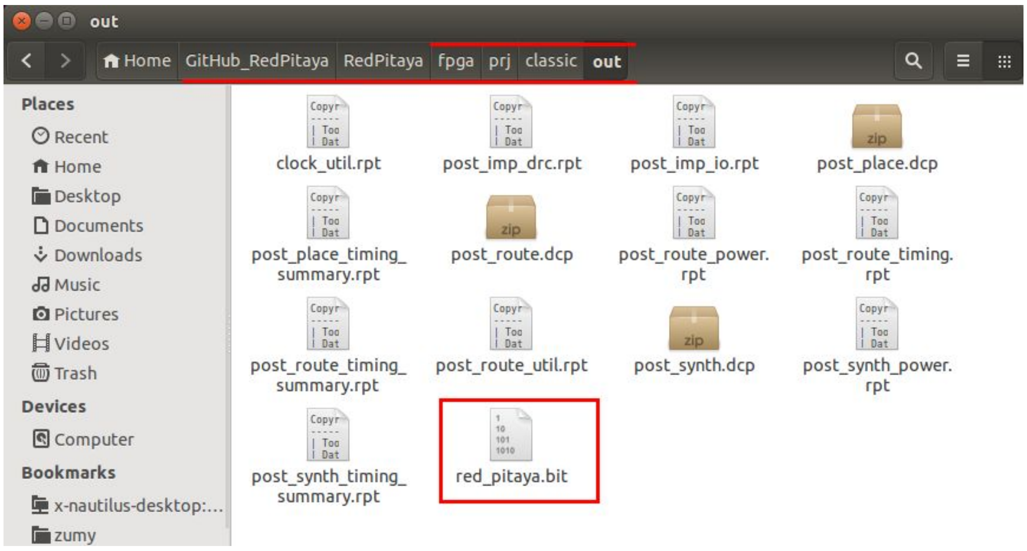


Figure 19: New bitstream located in **fpga/prj/classic/out** directory

6. Copy new bitstream to the Red Pitaya STEMlab board

After we have made a desired modification to the fpga bitstream we can modify existing Oscilloscope application so it will use our new bitstream. This is useful since with an Oscilloscope application we can observe changes made in the fpga bitstream code.
First you need to prepare SD card with the latest OS image from Red Pitaya.
How to prepare SD card is described here.

- After you prepare SD card you should have two partition.
- In FAT (128MB) partition open **fpga** directory.

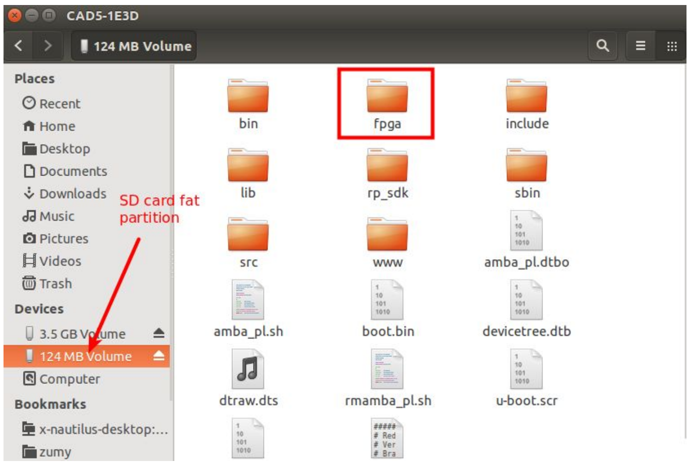


Figure 20: FAT partition on the SD card and fpga directory

- From PC copy new **red_bitaya.bit** bitstream to the **fpga** directory on the SD card.

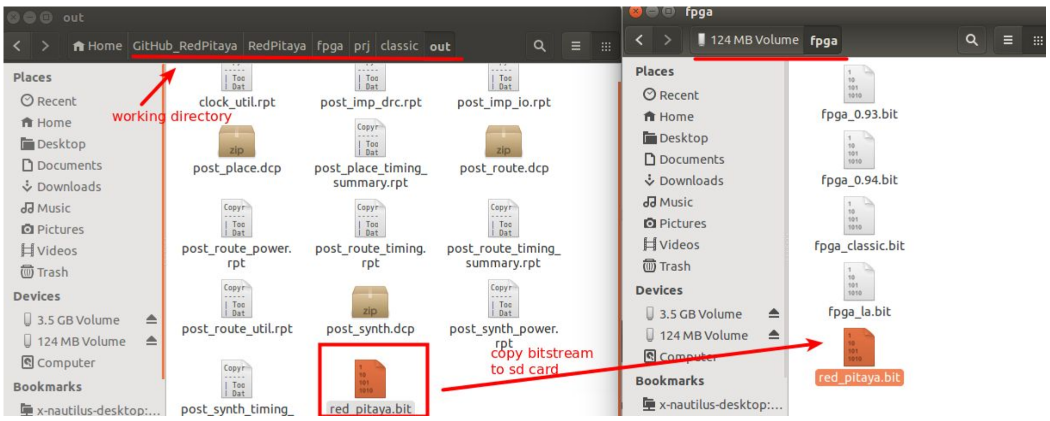


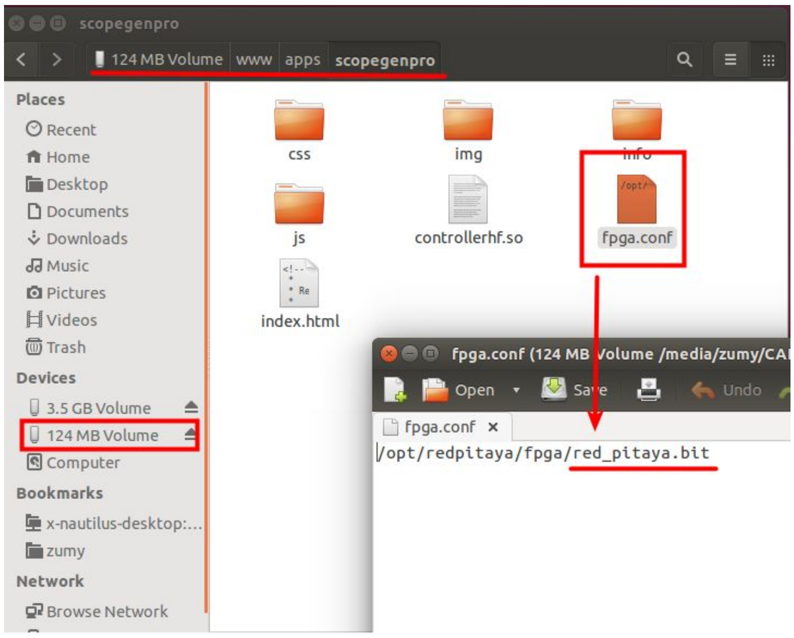
Figure 21: Copy **red_bitaya.bit** bitstream to the SD card

7. Modify existing Oscilloscope application which will work with new bitstream file.

In this step we will modify existing Oscilloscope application so when it is started it will load our new **fpga** bitstream and using it.

- On the SD card open the directory **/www/apps/scopegenpro**
- With preferable text editor open **fpga.conf** file

- Edit the **fpga.conf** file and change it from
  `/opt/redpitaya/fpga/fpga_classic.bit`
  to
  `/opt/redpitaya/fpga/red_pitaya.bit`



```
/opt/redpitaya/fpga/red_pitaya.bit
```

Figure 22: Editing fpga.conf file

8. Start your Red Pitaya STEMlab and observe changes

Take the SD card from your PC and insert it into Red Pitaya STEMlab board. Connect board to your local network as is shown here and start it. To observe changes in fpga code connect Red Pitaya STEMlab inputs and outputs as is shown in picture bellow.

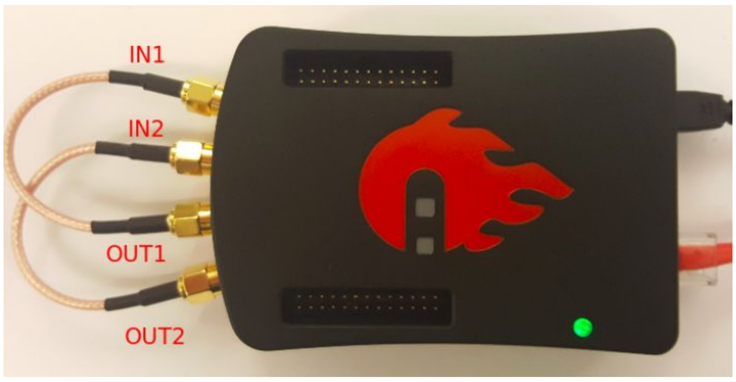


Figure 23: Connections for the measurements OUT1 -> IN1, OUT2 -> IN2

In order to check our code we need to generate signal on OUT1 which will be acquired on IN1. Signal acquired on IN1 will be then sent directly to OUT2. And if our code is working, on the IN2 we should observe same signal as it is on IN1. Since our code modification is setting OUT2 to generate anything what is acquired on IN1.

- Start the Oscilloscope application

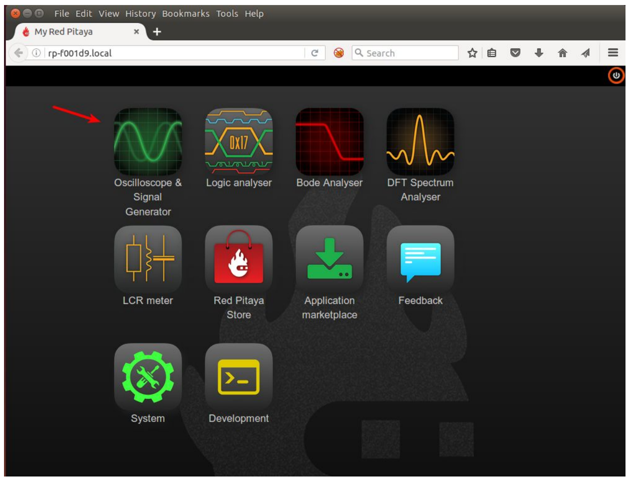
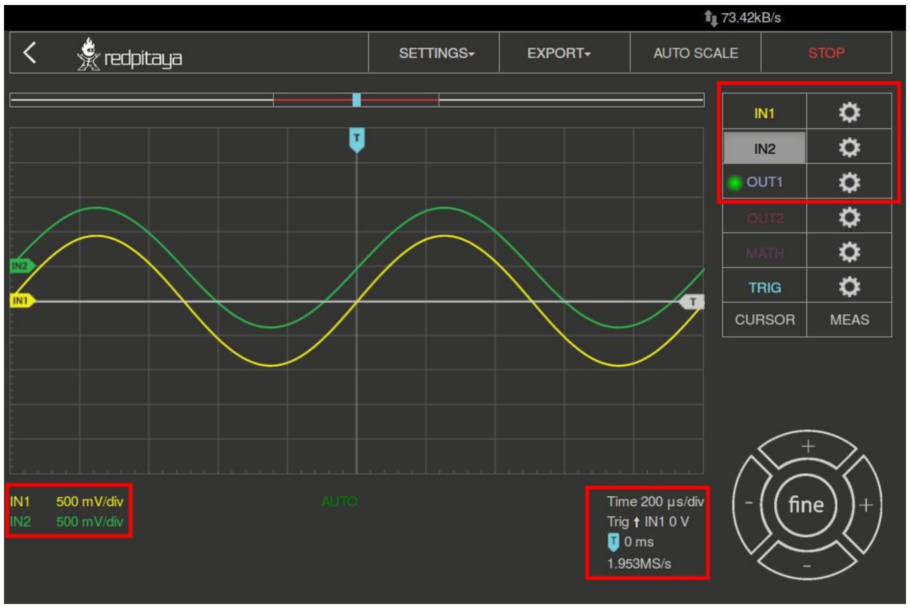


Figure 24: Start Oscilloscope application

- On the OUT1 settings menu deselect SHOW button and select ON button.
- Using horizontal +/- commands set time division to 200us/div
- Using vertical +/- commands set volt division for IN1 and IN2 to 500mV/div
- To see IN2 signal more clearly in IN2 settings set the voltage offset to 500mV



IN1 500 mV/div
IN2 500 mV/div

Time 200 µs/div
Trig ↑ IN1 0 V
T 0 ms
1.953MS/s

Figure 25: Start Oscilloscope application

From picture above you can see that with modifying fpga code we have achieved that OUT2 will generate any signal which is acquired on IN1. In this case, that is signal from OUT1. Modified fpga code is enabling OUT2 signal generation without any additional “enabling” on the application itself. You can try to play with OUT2 settings to see that there is no effect since fpga code is directly bypassing IN1 to OUT2.

Exercise:

1. Try to disconnect cable from IN1.
2. Try to change frequency of OUT1 and see what happens (hint: phase delay)
3. At large frequencies signal on IN2 is distorted. Why? Try to use 50 Ohm termination on OUT1 and OUT2.

NOTICE:
IF you wish to make additional changes and build bitstream you should go from the beginning - cloning the source code from GitHub repository. Normally you should Run >>make clean<< command, make additional changes in red_pitaya_top.sv file and then run >>make<< command to builld new bitstream. But the case is that >>make clean<< command currently is not working and it will not clean all artefacts generated in first >>make<<. Because of that you should always have this steps:

1. Fresh (default) source code
2. Make changes in red_pitaya_top.sv file
3. Run build / make
4. Try new bitstream
5. Go to step 1.

9. More complex project. Set the amplitude of output signal by multiplying it with some constant value.

In the previous example we have shown on how to modify FPGA code in order to send signal from IN1 to OUT2. Now let's continue this project in such way that we send signal from IN1 to OUT2 where we scale IN1 by some scaling factor x.
OUT2 = IN1 * x.
In order to implement this we need to define a constant (scale_factor) in the fpga and multiply this constant with the signal from IN1. In previous example we used **adc_dat_raw[0]** signal which is the signal directly from ADC. Signal values from ADC are sent in a 14 bit format where 1 bit(MSB) is representing a signal sign(-/+) and other 13 bits an amplitude value.

`adc_dat_raw[0] [13]` - MSB (14) bit representing signal sign
`adc_dat_raw[0] [12:0]` - bits 0 to 12 are representing amplitude value.

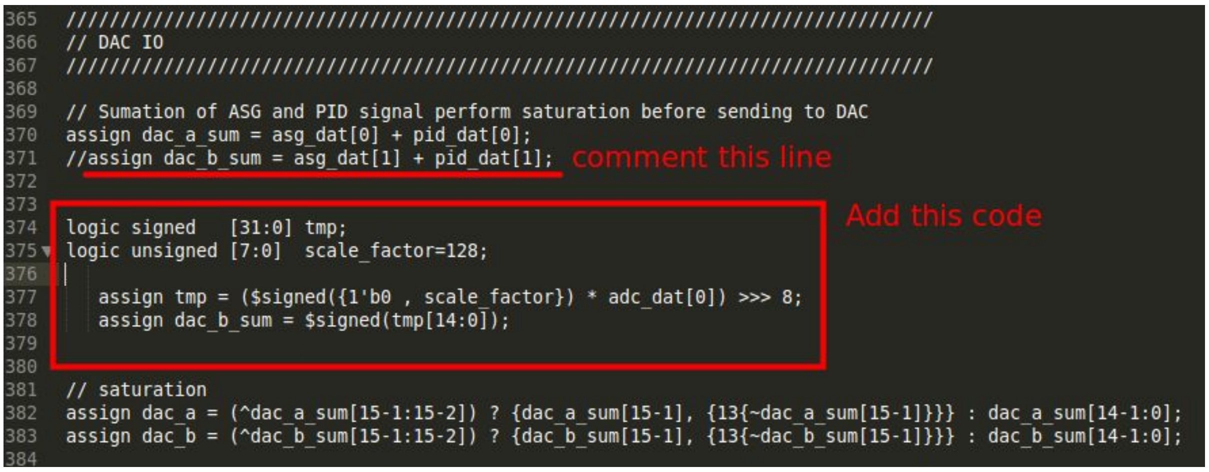
This is the signal format defined by ADC on the Red Pitaya STEMlab board. Also signal to DAC must be sent in the same format. In other words we need to preserve first (MSB) bit for signal sign regardless of the signal amplitude. So when we assigned **adc_dat_raw[0]** directly to **dac_dat_b** everything was okay since ADC and DAC logic was compatible. When we doing some calculation (adding , subtracting, shifting...) in the FPGA logic we use an **two's complement** logic. When using this logic there is no preservation of the first bit since our amplitude/value can be smaller than the desired bit length and we can not (directly) be sure that first bit fill define a sign of the amplitude.

Because of this, when doing some arithmetics on signals from ADC those signals needs to be transformed into two's complement format. After multiplication or similar when sending signals back to DAC we need to transform them back to ADC/DAC format.

In fpga code in **red_pitaya_top.sv** file we can use **adc_dat[0]** signal which is the signal from IN1 (**adc_dat_raw[0]**) transformed into two's complement. On **adc_dat[0]** signal we can apply our arithmetics and send it back to the OUT2 (DAC.)
For sending **adc_dat[0]** to the OUT2 DAC we can use **dac_b_sum** signal, which is the signal for OUT2 transformed into correct format for DAC.
In short our logic is:

dac_b_sum = adc_dat[0] *x.
dac_b_sum -> format correction(dac_a) -> DAC.

Implementation:
Use **default red_pitaya_top.sv file** and modify the code as is shown in picture bellow.



```
////////////////////////////////////////////////////////////////////////////////
// DAC IO
////////////////////////////////////////////////////////////////////////////////

// Sumation of ASG and PID signal perform saturation before sending to DAC
assign dac_a_sum = asg_dat[0] + pid_dat[0];
//assign dac_b_sum = asg_dat[1] + pid_dat[1];


logic signed   [31:0] tmp;
logic unsigned [7:0]  scale_factor=128;

    assign tmp = ($signed({1'b0 , scale_factor}) * adc_dat[0]) >>> 8;
    assign dac_b_sum = $signed(tmp[14:0]);


// saturation
assign dac_a = (^dac_a_sum[15-1:15-2]) ? {dac_a_sum[15-1], {13{~dac_a_sum[15-1]}}} : dac_a_sum[14-1:0];
assign dac_b = (^dac_b_sum[15-1:15-2]) ? {dac_b_sum[15-1], {13{~dac_b_sum[15-1]}}} : dac_b_sum[14-1:0];
```

Figure 26: Modifying red_pitaya_top.sv file to implement signal multiplication

As you can see from picture above we defined **scale_factor** as unsigned 8 bit value. The value of scale factor is 128(decimal). In the **tmp** signal we store the value of multiplication of **adc_dat[0]** with **scale_factor** and sifting with 8 bit in right. (>>> is sign preserving shift in system verilog). In part of the code **{1'b0, scale_factor}** we are adding 1 bit (0) to the scale_factor effectively making the scale_factor of 9 bits length. Adding this bit will preserve or sing(+/-) in the multiplication step. In next step we assign value of **tmp[14:0]** (bits from 0 to 14) to the **dac_b_sum**. **dac_b_sum** sum signal is saturated converted to DAC/ADC format and sent to DAC(**dac_b** signal).
In multiplication of **scale_factor** and **adc_dat[0]** signal we use Q formating in order it implement fixed point multiplication. We have defined **tmp** signal to be 32 bit long since when multiplying two number with certain bit length the results will be an number requiring the bit length equal to the sum of bit lengths of both numbers in the multiplication step.

In our example fe used following:
*fixed_point_scale_fac = x * 2^n*
Where,
n = integer value for our Q format (in our example 8)
x = scaling factor in floating point.

Example:
*fixed_point_scale_fac = 0.5 * 2^8*
fixed_point_scale_fac = 128

When we multiply or signal
adc_dat[0]*128
And shifting this results for 8 bits in right
(adc_dat[0]*128)>>>8
Effectively we are multiplying the adc_dat[0] by 0.5.

For trying this kind of code(multiplication and shifting) i recommend using EDA playground (System Verilog, Synopsis VCS 2015.10 Simulator) as an simulator for your fpga code. Here you can use printf function to check what the result of some multiplication will be. In real design on the FPGA you can't do that simply.

Okay, make this modification and try it out by repeating steps form first part of the tutorial (building image, copy bitstream, start Oscilloscope)

10. Adding registers for our scaling factor.

At the introduction of this tutorial we stated that state of the running fpga can be controlled via register. On Red Pitay STEMlab board we can read/write this registers using monitor terminal/console command. So in order to be able to change scaling factor on running fpga code we need to make a new register and define it as a scaling factor. This is done simply by defining new register in the **red_pitaya_top.sv** file and the **red_pitaya_hk.v** file. **red_pitaya_hk.v** file contains a code for house keeping (LEDs..) and also code for communication between CPU and FPGA.

Implementation.
Use **default red_pitaya_top.sv file** and modify the code as is shown in picture bellow.

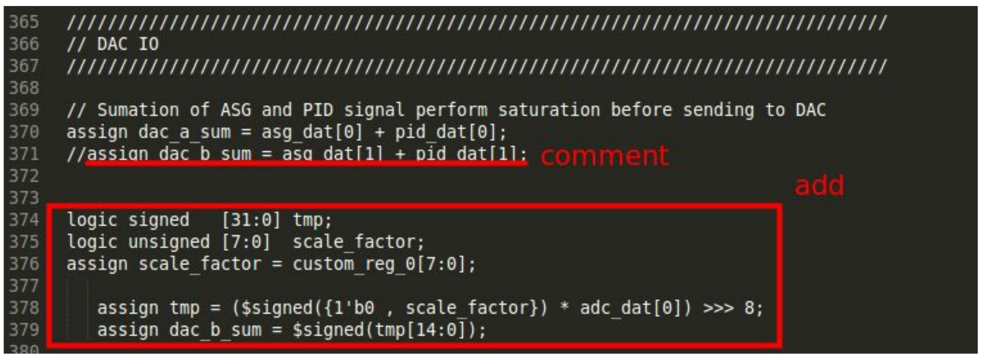
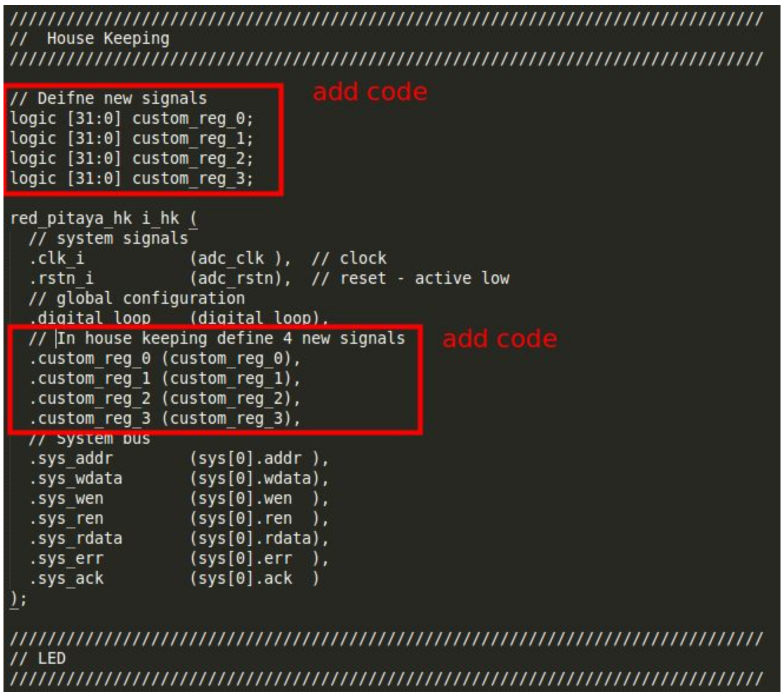

```
////////////////////////////////////////////////////////////////////////////////
// DAC IO
////////////////////////////////////////////////////////////////////////////////

// Sumation of ASG and PID signal perform saturation before sending to DAC
assign dac_a_sum = asg_dat[0] + pid_dat[0];
//assign dac_b_sum = asg_dat[1] + pid_dat[1];

logic signed   [31:0] tmp;
logic unsigned [7:0]  scale_factor;
assign scale_factor = custom_reg_0[7:0];

    assign tmp = ($signed({1'b0 , scale_factor}) * adc_dat[0]) >>> 8;
    assign dac_b_sum = $signed(tmp[14:0]);
```

Figure 27: Modifying red_pitaya_top.sv file to implement scale_factor register



```
////////////////////////////////////////////////////////////////////////////////
//  House Keeping
////////////////////////////////////////////////////////////////////////////////

// Deifne new signals
logic [31:0] custom_reg_0;
logic [31:0] custom_reg_1;
logic [31:0] custom_reg_2;
logic [31:0] custom_reg_3;

red_pitaya_hk i_hk (
  // system signals
  .clk_i           (adc_clk ),  // clock
  .rstn_i          (adc_rstn),  // reset - active low
  // global configuration
  .digital_loop    (digital_loop),
  // In house keeping define 4 new signals
  .custom_reg_0 (custom_reg_0),
  .custom_reg_1 (custom_reg_1),
  .custom_reg_2 (custom_reg_2),
  .custom_reg_3 (custom_reg_3),
  // System bus
  .sys_addr        (sys[0].addr ),
  .sys_wdata       (sys[0].wdata),
  .sys_wen         (sys[0].wen  ),
  .sys_ren         (sys[0].ren  ),
  .sys_rdata       (sys[0].rdata),
  .sys_err         (sys[0].err  ),
  .sys_ack         (sys[0].ack  )
);

////////////////////////////////////////////////////////////////////////////////
// LED
////////////////////////////////////////////////////////////////////////////////
```

Figure 28: Modifying red_pitaya_top.sv file to define new register

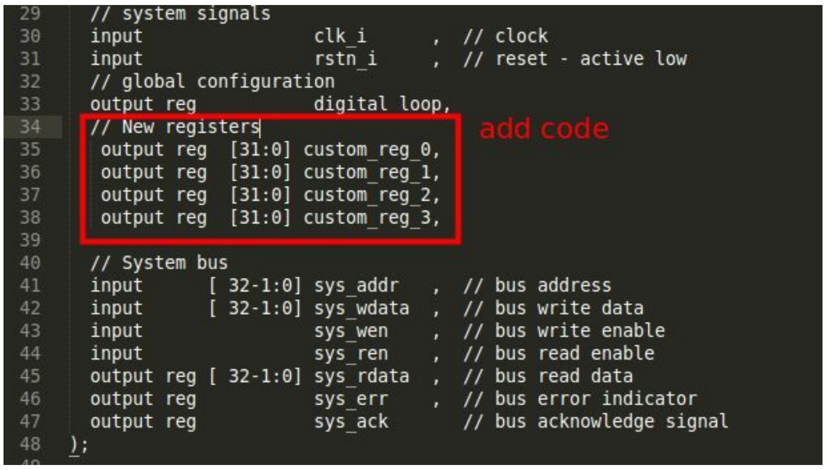
From pictures above you can see that we have created 4 new register; from custom_reg_0 to custom_reg_3. We need and use only one but maybe for next project we need to use more registers. To the **scale_factor** we assigned register custom_reg_0.

```
assign scale_factor = custom_reg_0[7:0];
```

When assigning value of **custom_reg_0** the the scale_factor we use lowest 8 bits of our register **custom_reg_0**. Next step is to define new registers in the **red_pitaya_hk.v** file. Code In this file we will connect our new register to the system bus and also define an address locations of each register.

Use default **red_pitaya_hk.v (it is on the same directory as an red_pitaya_top.sv) .** In this file modify the code as it is shown in pictures bellow. Notice: Red_pitaya_hk.v is written in Verilog (.v) and not SystemVerilog (.sv) so in **red_pitaya_hk.v** we use **reg** instead of **logic.**

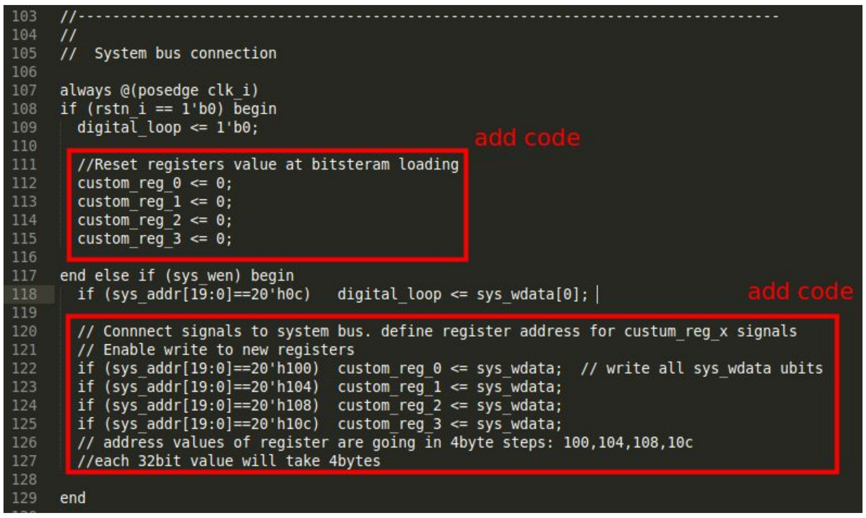
1. Define registers



```
    // system signals
    input                   clk_i      ,  // clock
    input                   rstn_i     ,  // reset - active low
    // global configuration
    output reg              digital_loop,
    // New registers
     output reg  [31:0] custom_reg_0,
     output reg  [31:0] custom_reg_1,
     output reg  [31:0] custom_reg_2,
     output reg  [31:0] custom_reg_3,

    // System bus
    input       [ 32-1:0] sys_addr   ,  // bus address
    input       [ 32-1:0] sys_wdata  ,  // bus write data
    input                 sys_wen    ,  // bus write enable
    input                 sys_ren    ,  // bus read enable
    output reg [ 32-1:0] sys_rdata  ,  // bus read data
    output reg            sys_err    ,  // bus error indicator
    output reg            sys_ack       // bus acknowledge signal
);
```

Figure 29: Define new registers in **red_pitaya_hk.v** file

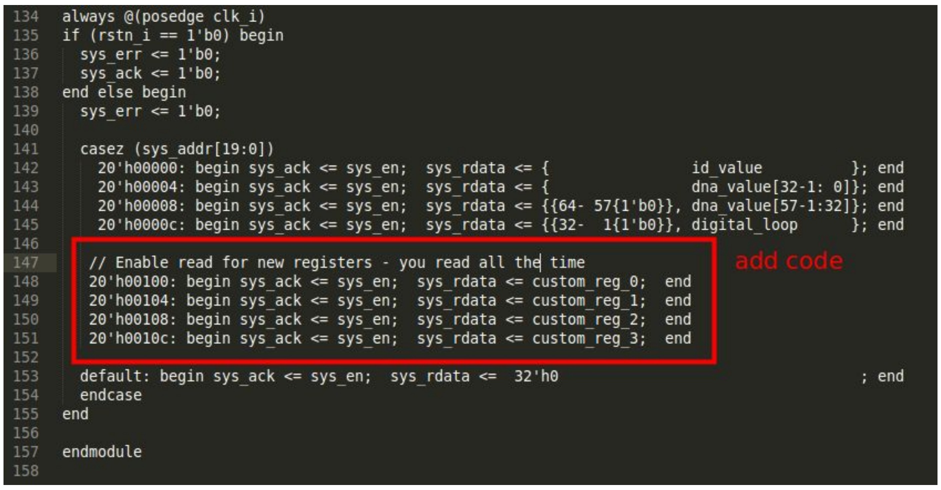
2. Connect new registers to system bus



```
//---------------------------------------------------------------------------------
//
//  System bus connection

always @(posedge clk_i)
if (rstn_i == 1'b0) begin
  digital_loop <= 1'b0;

  //Reset registers value at bitsteram loading
  custom_reg_0 <= 0;
  custom_reg_1 <= 0;
  custom_reg_2 <= 0;
  custom_reg_3 <= 0;

end else if (sys_wen) begin
  if (sys_addr[19:0]==20'h0c)   digital_loop <= sys_wdata[0];

  // Connnect signals to system bus. define register address for custum_reg_x signals
  // Enable write to new registers
  if (sys_addr[19:0]==20'h100)  custom_reg_0 <= sys_wdata;  // write all sys_wdata ubits
  if (sys_addr[19:0]==20'h104)  custom_reg_1 <= sys_wdata;
  if (sys_addr[19:0]==20'h108)  custom_reg_2 <= sys_wdata;
  if (sys_addr[19:0]==20'h10c)  custom_reg_3 <= sys_wdata;
  // address values of register are going in 4byte steps: 100,104,108,10c
  //each 32bit value will take 4bytes

end
```

Figure 30: Set reset of new registers and connect them to system bus in **red_pitaya_hk.v** file

3. Enable read function for our new registers



```
always @(posedge clk_i)
if (rstn_i == 1'b0) begin
  sys_err <= 1'b0;
  sys_ack <= 1'b0;
end else begin
  sys_err <= 1'b0;

  casez (sys_addr[19:0])
    20'h00000: begin sys_ack <= sys_en;  sys_rdata <= {                id_value          }; end
    20'h00004: begin sys_ack <= sys_en;  sys_rdata <= {                dna_value[32-1: 0]}; end
    20'h00008: begin sys_ack <= sys_en;  sys_rdata <= {{64- 57{1'b0}}, dna_value[57-1:32]}; end
    20'h0000c: begin sys_ack <= sys_en;  sys_rdata <= {{32-  1{1'b0}}, digital_loop      }; end

   // Enable read for new registers - you read all the time
   20'h00100: begin sys_ack <= sys_en;  sys_rdata <= custom_reg_0;  end
   20'h00104: begin sys_ack <= sys_en;  sys_rdata <= custom_reg_1;  end
   20'h00108: begin sys_ack <= sys_en;  sys_rdata <= custom_reg_2;  end
   20'h0010c: begin sys_ack <= sys_en;  sys_rdata <= custom_reg_3;  end

  default: begin sys_ack <= sys_en;  sys_rdata <=  32'h0                                ; end
  endcase
end

endmodule
```

Figure 31: Enable read function of new registers

Save the changes in modified **red_pitaya_top.sv** and modified **red_pitaya_hk.v** files and compile the code. Copy the new bitstream to the SD card and run an oscilloscope application.

In parallel also connect to your board using SSH connection to get an terminal/console access to the STEMlab board. With latest OS when connecting with SSH you don't need to use IP of your board just

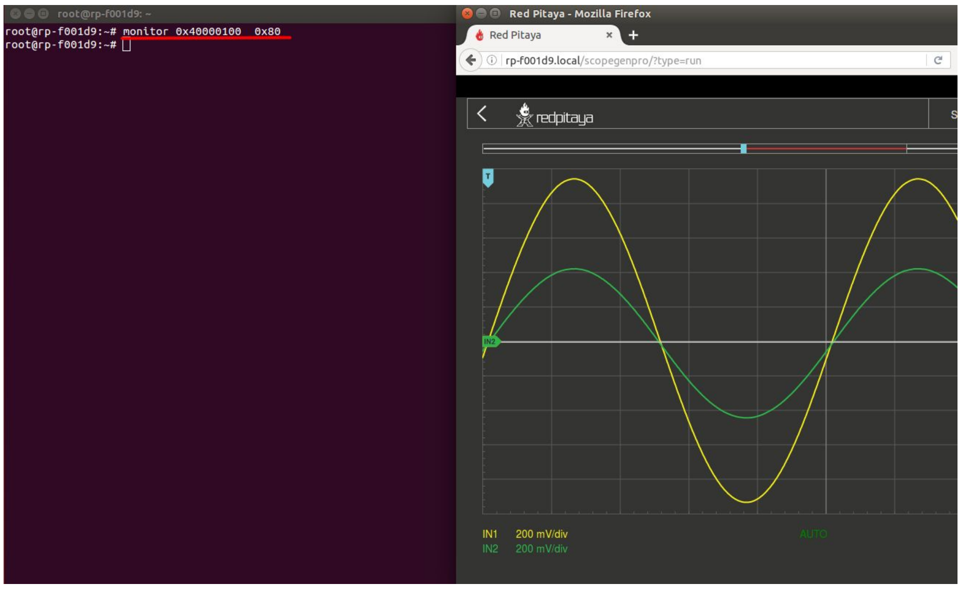
```
ssh root@rp-xxxxxx.local
```

where **xxxxxx** are last 6 character of the MAC address of your SEMlab board. (MAC address is written on ethernet connector). When SSH connection is running in the terminal of the STEMlab board run the **monitor** command and write some value in our new register.

Example: If we want to multiply IN1 value by factor 0.5 we need to write 128(0x80 in hex) in custum_reg_0. Using monitor command it looks like this

```
monitor 0x40000100  0x80
```

0x40000100 -> is the system address of our **custum_reg_0**
(in the code we defined address to be 00100. But we have an address offset of 0x400)



```
root@rp-f001d9:~# monitor 0x40000100  0x80
root@rp-f001d9:~# 
```

Figure 32: Testing the modified code using monitor command and Oscilloscope application

%code
DELOJOČA TESTNA CODA V EDA PLYGROUND
System verilog
Synopsis VCS 2015.10 Simulator

```
module MULTIPLY;

  //logic clk;
  //initial clk = 1'b1;
  //always #8 clk = ~clk;

  // scale factor
  logic unsigned [7:0]  a = 254;
      //podatki iz adc_dat
  logic signed   [13:0] b = 100;


  logic signed   [31:0] tmp;
  logic signed   [31:0] tmp_2;

  assign tmp = (a*b)>>>8;
  assign tmp_2=($signed({1'b0,a})*b)>>>8;

  //always @ (posedge clk)
  //begin
  //  tmp <= (a*b)>>8;
  initial
  begin
      #1 $display("DEC: b = %d", b, " BIN: b = %b", b);

      #1 $display("DEC: tmp = %d", tmp ," BIN: tmp = %b", tmp);

      #1 $display("DEC: tmp_2 = %d", $signed(tmp_2[14:0]) ," BIN: tmp_2 =
%b", tmp_2);
  end
  //end
  //initial #1000 $finish();


 endmodule
```